

Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

February 11, 2016

Contents

1	Transitions	5
1.1	More theorems about Closures	5
1.2	Full Transitions	6
1.3	Well-Foundedness and Full Transitions	7
1.4	More Well-Foundedness	8
2	Various Lemmas	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	33
7.2.1	Invariant while lifting of the rewriting relation	33
7.2.2	Invariant after all rewriting	34
8	Rewrite Rules	36
8.1	Elimination of the equivalences	36
8.2	Eliminate Implication	37
8.3	Eliminate all the True and False in the formula	39
8.4	PushNeg	45
8.5	Push inside	50
8.5.1	Only one type of connective in the formula (+ not)	59

8.5.2	Push Conjunction	63
8.5.3	Push Disjunction	63
9	The full transformations	64
9.1	Abstract Property characterizing that only some connective are inside the others	64
9.1.1	Definition	64
9.2	Conjunctive Normal Form	67
9.2.1	Full CNF transformation	67
9.3	Disjunctive Normal Form	68
9.3.1	Full DNF transform	68
10	More aggressive simplifications: Removing true and false at the beginning	69
10.1	Transformation	69
10.2	More invariants	70
10.3	The new CNF and DNF transformation	75
11	Partial Clausal Logic	75
11.1	Clauses	76
11.2	Partial Interpretations	76
11.2.1	Consistency	76
11.2.2	Atoms	76
11.2.3	Totality	78
11.2.4	Interpretations	80
11.2.5	Satisfiability	82
11.2.6	Entailment for Multisets of Clauses	83
11.2.7	Tautologies	85
11.2.8	Entailment for clauses and propositions	86
11.3	Subsumptions	91
11.4	Removing Duplicates	93
11.5	Set of all Simple Clauses	93
11.6	Experiment: Expressing the Entailments as Locales	99
11.7	Entailment to be extended	100
12	Resolution	101
12.1	Simplification Rules	101
12.2	Unconstrained Resolution	102
12.2.1	Subsumption	103
12.3	Inference Rule	103
12.4	Lemma about the simplified state	118
12.5	Resolution and Invariants	121
12.5.1	Invariants	121
12.5.2	well-foundedness if the relation	127
13	Partial Clausal Logic	143
13.1	Marked Literals	143
13.1.1	Definition	143
13.1.2	Entailment	144
13.1.3	Defined and undefined literals	146
13.2	Backtracking	147
13.3	Decomposition with respect to the marked literals	147

13.4	Negation of Clauses	154
13.5	Other	158
14	NOT's CDCL	159
14.1	Auxiliary Lemmas and Measure	160
14.2	Initial definitions	163
14.2.1	The state	163
14.2.2	Definition of the operation	166
14.3	DPLL with backjumping	167
14.3.1	Definition	168
14.3.2	Basic properties	168
14.3.3	Termination	171
14.3.4	Normal Forms	176
14.4	CDCL	183
14.4.1	Learn and Forget	183
14.4.2	Definition of CDCL	184
14.5	CDCL with invariant	187
14.6	Termination	193
14.6.1	Restricting learn and forget	193
14.7	CDCL with restarts	204
14.7.1	Definition	204
14.7.2	Increasing restarts	205
14.8	Merging backjump and learning	212
14.8.1	Instantiations	224
15	DPLL as an instance of NOT	239
15.1	DPLL with simple backtrack	239
15.2	Adding restarts	244
16	DPLL	244
16.1	Rules	244
16.2	Invariants	245
16.3	Termination	253
16.4	Final States	255
16.5	Link with NOT's DPLL	257
16.5.1	Level of literals and clauses	258
16.5.2	Properties about the levels	262
17	Weidenbach's CDCL	265
17.1	The State	265
17.2	Special Instantiation: using Triples as State	271
17.3	CDCL Rules	271
17.4	Invariants	277
17.4.1	Properties of the trail	277
17.4.2	Better-Suited Induction Principle	281
17.4.3	Compatibility with $op \sim$	285
17.4.4	Conservation of some Properties	287
17.4.5	Learned Clause	288
17.4.6	No alien atom in the state	289
17.4.7	No duplicates all around	292

17.4.8	Conflicts and co	293
17.4.9	Putting all the invariants together	301
17.4.10	No tautology is learned	304
17.5	CDCL Strong Completeness	305
17.6	Higher level strategy	307
17.6.1	Definition	307
17.6.2	Invariants	309
17.6.3	Literal of highest level in conflicting clauses	315
17.6.4	Literal of highest level in marked literals	318
17.6.5	Strong completeness	327
17.6.6	No conflict with only variables of level less than backtrack level	333
17.6.7	Final States are Conclusive	344
17.7	Termination	350
17.8	No Relearning of a clause	351
17.9	Decrease of a measure	366
18	Simple Implementation of the DPLL and CDCL	373
18.1	Common Rules	373
18.1.1	Propagation	373
18.1.2	Unit propagation for all clauses	374
18.1.3	Decide	375
18.2	Simple Implementation of DPLL	376
18.2.1	Combining the propagate and decide: a DPLL step	376
18.2.2	Adding invariants	378
18.2.3	Code export	385
18.3	CDCL Implementation	387
18.3.1	Definition of the rules	387
18.3.2	Propagate	389
18.3.3	Code generation	400
19	Link between Weidenbach's and NOT's CDCL	413
19.1	Inclusion of the states	413
19.2	Additional Lemmas between NOT and W states	417
19.3	More lemmas conflict-propagate and backjumping	418
19.3.1	Termination	418
19.3.2	More backjumping	419
19.4	CDCL FW	431
19.5	FW with strategy	441
19.5.1	The intermediate step	441
19.6	Adding Restarts	477
20	Incremental SAT solving	487
21	2-Watched-Literal	499
21.1	Datastructure and Access Functions	499
21.2	Invariants	500
21.3	Abstract 2-WL	508
21.4	Instanciación of the previous locale	510
21.5	Interpretation for <i>cdcl_W-ops.cdcl_W</i>	519
21.5.1	Direct Interpretation	519

21.5.2 Opaque Type with Invariant	519
21.6 We can now define the rules of the calculus	535
theory <i>Wellfounded-More</i>	
imports <i>Main</i>	

begin

1 Transitions

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

1.1 More theorems about Closures

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

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

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*: $rtrancpl r a b \iff (a = b \vee trancpl r a b)$

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

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

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

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

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

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

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: rtrancp-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-rtrancp-relation*: $\neg \text{full1}\ R^{**}\ a\ b$

by (*meson full1-def rtrancp.rtrancp-refl*)

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

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

lemma *full1-trancp-relation-full*:

$\text{full1}\ R^{++}\ a\ b \longleftrightarrow \text{full1}\ R\ a\ b$
by (*metis converse-trancpE full1-def reflcp-trancp rtrancpD rtrancp-idemp rtrancp-reflcp*
trancp.r-into-trancp trancp-into-rtrancp)

lemma *full-trancp-relation-full*:

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

lemma *rtrancp-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 rtrancp.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-trancp-relation-full*)

then have *?thesis*

using $f1$ **by** *blast* }

then have *?thesis*

by *auto* }

then show *?thesis*

by *fastforce*

qed

lemma *trancp-full1-full1*:

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

by (*metis full1-def rtrancp-full1-eq-or-full1 trancp-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  $wf: wf \{ (x, y). R y x \}$ 
  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: $wf \text{ ?}r = (\neg (\exists f. \forall i. (f (\text{Suc } i), f i) \in \text{?}r)), \llbracket wf \text{ ?}r; \bigwedge k. (\text{?}f (\text{Suc } k), \text{?}f k) \notin \text{?}r \implies \text{?thesis} \rrbracket \implies \text{?thesis}$

```

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

```

```

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

```

```

lemma wf-if-measure-f:
  assumes  $wf r$ 
  shows  $wf \{ (b, a). (f b, f a) \in r \}$ 

```


using *assms* by (*metis inv-image-def wf-inv-image*)

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

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

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

proof –

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

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

using *wf-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} <*>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  $\Rightarrow$  bool and x :: 'c  $\times$  'a
        assume H:  $\forall x. (\forall y. (y, x) \in \{(M', N'), (M, N)). R N' y\} \longrightarrow P y) \longrightarrow P x$ 
        obtain a b where x: x = (a, b) by (cases x)
        have P: P x = (P o ( $\lambda(a, b). (b, a)$ )) (b, a)
          unfolding x by auto
        show P x
          using wf[of P o ( $\lambda(a, b). (b, a)$ )] apply rule
          using H apply simp
          unfolding P by blast
      qed
  qed

lemma wf-if-measure-f-notation2:
  assumes wf r
  shows wf {(b, h a)|b a. (f b, f (h a))  $\in$  r}
  apply (rule wf-subset)
  using wf-if-measure-f[OF assms, of f] by auto

lemma wf-wf-if-measure'-notation2:
  assumes wf r and H: ( $\bigwedge x y. P x \implies g x y \implies (f y, f (h x)) \in r$ )
  shows wf {(y, h x)| y x. P x  $\wedge$  g x y}
  proof -
    have wf {(b, h a)|b a. (f b, f (h a))  $\in$  r} using assms(1) wf-if-measure-f-notation2 by auto
    then have wf {(b, h a)|b a. P a  $\wedge$  g a b  $\wedge$  (f b, f (h a))  $\in$  r}
      using wf-subset[of - {(b, h a)| b a. P a  $\wedge$  g a b  $\wedge$  (f b, f (h a))  $\in$  r}] by auto
    moreover have {(b, h a)|b a. P a  $\wedge$  g a b  $\wedge$  (f b, f (h a))  $\in$  r}
       $\subseteq$  {(b, h a)|b a. (f b, f (h a))  $\in$  r} by auto
    moreover have {(b, h a)|b a. P a  $\wedge$  g a b  $\wedge$  (f b, f (h a))  $\in$  r} = {(b, h a)|b a. P a  $\wedge$  g a b}
      using H by auto
    ultimately show ?thesis using wf-subset by simp
  qed

end
theory List-More
imports Main

```

begin

2 Various Lemmas

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

thm *nat-less-induct*

lemma *nat-less-induct-case*[*case-names 0 Suc*]:

assumes

$P\ 0$ **and**

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

shows $P\ n$

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

by (*case-tac n*) (*auto intro: assms*)

This is only proved in simple cases by auto. In assumptions, nothing happens, and $?P$ (*if ?Q then ?x else ?y*) = $(\neg (?Q \wedge \neg ?P\ ?x \vee \neg ?Q \wedge \neg ?P\ ?y))$ can blow up goals (because of other if expression).

lemma *if-0-1-ge-0[simp]*:

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

by *auto*

Bounded function have not been defined in Isabelle.

definition *bounded* **where**

$\text{bounded } f \longleftrightarrow (\exists b. \forall n. f\ n \leq b)$

abbreviation *unbounded* :: $('a \Rightarrow 'b::\text{ord}) \Rightarrow \text{bool}$ **where**

$\text{unbounded } f \equiv \neg \text{bounded } f$

lemma *not-bounded-nat-exists-larger*:

fixes $f :: \text{nat} \Rightarrow \text{nat}$

assumes *unbound*: $\text{unbounded } f$

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

proof (*rule ccontr*)

assume $H: \neg ?thesis$

have *finite* $\{f\ n \mid n. n \leq n_0\}$

by *auto*

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

apply (*case-tac n* $n \leq n_0$)

apply (*metis* (*mono-tags, lifting*) *Max-ge Un-insert-right* $\langle \text{finite } \{f\ n \mid n. n \leq n_0\} \rangle$

finite-insert insertCI mem-Collect-eq sup-bot.right-neutral)

by (*metis* (*no-types, lifting*) H *Max-less-iff Un-insert-right* $\langle \text{finite } \{f\ n \mid n. n \leq n_0\} \rangle$

finite-insert insertI1 insert-not-empty leI sup-bot.right-neutral)

then show *False*

using *unbound unfolding bounded-def* **by** *auto*

qed

lemma *bounded-const-product*:

fixes $k :: \text{nat}$ **and** $f :: \text{nat} \Rightarrow \text{nat}$

assumes $k > 0$

shows $\text{bounded } f \longleftrightarrow \text{bounded } (\lambda i. k * f\ i)$

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

```

3 More List

3.1 *upt*

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

```

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

```

```

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

```

```

declare upt.simps(2)[simp del]

```

```

lemma
  assumes  $i \leq n - m$ 
  shows 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 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.. $< m + \text{length } A$ ] \text{ and } B = [m + \text{length } A.. $< n$ ]$ 
proof -
  have take (length A) (A @ B) = A by auto
  moreover
    have length A  $\leq n - m$  using assms linear calculation by fastforce
    then have take (length A) [m.. $< n$ ] = [m.. $< m + \text{length } A$ ] 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 $A @ i \# B = [m + 1..<n]$ **by** (*metis append-Cons upt-eq-Cons-conv*)

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

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

qed

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

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

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

proof –

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

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

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

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

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

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

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

qed

```

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

```

```

lemma upt-decomp-lt:
  assumes H:  $xs @ i \# ys @ j \# zs = [m ..<n]$ 
  shows  $i < j$ 
proof -
  have xs:  $xs = [m ..<i]$  and ys:  $ys = [\text{Suc } i ..<j]$  and zs:  $zs = [\text{Suc } j ..<n]$ 
  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 propo =
  FT | FF | FVar 'v | FNot 'v propo | FAnd 'v propo 'v propo | FOr 'v propo 'v propo
  | FImp 'v propo 'v propo | FEq 'v propo 'v propo

```

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

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

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

abbreviation *nullary-connective* $\equiv \{CF\} \cup \{CT\} \cup \{CVar\ x \mid x. True\}$

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

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 COOr ($\varphi\# [\psi]$) = *FOr* $\varphi\ \psi$ |
conn CImp ($\varphi\# [\psi]$) = *FImp* $\varphi\ \psi$ |
conn CEq ($\varphi\# [\psi]$) = *FEq* $\varphi\ \psi$ |
conn - - = *FF*

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

lemma *connective-cases-arity*:

assumes *nullary*: $\bigwedge x. c = CT \vee c = CF \vee c = CVar\ x \implies P$
and *binary*: $c \in \text{binary-connectives} \implies P$
and *unary*: $c = CNot \implies P$
shows P
using *assms* **by** (*case-tac c, auto simp add: binary-connectives-def*)

lemma *connective-cases-arity-2*[*case-names nullary unary binary*]:

assumes *nullary*: $c \in \text{nullary-connective} \implies P$
and *unary*: $c = CNot \implies P$
and *binary*: $c \in \text{binary-connectives} \implies P$
shows P
using *assms* **by** (*case-tac c, auto simp add: binary-connectives-def*)

Our previous definition is not necessary correct (connective and list of arguments) , so we define an inductive predicate.

inductive *wf-conn* :: 'v connective \Rightarrow 'v propo list \Rightarrow bool **for** *c* :: 'v connective **where**

wf-conn-nullary[simp]: (*c* = *CT* \vee *c* = *CF* \vee *c* = *CVar* *v*) \Longrightarrow *wf-conn* *c* [] |

wf-conn-unary[simp]: *c* = *CNot* \Longrightarrow *wf-conn* *c* [*ψ*] |

wf-conn-binary[simp]: *c* \in *binary-connectives* \Longrightarrow *wf-conn* *c* (*ψ* # *ψ'* # [])

thm *wf-conn.induct*

lemma *wf-conn-induct*[consumes 1, case-names *CT CF CVar CNot COr CAnd CImp CEq*]:

assumes *wf-conn* *c* *x* **and**

($\bigwedge v. c = CT \Longrightarrow P []$) **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 add: binary-connectives-def*)

4.2 properties of the abstraction

First we can define simplification rules.

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

wf-conn *CT* *l* \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* (*ξ* @ *φ* # *ξ'*) $\longleftrightarrow \xi = [] \wedge \xi' = []$

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 $\Longrightarrow (\exists\ a\ b. l = a \# b \# [])$

apply (*induct* *l*, *auto*)

by (case-tac l, auto)

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

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

fixes $l :: 'v \text{ propo list}$

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

shows $\text{length } l = 2 \longleftrightarrow \text{wf-conn } c \ l$

proof

assume $\text{length } l = 2$

thus *wf-conn* $c \ l$ **using** *wf-conn-binary list-length2-decomp* **using** *conn* **by** *metis*

next

assume *wf-conn* $c \ l$

thus $\text{length } l = 2$ (**is** ?*P* l)

proof (*cases* rule: *wf-conn.induct*)

case *wf-conn-nullary*

thus ?*P* [] **using** *conn* *binary-connectives-def*

using *connective.distinct*(11) *connective.distinct*(13) *connective.distinct*(9) **by** *blast*

next

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

case *wf-conn-unary*

thus ?*P* [ψ] **using** *conn* *binary-connectives-def*

using *connective.distinct* **by** *blast*

next

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

show ?*P* [ψ, ψ'] **by** *auto*

qed

qed

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

fixes $l :: 'v \text{ propo list}$

shows *wf-conn* *CNot* $l \longleftrightarrow \text{length } l = 1$

apply *auto*

apply (*metis* *append-Nil* *connective.distinct*(5,17,27) *length-Cons* *list.size*(3) *wf-conn.simps* *wf-conn-list-decomp*(4))

by (*simp* add: *length-Suc-conv* *wf-conn.simps*)

Decomposing the Not into an element is moreover very useful.

lemma *wf-conn-Not-decomp*:

fixes $l :: 'v \text{ propo list}$ **and** $a :: 'v$

assumes *corr*: *wf-conn* *CNot* l

shows $\exists a. l = [a]$

by (*metis* (*no-types*, *lifting*) *One-nat-def* *Suc-length-conv* *corr* *length-0-conv* *wf-conn-not-list-length*)

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

lemma *wf-conn-no-arity-change*:

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

proof –

{

fix $l \ l'$

have $\text{length } l = \text{length } l' \implies \text{wf-conn } c \ l \implies \text{wf-conn } c \ l'$

apply (*cases* $c \ l$ rule: *wf-conn.induct*, *auto*)

by (*metis* *wf-conn-bin-list-length*)

}

thus $\text{length } l = \text{length } l' \implies \text{wf-conn } c \ l = \text{wf-conn } c \ l'$ **by** *metis*

qed

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

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

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

4.3 Subformulas and properties

A characterization using sub-formulas is interesting for rewriting: we will define our relation on the sub-term level, and then lift the rewriting on the term-level. So the rewriting takes place on a subformula.

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

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

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:

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

lemma subformula-in-binary-conn:

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

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

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

proof –

have a : $\text{wf-conn } c\ (f\# [g])$ **using** conn wf-conn-binary binary-connectives-def **by** auto

moreover **have** b : $f \preceq f$ **using** subformula-refl **by** auto

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

by (metis append-Nil in-set-conv-decomp subformula-into-subformula)

next

have a : $\text{wf-conn } c\ ([f] @ [g])$ **using** conn wf-conn-binary binary-connectives-def **by** auto

moreover **have** b : $g \preceq g$ **using** subformula-refl **by** auto

ultimately show $g \preceq \text{conn } c\ [f, g]$ **using** subformula-into-subformula **by** force

qed

lemma subformula-trans:

$\psi \preceq \psi' \implies \varphi \preceq \psi \implies \varphi \preceq \psi'$

apply (induct ψ' rule: subformula.inducts)

by (auto simp add: subformula-into-subformula)

lemma subformula-leaf:

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

assumes incl: $\varphi \preceq \psi$

and simple: $\psi = FT \vee \psi = FF \vee \psi = FVar\ x$

shows $\varphi = \psi$

using incl simple

by (induct rule: subformula.induct, auto simp add: wf-conn-list)

lemma subformula-not-incl-eq:

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

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

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

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

using assms **apply** (induction conn $c\ l$ rule: subformula.induct, auto)

using conn-inj **by** blast

lemma wf-subformula-conn-cases:

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

apply standard

using subformula-not-incl-eq **apply** metis

by (auto simp add: subformula-into-subformula)

lemma subformula-decomp-explicit[simp]:

$\varphi \preceq FAnd\ \psi\ \psi' \longleftrightarrow (\varphi = FAnd\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$ (**is** $?P\ FAnd$)

$\varphi \preceq FOr\ \psi\ \psi' \longleftrightarrow (\varphi = FOr\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$

$\varphi \preceq FEq\ \psi\ \psi' \longleftrightarrow (\varphi = FEq\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$

$\varphi \preceq FImp\ \psi\ \psi' \longleftrightarrow (\varphi = FImp\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$

```

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

```

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

```

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

```

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

by (cases φ) force+

lemma subformula-conn-decomp[simp]:

```

wf-conn  $c l \implies \varphi \preceq \text{conn } c l \longleftrightarrow (\varphi = \text{conn } c l \vee (\exists \psi \in \text{set } l. \varphi \preceq \psi))$ 
apply auto

```

proof –

```

{
  fix  $\xi$ 
  have  $\varphi \preceq \xi \implies \xi = \text{conn } c l \implies \text{wf-conn } c l \implies \forall x::'a \text{ propo} \in \text{set } l. \neg \varphi \preceq x \implies \varphi = \text{conn } c l$ 
    apply (induct rule: subformula.induct)
    apply simp
    using conn-inj by blast
}

```

moreover assume wf-conn $c l$ **and** $\varphi \preceq \text{conn } c l$ **and** $\forall x::'a \text{ propo} \in \text{set } l. \neg \varphi \preceq x$

ultimately show $\varphi = \text{conn } c l$ **by** metis

next

fix ψ

assume wf-conn $c l$ **and** $\psi \in \text{set } l$ **and** $\varphi \preceq \psi$

thus $\varphi \preceq \text{conn } c l$ **using** wf-subformula-conn-cases **by** blast

qed

```

lemma subformula-leaf-explicit[simp]:
   $\varphi \preceq FT \iff \varphi = FT$ 
   $\varphi \preceq FF \iff \varphi = FF$ 
   $\varphi \preceq FVar\ x \iff \varphi = FVar\ x$ 
  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  $\varphi$ ) = vars-of-prop  $\varphi$  |
  vars-of-prop (FAnd  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi$   $\cup$  vars-of-prop  $\psi$  |
  vars-of-prop (FOr  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi$   $\cup$  vars-of-prop  $\psi$  |
  vars-of-prop (FImp  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi$   $\cup$  vars-of-prop  $\psi$  |
  vars-of-prop (FEq  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi$   $\cup$  vars-of-prop  $\psi$ 

```

```

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
  hence False using corr incl by auto
  thus 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
  hence  $\psi = a \vee \psi = b$  using incl by auto
  thus 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 = [ $\psi$ ] using corr c incl split-list by force
  thus 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
hence 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 $?R$ **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** $prop\text{-}size\ \varphi = card\ (pos\ \varphi)$

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

fixes $\varphi :: 'v\ propo$

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

unfolding *prop-size-def* **apply** (*induct* φ , *auto simp add: cons-inject finite-inj-comp-set finite-pos*)

proof –

fix $\varphi1\ \varphi2 :: 'v\ propo$

assume *IH1*: $card\ (vars\text{-}of\text{-}prop\ \varphi1) \leq card\ (pos\ \varphi1)$

and *IH2*: $card\ (vars\text{-}of\text{-}prop\ \varphi2) \leq card\ (pos\ \varphi2)$

let $?L = \{L \# p \mid p. p \in pos\ \varphi1\}$

let $?R = \{R \# p \mid p. p \in pos\ \varphi2\}$

have $card\ (?L \cup ?R) = card\ ?L + card\ ?R$

using *card-seperate finite-pos* **by** *blast*

moreover have $\dots = card\ (pos\ \varphi1) + card\ (pos\ \varphi2)$

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

moreover have $\dots \geq card\ (vars\text{-}of\text{-}prop\ \varphi1) + card\ (vars\text{-}of\text{-}prop\ \varphi2)$ **using** *IH1 IH2* **by** *arith*

hence $\dots \geq card\ (vars\text{-}of\text{-}prop\ \varphi1 \cup vars\text{-}of\text{-}prop\ \varphi2)$ **using** *card-Un-le le-trans* **by** *blast*

ultimately

show $card\ (vars\text{-}of\text{-}prop\ \varphi1 \cup vars\text{-}of\text{-}prop\ \varphi2) \leq Suc\ (card\ (?L \cup ?R))$

$card\ (vars\text{-}of\text{-}prop\ \varphi1 \cup vars\text{-}of\text{-}prop\ \varphi2) \leq Suc\ (card\ (?L \cup ?R))$

$card\ (vars\text{-}of\text{-}prop\ \varphi1 \cup vars\text{-}of\text{-}prop\ \varphi2) \leq Suc\ (card\ (?L \cup ?R))$

$card\ (vars\text{-}of\text{-}prop\ \varphi1 \cup vars\text{-}of\text{-}prop\ \varphi2) \leq Suc\ (card\ (?L \cup ?R))$

by *auto*

qed

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

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

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

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

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

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

$\implies path\text{-}to\ (R \# p)\ (conn\ c\ (\psi \# \varphi \# []))\ \varphi'$

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

lemma *path-to-subformula*:

$path\text{-}to\ p\ \varphi\ \varphi' \implies \varphi' \preceq \varphi$

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 $\varphi\ \varphi' :: 'v\ propo$

```

shows  $\varphi' \preceq \varphi \implies \exists p. \text{path-to } p \varphi \varphi'$ 
proof (induct rule: subformula.induct)
  case subformula-refl
  have path-to []  $\varphi' \varphi'$  by auto
  thus  $\exists p. \text{path-to } p \varphi' \varphi'$  by metis
next
case (subformula-into-subformula  $\psi \ l \ c$ )
note wf = this(2) and IH = this(4) and  $\psi = \text{this}(1)$ 
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
  hence False using subformula-into-subformula by auto
  hence  $\exists p. \text{path-to } p (\text{conn } c \ l) \varphi'$  by blast
}
moreover {
  assume c: c = CNot
  hence l = [ $\psi$ ] using wf  $\psi$  wf-conn-Not-decomp by fastforce
  hence path-to (L # p) (conn c l)  $\varphi'$  by (metis c wf-conn-unary p path-to-l)
  hence  $\exists p. \text{path-to } p (\text{conn } c \ l) \varphi'$  by blast
}
moreover {
  assume c: c  $\in$  binary-connectives
  obtain a b where ab: [ $a, b$ ] = l using subformula-into-subformula c wf-conn-bin-list-length
    list-length2-decomp by metis
  hence a =  $\psi \vee b = \psi$  using  $\psi$  by auto
  hence 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)
  hence  $\exists p. \text{path-to } p (\text{conn } c \ l) \varphi'$  by blast
}
ultimately show  $\exists p. \text{path-to } p (\text{conn } c \ l) \varphi'$  using connective-cases-arity by metis
qed

```

```

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

```

5 Semantics over the syntax

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

```

fun eval :: ('v  $\Rightarrow$  bool)  $\Rightarrow$  'v propo  $\Rightarrow$  bool (infix  $\models$  50) where
 $\mathcal{A} \models FT = \text{True}$  |
 $\mathcal{A} \models FF = \text{False}$  |
 $\mathcal{A} \models FVar \ v = (\mathcal{A} \ v)$  |
 $\mathcal{A} \models FNot \ \varphi = (\neg(\mathcal{A} \models \varphi))$  |

```


$\mathcal{A} \models FAnd \ \varphi_1 \ \varphi_2 = (\mathcal{A} \models \varphi_1 \wedge \mathcal{A} \models \varphi_2) \mid$
 $\mathcal{A} \models FOr \ \varphi_1 \ \varphi_2 = (\mathcal{A} \models \varphi_1 \vee \mathcal{A} \models \varphi_2) \mid$
 $\mathcal{A} \models FImp \ \varphi_1 \ \varphi_2 = (\mathcal{A} \models \varphi_1 \longrightarrow \mathcal{A} \models \varphi_2) \mid$
 $\mathcal{A} \models FEq \ \varphi_1 \ \varphi_2 = (\mathcal{A} \models \varphi_1 \longleftrightarrow \mathcal{A} \models \varphi_2)$

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

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

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

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

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

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

next

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

proof (*rule ccontr*)

assume $\neg \varphi \models_f \psi$

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

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

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

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

qed

qed

A shorter proof:

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

by (*simp add: evalf-def*)

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

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

lemma *same-over-set-eval*:

assumes *same-over-set* $A\ B$ (*vars-of-prop* φ)

shows $A \models \varphi \longleftrightarrow B \models \varphi$

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

end

theory *Prop-Abstract-Transformation*

imports *Main Prop-Logic Wellfounded-More*

begin

This file is devoted to abstract properties of the transformations, like consistency preservation and lifting from terms to proposition.

6 Rewrite systems and properties

6.1 Lifting of rewrite rules

We can lift a rewrite relation r over a full formula: the relation r works on terms, while *propo-rew-step* works on formulas.

inductive *propo-rew-step* :: ($'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$) $\Rightarrow 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$

for $r :: 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$ **where**

global-rel: $r\ \varphi\ \psi \Longrightarrow \text{propo-rew-step}\ r\ \varphi\ \psi$ |

propo-rew-one-step-lift: $\text{propo-rew-step}\ r\ \varphi\ \varphi' \Longrightarrow \text{wf-conn}\ c\ (\psi s\ @\ \varphi\ \# \psi s') \Longrightarrow \text{propo-rew-step}\ r\ (\text{conn}\ c\ (\psi s\ @\ \varphi\ \# \psi s'))\ (\text{conn}\ c\ (\psi s\ @\ \varphi' \# \psi s'))$

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

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 ψ' .

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

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

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

proof (*induct* φ *rule*: *subformula.induct*)

case *subformula-refl*

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

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

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

next

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

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

then obtain φ' where $*$: $\psi' \preceq \varphi' \wedge \text{propo-rew-step } r \psi'' \varphi'$ by *metis*
 moreover obtain $\xi \xi' :: 'v \text{ propo list where}$
 $l: l = \xi @ \psi'' \# \xi'$ using *List.split-list* ψ'' by *metis*
 ultimately have $\text{propo-rew-step } r (\text{conn } c \ l) (\text{conn } c \ (\xi @ \varphi' \# \xi'))$
 using *propo-rew-step.intros(2)* wf by *metis*
 moreover have $\psi' \preceq \text{conn } c \ (\xi @ \varphi' \# \xi')$
 using $wf * wf\text{-conn-no-arity-change}$ *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 $wf: wf\text{-conn } c \ l$ and $wf': wf\text{-conn } c \ l'$
 and *same*: $\forall n. (A \models l ! n \longleftrightarrow (A \models l' ! n))$
 shows $(A \models \text{conn } c \ l) = (A \models \text{conn } c \ l')$
proof (*cases c rule: connective-cases-arity-2*)
 case *nullary*
 thus $(A \models \text{conn } c \ l) \longleftrightarrow (A \models \text{conn } c \ l')$ using $wf \ wf'$ by *auto*
 next
 case *unary note* $c = \text{this}$
 then obtain a where $l: l = [a]$ using *wf-conn-Not-decomp* wf by *metis*
 obtain a' where $l': l' = [a']$ using *wf-conn-Not-decomp* $wf' \ c$ by *metis*
 have $A \models a \longleftrightarrow A \models a'$ using $l \ l'$ by (*metis* *nth-Cons-0* *same*)
 thus $A \models \text{conn } c \ l \longleftrightarrow A \models \text{conn } c \ l'$ using $l \ l' \ c$ by *auto*
 next
 case *binary note* $c = \text{this}$
 then obtain $a \ b$ where $l: l = [a, b]$
 using *wf-conn-bin-list-length* *list-length2-decomp* wf by *metis*
 obtain $a' \ b'$ where $l': l' = [a', b']$
 using *wf-conn-bin-list-length* *list-length2-decomp* $wf' \ c$ by *metis*

 have $p: A \models a \longleftrightarrow A \models a' \ A \models b \longleftrightarrow A \models b'$
 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* $[] \varphi \varphi$ by *auto*
 moreover have *replace-at* $[] \varphi \psi = \psi$ by *auto*
 ultimately show *?case* by *metis*
 next
 case (*propo-rew-one-step-lift* $\varphi \varphi' \ c \ \xi \ \xi'$) **note** $\text{rel} = \text{this}(1)$ and $\text{IH0} = \text{this}(2)$ and $\text{corr} = \text{this}(3)$

obtain $\psi \psi' p$ where *IH*: $r \psi \psi' \wedge \text{path-to } p \varphi \psi \wedge \text{replace-at } p \varphi \psi' = \varphi'$ using *IH0* by *metis*

```

{
  fix x :: 'v
  assume c = CT  $\vee$  c = CF  $\vee$  c = CVar x
  hence False using corr by auto
  hence  $\exists \psi \psi' p. r \psi \psi' \wedge \text{path-to } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \psi$ 
     $\wedge \text{replace-at } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \psi' = \text{conn } c (\xi @ (\varphi' \# \xi'))$ 
    by fast
}
moreover {
  assume c: c = CNot
  hence empty:  $\xi = [] \ \xi' = []$  using corr by auto
  have path-to (L#p) (conn c ( $\xi @ (\varphi \# \xi')$ ))  $\psi$ 
    using c empty IH wf-conn-unary path-to-l by fastforce
  moreover have replace-at (L#p) (conn c ( $\xi @ (\varphi \# \xi')$ ))  $\psi' = \text{conn } c (\xi @ (\varphi' \# \xi'))$ 
    using c empty IH by auto
  ultimately have  $\exists \psi \psi' p. r \psi \psi' \wedge \text{path-to } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \psi$ 
     $\wedge \text{replace-at } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \psi' = \text{conn } c (\xi @ (\varphi' \# \xi'))$ 
    using IH by metis
}
moreover {
  assume c: c  $\in$  binary-connectives
  have length ( $\xi @ \varphi \# \xi'$ ) = 2 using wf-conn-bin-list-length corr c by metis
  hence length  $\xi$  + length  $\xi' = 1$  by auto
  hence ld: (length  $\xi = 1 \wedge \text{length } \xi' = 0$ )  $\vee$  (length  $\xi = 0 \wedge \text{length } \xi' = 1$ ) by arith
  obtain a b where ab: ( $\xi = [] \wedge \xi' = [b]$ )  $\vee$  ( $\xi = [a] \wedge \xi' = []$ )
    using ld by (case-tac  $\xi$ , case-tac  $\xi'$ , auto)
  {
    assume  $\varphi: \xi = [] \wedge \xi' = [b]$ 
    have path-to (L#p) (conn c ( $\xi @ (\varphi \# \xi')$ ))  $\psi$ 
      using  $\varphi$  c IH ab corr by (simp add: path-to-l)
    moreover have replace-at (L#p) (conn c ( $\xi @ (\varphi \# \xi')$ ))  $\psi' = \text{conn } c (\xi @ (\varphi' \# \xi'))$ 
      using c IH ab  $\varphi$  unfolding binary-connectives-def by auto
    ultimately have  $\exists \psi \psi' p. r \psi \psi' \wedge \text{path-to } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \psi$ 
       $\wedge \text{replace-at } p (\text{conn } c (\xi @ (\varphi \# \xi'))) \psi' = \text{conn } c (\xi @ (\varphi' \# \xi'))$ 
      using IH by metis
  }
  moreover {
    assume  $\varphi: \xi = [a] \ \xi' = []$ 
    hence path-to (R#p) (conn c ( $\xi @ (\varphi \# \xi')$ ))  $\psi$ 
      using c IH corr path-to-r corr  $\varphi$  by (simp add: path-to-r)
    moreover have replace-at (R#p) (conn c ( $\xi @ (\varphi \# \xi')$ ))  $\psi' = \text{conn } c (\xi @ (\varphi' \# \xi'))$ 
      using c IH ab  $\varphi$  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

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

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

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* \implies *all-subformula-st test-symb* *FT*
test-symb *FF* \implies *all-subformula-st test-symb* *FF*
test-symb (*FVar* *x*) \implies *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 $\varphi \implies \text{test-symb } \varphi$
unfolding *all-subformula-st-def* **by** *auto*

lemma *all-subformula-st-decomp-imp*:
 $\text{wf-conn } c \ l \implies (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$
 $\implies \text{all-subformula-st test-symb } (\text{conn } c \ l)$
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 ($\text{conn } c \ l$) $\implies \text{wf-conn } c \ l$
 $\implies (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$
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* ($\text{conn } c \ l$)
 $\longleftrightarrow (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$
using *assms* *all-subformula-st-decomp-rec* *all-subformula-st-decomp-imp* **by** *metis*

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

lemma *all-subformula-st-decomp-explicit[simp]*:
fixes $\varphi \ \psi$:: '*v* propo
shows *all-subformula-st test-symb* (*FAnd* $\varphi \ \psi$)
 $\longleftrightarrow (\text{test-symb } (\text{FAnd } \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb* (*FOr* $\varphi \ \psi$)
 $\longleftrightarrow (\text{test-symb } (\text{FOr } \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb* (*FNot* φ)

$\longleftrightarrow (\text{test-symb } (F\text{Not } \varphi) \wedge \text{all-subformula-st test-symb } \varphi)$
and $\text{all-subformula-st test-symb } (F\text{Eq } \varphi \psi)$
 $\longleftrightarrow (\text{test-symb } (F\text{Eq } \varphi \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and $\text{all-subformula-st test-symb } (F\text{Imp } \varphi \psi)$
 $\longleftrightarrow (\text{test-symb } (F\text{Imp } \varphi \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
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  $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$  and  $\text{test-symb} :: 'v \text{ propo} \Rightarrow \text{bool}$  and  $x :: 'v$ 
and  $\varphi :: 'v \text{ propo}$ 
assumes  $\text{test-symb-false-nullary}: \forall x. \text{test-symb } FF \wedge \text{test-symb } FT \wedge \text{test-symb } (FVar\ x)$ 
and  $\forall \varphi'. \varphi' \preceq \varphi \longrightarrow (\neg \text{test-symb } \varphi') \longrightarrow (\exists \psi. r\ \varphi'\ \psi)$  and
 $\neg \text{all-subformula-st test-symb } \varphi$ 
shows  $(\exists \psi\ \psi'. \psi \preceq \varphi \wedge r\ \psi\ \psi')$ 
using assms
proof (induct  $\varphi$  rule: propo-induct-arity)
  case (nullary  $\varphi\ x$ )
  thus  $\exists \psi\ \psi'. \psi \preceq \varphi \wedge r\ \psi\ \psi'$ 
    using wf-conn-nullary test-symb-false-nullary by fastforce
next
  case (unary  $\varphi$ ) note  $IH = \text{this}(1)[OF\ \text{this}(2)]$  and  $r = \text{this}(2)$  and  $nst = \text{this}(3)$  and  $\text{subf} = \text{this}(4)$ 
  from  $r\ IH\ nst$  have  $H: \neg \text{all-subformula-st test-symb } \varphi \Longrightarrow \exists \psi. \psi \preceq \varphi \wedge (\exists \psi'. r\ \psi\ \psi')$ 
    by (metis subformula-in-subformula-not subformula-refl subformula-trans)
  {
    assume  $n: \neg \text{test-symb } (FNot\ \varphi)$ 
    obtain  $\psi$  where  $r\ (FNot\ \varphi)\ \psi$  using subformula-refl r n nst by blast
    moreover have  $FNot\ \varphi \preceq FNot\ \varphi$  using subformula-refl by auto
    ultimately have  $\exists \psi\ \psi'. \psi \preceq FNot\ \varphi \wedge r\ \psi\ \psi'$  by metis
  }
  moreover {
    assume  $n: \text{test-symb } (FNot\ \varphi)$ 
    hence  $\neg \text{all-subformula-st test-symb } \varphi$ 
    using all-subformula-st-decomp-explicit(3) nst subf by blast
    hence  $\exists \psi\ \psi'. \psi \preceq FNot\ \varphi \wedge r\ \psi\ \psi'$ 
    using  $H$  subformula-in-subformula-not subformula-refl subformula-trans by blast
  }
  ultimately show  $\exists \psi\ \psi'. \psi \preceq FNot\ \varphi \wedge r\ \psi\ \psi'$  by blast
next
  case (binary  $\varphi\ \varphi1\ \varphi2$ )
  note  $IH\varphi1-0 = \text{this}(1)[OF\ \text{this}(4)]$  and  $IH\varphi2-0 = \text{this}(2)[OF\ \text{this}(4)]$  and  $r = \text{this}(4)$ 
    and  $\varphi = \text{this}(3)$  and  $le = \text{this}(5)$  and  $nst = \text{this}(6)$ 

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

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

$H': \forall (c:: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \text{propo-rew-step } r \varphi \varphi' \longrightarrow \text{wf-conn } c (\xi @ \varphi \# \xi') \\ \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi')) \text{ and}$
 $\text{full: full } (\text{propo-rew-step } r) \varphi \psi \text{ and}$
 $\text{init: all-subformula-st test-symb } \varphi$
shows $\text{all-subformula-st test-symb } \psi$
using $\text{full-propo-rew-step-inv-stay-with-inc[of } r \text{ test-symb } \varphi] \text{ assms subformula-refl by metis}$

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

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

proof –

have $\text{rel: } (\text{propo-rew-step } r)^{**} \varphi \psi$
using *full unfolding full-def* **by** *auto*
thus $\text{all-subformula-st test-symb } \psi$
using *init*
proof (*induct rule: rtrancpl-induct*)
case *base*
thus $\text{all-subformula-st test-symb } \varphi$ **by** *blast*
next
case (*step b c*)
note $\text{star} = \text{this}(1)$ **and** $\text{IH} = \text{this}(3)$ **and** $\text{one} = \text{this}(2)$ **and** $\text{all} = \text{this}(4)$
hence $\text{all-subformula-st test-symb } b$ **by** *metis*
thus $\text{all-subformula-st test-symb } c$
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** $\text{test-symb}:: 'v \text{ propo} \Rightarrow \text{bool}$ **and** $x:: 'v$
and $\varphi \psi:: 'v \text{ propo}$
assumes
 $H: \forall \varphi \psi. r \varphi \psi \longrightarrow \text{all-subformula-st test-symb } \varphi \longrightarrow \text{all-subformula-st test-symb } \psi \text{ and}$
 $H': \forall (c:: 'v \text{ connective}) l l'. \text{wf-conn } c l \longrightarrow \text{wf-conn } c l' \\ \longrightarrow (\text{test-symb } (\text{conn } c l) \longleftrightarrow \text{test-symb } (\text{conn } c l')) \text{ and}$
 $\text{full: full } (\text{propo-rew-step } r) \varphi \psi \text{ and}$
 $\text{init: all-subformula-st test-symb } \varphi$
shows $\text{all-subformula-st test-symb } \psi$

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 $\text{all-subformula-st test-symb } \psi$
using H *full init full-propo-rew-step-inv-stay* **by** *blast*
qed

end

```

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

```

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

8 Rewrite Rules

The idea of Christoph Weidenbach's book is to remove gradually the operators: first equivalencies, then implication, after that the unused true/false and finally the reorganizing the or/and. We will prove each transformation separately.

8.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

```

inductive elim-equiv :: 'v propo  $\Rightarrow$  'v propo  $\Rightarrow$  bool where
  elim-equiv[simp]: elim-equiv (FEq  $\varphi$   $\psi$ ) (FAnd (FImp  $\varphi$   $\psi$ ) (FImp  $\psi$   $\varphi$ ))

```

```

lemma elim-equiv-transformation-consistent:
  A  $\models$  FEq  $\varphi$   $\psi$   $\longleftrightarrow$  A  $\models$  FAnd (FImp  $\varphi$   $\psi$ ) (FImp  $\psi$   $\varphi$ )
by auto

```

```

lemma elim-equiv-explicit: elim-equiv  $\varphi$   $\psi$   $\Longrightarrow$   $\forall A. A \models \varphi \longleftrightarrow A \models \psi$ 
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

```

```

    ¬no-equiv (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 $\varphi \psi :: 'v$ propo

shows

```

    no-equiv (FNot  $\varphi$ )  $\longleftrightarrow$  no-equiv  $\varphi$ 
    no-equiv (FAnd  $\varphi \psi$ )  $\longleftrightarrow$  (no-equiv  $\varphi \wedge$  no-equiv  $\psi$ )
    no-equiv (FOr  $\varphi \psi$ )  $\longleftrightarrow$  (no-equiv  $\varphi \wedge$  no-equiv  $\psi$ )
    no-equiv (FImp  $\varphi \psi$ )  $\longleftrightarrow$  (no-equiv  $\varphi \wedge$  no-equiv  $\psi$ )
by (auto simp add: no-equiv-def)

```

A theorem to show the link between the rewrite relation *elim-equiv* and the function *no-equiv-symb*. This theorem is one of the assumption we need to characterize the transformation.

lemma *no-equiv-elim-equiv-step*:

fixes $\varphi :: 'v$ propo

assumes *no-equiv*: \neg no-equiv φ

shows $\exists \psi \psi'. \psi \preceq \varphi \wedge$ *elim-equiv* $\psi \psi'$

proof –

have *test-symb-false-nullary*:

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

unfolding no-equiv-def **by** auto

moreover {

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

assume $a1: \text{elim-equiv } (\text{conn } c\ l)\ \psi$

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

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

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

}

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

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

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

by (case-tac φ , auto simp add: *elim-equiv.simps*)

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

ultimately show ?thesis

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

qed

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

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

full (propo-rew-step *elim-equiv*) $\varphi\ \psi \implies$ no-equiv ψ

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* $:: 'v$ propo $\Rightarrow 'v$ propo \Rightarrow bool **where**

[simp]: *elim-imp* (FImp $\varphi\ \psi$) (FOr (FNot φ) ψ)

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

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

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

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

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

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

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

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

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

Invariant of the *elim-imp* transformation

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

lemma *elim-imp-inv*:
fixes $\varphi \psi :: 'v \ propo$
assumes *full (propo-rew-step elim-imp) $\varphi \psi$*
and *no-equiv 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  $\varphi :: 'v \text{ propo}$ 
  assumes no-equiv:  $\neg \text{no-imp } \varphi$ 
  shows  $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elim-imp } \psi \psi'$ 
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  $c :: 'v \text{ connective}$  and  $l :: 'v \text{ propo list}$  and  $\psi :: 'v \text{ propo}$ 
    have  $H: \text{elim-imp } (\text{conn } c \ l) \ \psi \implies \neg \text{no-imp-symb } (\text{conn } c \ l)$ 
      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  $\bigwedge \varphi. \neg \text{no-imp-symb } \varphi \implies \exists \psi. \text{elim-imp } \varphi \ \psi$ 
    apply (case-tac  $\varphi$ ) using elim-imp.simps by force+
  hence  $(\bigwedge \varphi'. \varphi' \preceq \varphi \implies \neg \text{no-imp-symb } \varphi' \implies \exists \psi. \text{elim-imp } \varphi' \ \psi)$  by force
  ultimately show ?thesis
    using no-test-symb-step-exists no-equiv test-symb-false-nullary unfolding no-imp-def by blast
qed

```

```

lemma no-imp-full-propo-rew-step-elim-imp:  $\text{full } (\text{propo-rew-step } \text{elim-imp}) \ \varphi \ \psi \implies \text{no-imp } \psi$ 
  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 (induct-tac rule: elimTB.inducts) auto
}
thus ?thesis using preserves-un-sat-def by auto
qed

```

inductive *no-T-F-symb* :: 'v propo \Rightarrow bool **where**
no-T-F-symb-comp: $c \neq CF \Rightarrow c \neq CT \Rightarrow \text{wf-conn } c \ l \Rightarrow (\forall \varphi \in \text{set } l. \varphi \neq FT \wedge \varphi \neq FF)$
 $\Rightarrow \text{no-T-F-symb } (\text{conn } c \ l)$

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

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

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]*:

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

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

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

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

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

lemma *no-T-F-symb-fnot-imp*:

$\neg \text{no-T-F-symb } (F\text{Not } \varphi) \Rightarrow \varphi = FT \vee \varphi = FF$

proof (*rule ccontr*)

assume *n*: $\neg \text{no-T-F-symb } (F\text{Not } \varphi)$

assume $\neg (\varphi = FT \vee \varphi = FF)$

hence $\forall \varphi' \in \text{set } [\varphi]. \varphi' \neq FT \wedge \varphi' \neq FF$ **by** *auto*

moreover **have** *wf-conn CNot* $[\varphi]$ **by** *simp*

ultimately **have** *no-T-F-symb* (*FNot* φ)

using *no-T-F-symb.intros* **by** (*metis* *conn.simps(4)* *connective.distinct(5,17)*)

thus *False* **using** *n* **by** *blast*

qed

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

Actually it is not possible to remove every *FT* and *FF*: if the formula is equal to true or false, we can not remove it.

inductive *no-T-F-symb-except-toplevel* **where**
no-T-F-symb-except-toplevel-true[simp]: *no-T-F-symb-except-toplevel FT* |
no-T-F-symb-except-toplevel-false[simp]: *no-T-F-symb-except-toplevel FF* |
noTrue-no-T-F-symb-except-toplevel[simp]: *no-T-F-symb $\varphi \implies no-T-F-symb-except-toplevel \varphi$*

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

lemma *no-T-F-symb-except-toplevel-not-decom*:
 $\varphi \neq FT \implies \varphi \neq FF \implies no-T-F-symb-except-toplevel (FNot \varphi)$
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* ∈ *binary-connectives*
shows *no-T-F-symb-except-toplevel (conn c [φ , ψ])*
by (*metis (no-types, lifting) assms c conn.simps(4) list.discI noTrue-no-T-F-symb-except-toplevel wf-conn-no-T-F-symb-iff no-T-F-symb-fnot set.ConsD wf-conn-binary wf-conn-helper-facts(1) wf-conn-list-decomp(1,2)*)

lemma *no-T-F-symb-except-toplevel-if-is-a-true-false*:
fixes *l* :: 'v *propo list* **and** *c* :: 'v *connective*
assumes *corr*: *wf-conn c l*
and *FT* ∈ *set l* ∨ *FF* ∈ *set l*
shows $\neg no-T-F-symb-except-toplevel (conn c l)$
by (*metis assms empty-iff no-T-F-symb-except-toplevel.simps wf-conn-no-T-F-symb-iff set-empty wf-conn-list(1,2)*)

lemma *no-T-F-symb-except-top-level-false-example[simp]*:
fixes $\varphi \psi$:: 'v *propo*
assumes $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
shows
 $\neg no-T-F-symb-except-toplevel (FAnd \varphi \psi)$
 $\neg no-T-F-symb-except-toplevel (FOr \varphi \psi)$
 $\neg no-T-F-symb-except-toplevel (FImp \varphi \psi)$
 $\neg no-T-F-symb-except-toplevel (FEq \varphi \psi)$
using *assms no-T-F-symb-except-toplevel-if-is-a-true-false unfolding binary-connectives-def*
by (*metis (no-types) conn.simps(5-8) insert-iff list.simps(14-15) wf-conn-helper-facts(5-8)*) +

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

```

fixes  $\varphi \ \psi :: 'v \text{ propo}$ 
assumes  $\varphi = FT \vee \varphi = FF$ 
shows
   $\neg \text{no-}T\text{-}F\text{-symb-except-toplevel} (F\text{Not } \varphi)$ 
by (simp add: assms no-}T\text{-}F\text{-symb-except-toplevel.simps)

```

This is the local extension of *no-}T\text{-}F\text{-symb-except-toplevel*.

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

lemma *no-}T\text{-}F\text{-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 $\neg \text{no-}T\text{-}F\text{-except-top-level} (\text{conn } c \ l)$
by (*simp add: all-subformula-st-decomp assms no-}T\text{-}F\text{-except-top-level-def*
no-}T\text{-}F\text{-symb-except-toplevel-if-is-a-true-false)

lemma *no-}T\text{-}F\text{-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
 $\neg \text{no-}T\text{-}F\text{-except-top-level} (F\text{And } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-except-top-level} (F\text{Or } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-except-top-level} (F\text{Eq } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-except-top-level} (F\text{Imp } \varphi \ \psi)$
by (*metis all-subformula-st-test-symb-true-phi assms no-}T\text{-}F\text{-except-top-level-def*
no-}T\text{-}F\text{-symb-except-top-level-false-example)+

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

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

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

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

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

lemma *no-}T\text{-}F\text{-no-}T\text{-}F\text{-except-top-level'}*[*simp*]:

$no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\ \varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee no\text{-}T\text{-}F\ \varphi)$
apply *auto*
using *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-no-T-F-except-top-level*
by *blast+*

lemma *no-T-F-bin-decomp[simp]*:
assumes *c: c ∈ binary-connectives*
shows $no\text{-}T\text{-}F\ (conn\ c\ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F\ \varphi \wedge no\text{-}T\text{-}F\ \psi)$
proof –
have *wf: wf-conn c [ϕ, ψ] using c by auto*
hence $no\text{-}T\text{-}F\ (conn\ c\ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F\text{-}symb\ (conn\ c\ [\varphi, \psi]) \wedge no\text{-}T\text{-}F\ \varphi \wedge no\text{-}T\text{-}F\ \psi)$
by (*simp add: all-subformula-st-decomp no-T-F-def*)
thus $no\text{-}T\text{-}F\ (conn\ c\ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F\ \varphi \wedge no\text{-}T\text{-}F\ \psi)$
using *c wf all-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 ∨ c = COr ∨ c = CEq ∨ c = CImp*
shows $no\text{-}T\text{-}F\ (conn\ c\ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F\ \varphi \wedge no\text{-}T\text{-}F\ \psi)$
using *no-T-F-bin-decomp assms unfolding binary-connectives-def by blast*

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

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

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

lemma *no-T-F-symb-except-toplevel-step-exists*:
fixes $\varphi\ \psi :: 'v\ propo$
assumes $no\text{-}equiv\ \varphi$ **and** $no\text{-}imp\ \varphi$

shows $\psi \preceq \varphi \implies \neg \text{no-}T\text{-}F\text{-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTB } \psi \ \psi'$
proof (*induct* ψ *rule: propo-induct-arity*)
 case (*nullary* $\varphi' \ x$)
 hence *False* **using** *no-}T\text{-}F\text{-symb-except-toplevel-true no-}T\text{-}F\text{-symb-except-toplevel-false* **by** *auto*
 thus *?case* **by** *blast*
next
 case (*unary* ψ)
 hence $\psi = FF \vee \psi = FT$ **using** *no-}T\text{-}F\text{-symb-except-toplevel-not-decom* **by** *blast*
 thus *?case* **using** *ElimTB5 ElimTB6* **by** *blast*
next
 case (*binary* $\varphi' \ \psi1 \ \psi2$)
 note *IH1 = this(1) and IH2 = this(2) and $\varphi' = \text{this}(3)$ and $F\varphi = \text{this}(4)$ and $n = \text{this}(5)$*
 {
 assume $\varphi' = FImp \ \psi1 \ \psi2 \vee \varphi' = FEq \ \psi1 \ \psi2$
 hence *False* **using** *n F φ subformula-all-subformula-st assms* **by** (*metis (no-types) no-equiv-eq(1)*
 no-equiv-def no-imp-Imp(1) no-imp-def)
 hence *?case* **by** *blast*
 }
 moreover {
 assume $\varphi': \varphi' = FAnd \ \psi1 \ \psi2 \vee \varphi' = FOr \ \psi1 \ \psi2$
 hence $\psi1 = FT \vee \psi2 = FT \vee \psi1 = FF \vee \psi2 = FF$
 using *no-}T\text{-}F\text{-symb-except-toplevel-bin-decom conn.simps(5,6) n* **unfolding** *binary-connectives-def*
 by *fastforce+*
 hence *?case* **using** *elimTB.intros φ'* **by** *blast*
 }
 ultimately show *?case* **using** φ' **by** *blast*
qed

lemma *no-}T\text{-}F\text{-except-top-level-rew:*

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

```

lemma elimTB-inv:
  fixes  $\varphi \psi :: 'v \text{ propo}$ 
  assumes full (propo-rew-step elimTB)  $\varphi \psi$ 
  and no-equiv  $\varphi$  and no-imp  $\varphi$ 
  shows no-equiv  $\psi$  and no-imp  $\psi$ 
proof –
  {
    fix  $\varphi \psi :: 'v \text{ propo}$ 
    have  $H: \text{elimTB } \varphi \psi \implies \text{no-equiv } \varphi \implies \text{no-equiv } \psi$ 
      by (induct  $\varphi \psi$  rule: elimTB.induct, auto)
  }
  thus no-equiv  $\psi$ 
  using full-propo-rew-step-inv-stay-conn[of elimTB no-equiv-symb  $\varphi \psi$ ]
    no-equiv-symb-conn-characterization assms unfolding no-equiv-def by metis
next
  {
    fix  $\varphi \psi :: 'v \text{ propo}$ 
    have  $H: \text{elimTB } \varphi \psi \implies \text{no-imp } \varphi \implies \text{no-imp } \psi$ 
      by (induct  $\varphi \psi$  rule: elimTB.induct, auto)
  }
  thus 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  $\varphi \psi :: 'v \text{ propo}$ 
  assumes no-equiv  $\varphi$  and no-imp  $\varphi$  and full (propo-rew-step elimTB)  $\varphi \psi$ 
  shows no-T-F-except-top-level  $\psi$ 
  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 literal.

inductive *pushNeg* **where**

```

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

```

lemma *pushNeg-transformation-consistent*:

```

 $A \models \text{FNot } (\text{FAnd } \varphi \psi) \longleftrightarrow A \models (\text{FOR } (\text{FNot } \varphi) (\text{FNot } \psi))$ 
 $A \models \text{FNot } (\text{FOR } \varphi \psi) \longleftrightarrow A \models (\text{FAnd } (\text{FNot } \varphi) (\text{FNot } \psi))$ 
 $A \models \text{FNot } (\text{FNot } \varphi) \longleftrightarrow A \models \varphi$ 
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 $\varphi \longleftrightarrow (\varphi = FT \vee \varphi = FF \vee (\exists x. \varphi = FVar x))$
by (*case-tac φ , auto*)

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

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

fun *simple-not-symb* **where**
simple-not-symb (FNot φ) = (simple φ) |
simple-not-symb - = True

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

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

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

lemma *simple-not-rew*:
fixes $\varphi :: 'v \text{ propo}$
assumes *noTB*: $\neg \text{simple-not } \varphi$ **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 $\forall x. \text{simple-not-symb } (FF :: 'v \text{ propo}) \wedge \text{simple-not-symb } FT \wedge \text{simple-not-symb } (FVar (x :: 'v))$
by *auto*
moreover {
fix $c :: 'v \text{ connective}$ **and** $l :: 'v \text{ propo list}$ **and** $\psi :: 'v \text{ propo}$
have $H: \text{pushNeg } (\text{conn } c \ l) \ \psi \implies \neg \text{simple-not-symb } (\text{conn } c \ l)$
by (*case-tac* (*conn* $c \ l$) *rule: pushNeg.cases, simp-all*)
}
moreover {
fix $x :: 'v$
have $H': \text{simple-not } FT \text{ simple-not } FF \text{ simple-not } (FVar \ x)$
by *simp-all*
}
moreover {
fix $\psi :: 'v \text{ propo}$
have $\psi \preceq \varphi \implies \neg \text{simple-not-symb } \psi \implies \exists \psi'. \text{pushNeg } \psi \psi'$
using *simple-not-step-exists no-equiv no-imp* **by** *blast*
}
ultimately show *?thesis* **using** *no-test-symb-step-exists noTB unfolding simple-not-def* **by** *blast*
qed

lemma *no-T-F-except-top-level-pushNeg1*:
 $\text{no-T-F-except-top-level } (FNot (FAnd \ \varphi \ \psi)) \implies \text{no-T-F-except-top-level } (FOr (FNot \ \varphi) (FNot \ \psi))$
using *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-comp-not no-T-F-decomp(1)*
 $\text{no-T-F-decomp}(2) \text{ no-T-F-no-T-F-except-top-level}$ **by** (*metis no-T-F-comp-expanded-explicit(2)*
 $\text{propo.distinct}(5,17)$)

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

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

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

proof –
fix $\varphi \varphi' :: 'a \text{ propo}$ **and** $c :: 'a \text{ connective}$ **and** $\xi \xi' :: 'a \text{ propo list}$
assume *rel*: *propo-rew-step pushNeg* $\varphi \ \varphi'$
and *IH*: $\text{no-T-F } \varphi \implies \text{no-T-F-symb } \varphi \implies \text{no-T-F-symb } \varphi'$
and *wf*: *wf-conn* $c \ (\xi @ \varphi \# \xi')$
and $n: \text{conn } c \ (\xi @ \varphi \# \xi') = FF \vee \text{conn } c \ (\xi @ \varphi \# \xi') = FT \vee \text{no-T-F } (\text{conn } c \ (\xi @ \varphi \# \xi'))$

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

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

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

proof –
 {
 fix $\varphi \psi :: 'v \text{ propo}$
 assume *rel*: *propo-rew-step pushNeg* $\varphi \psi$
 and *no*: *no-T-F-except-top-level* φ
 hence *no-T-F-except-top-level* ψ
proof –
 {
 assume $\varphi = FT \vee \varphi = FF$
 from *rel this* have *False*
 apply (*induct rule: propo-rew-step.induct*)
 using *pushNeg.cases* apply *blast*
 using *wf-conn-list(1) wf-conn-list(2)* by *auto*
 }
 }


```

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

```

```

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

```

```

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

```

8.5 Push inside

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

$\text{not-c-in-c'-symb-l[simp]: wf-conn } c \text{ [conn } c' [\varphi, \varphi'], \psi] \implies \text{wf-conn } c' [\varphi, \varphi']$
 $\implies \text{not-c-in-c'-symb } c \text{ } c' (\text{conn } c \text{ [conn } c' [\varphi, \varphi'], \psi]) \mid$
 $\text{not-c-in-c'-symb-r[simp]: wf-conn } c \text{ } [\psi, \text{conn } c' [\varphi, \varphi']] \implies \text{wf-conn } c' [\varphi, \varphi']$
 $\implies \text{not-c-in-c'-symb } c \text{ } c' (\text{conn } c \text{ } [\psi, \text{conn } c' [\varphi, \varphi']])$

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

lemma $c\text{-in-c'-symb-simp}$:

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

apply (induct rule: $\text{not-c-in-c'-symb.induct}$, auto simp add: $\text{wf-conn.simps wf-conn-list}(1-3)$)
using $\text{conn-inj-not}(2)$ wf-conn-binary **unfolding** $\text{binary-connectives-def}$ **by** fastforce+

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

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

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

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

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

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

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

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

proof (induct rule: $\text{not-c-in-c'-symb.induct}$)

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

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

have $\text{wf-conn } c \text{ } [\text{conn } c' \text{ } [\varphi'', \psi'], \varphi]$

using $H(1)$ $\text{wf-conn-no-arity-change length-Cons}$ **by** metis

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

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** $H = \text{this}$

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

moreover have $\text{wf-conn } c \text{ } [\psi', \text{conn } c' \text{ } [\varphi', \varphi'']]$

using $H(1)$ $\text{wf-conn-no-arity-change length-Cons}$ **by** metis

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

using $\text{not-c-in-c'-symb.intros}(2)$ $\text{conn-inj not-c-in-c'-symb-l.hyps}$

$\text{not-c-in-c'-symb-l.prem}(1,2)$ **by** blast

qed

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

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

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

assumes *wf*: *wf-conn* c $[\varphi, \psi]$

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

proof –

have $?A \longleftrightarrow (c\text{-in-}c'\text{-symb } c \ c' \ (conn \ c \ [\varphi, \psi])$

$\wedge (\forall \xi \in set \ [\varphi, \psi]. \ all\text{-subformula-st } (c\text{-in-}c'\text{-symb } c \ c') \ \xi))$

using *all-subformula-st-decomp* *wf* **unfolding** *c-in-c'-only-def* **by** *fastforce*

also have $\dots \longleftrightarrow (c\text{-in-}c'\text{-symb } c \ c' \ (conn \ c \ [\psi, \varphi])$

$\wedge (\forall \xi \in set \ [\psi, \varphi]. \ all\text{-subformula-st } (c\text{-in-}c'\text{-symb } c \ c') \ \xi))$

using *not-c-in-c'-symb-commute'* *wf* **by** *auto*

also

have *wf-conn* c $[\psi, \varphi]$ **using** *wf-conn-no-arity-change* *wf* **by** (*metis length-Cons*)

hence $(c\text{-in-}c'\text{-symb } c \ c' \ (conn \ c \ [\psi, \varphi])$

$\wedge (\forall \xi \in set \ [\psi, \varphi]. \ all\text{-subformula-st } (c\text{-in-}c'\text{-symb } c \ c') \ \xi))$

$\longleftrightarrow ?B$

using *all-subformula-st-decomp* **unfolding** *c-in-c'-only-def* **by** *fastforce*

finally show *?thesis* .

qed

lemma *not-c-in-c'-simp[simp]*:

fixes $\varphi 1 \ \varphi 2 \ \psi :: 'v \ propo$ **and** $x :: 'v$

shows

$c\text{-in-}c'\text{-symb } c \ c' \ FT$

$c\text{-in-}c'\text{-symb } c \ c' \ FF$

$c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ x)$

$wf\text{-conn } c \ [conn \ c' \ [\varphi 1, \varphi 2], \psi] \implies wf\text{-conn } c' \ [\varphi 1, \varphi 2]$

$\implies \neg c\text{-in-}c'\text{-only } c \ c' \ (conn \ c \ [conn \ c' \ [\varphi 1, \varphi 2], \psi])$

apply (*simp-all add: c-in-c'-only-def*)

using *all-subformula-st-test-symb-true-phi* *not-c-in-c'-symb-l* **by** *blast*

lemma *c-in-c'-symb-not[simp]*:

fixes $c \ c' :: 'v \ connective$ **and** $\psi :: 'v \ propo$

shows $c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ \psi)$

proof –

{

fix $\xi :: 'v \ propo$

have $not\text{-}c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ \psi) \implies False$

apply (*induct FNot* ψ *rule: not-c-in-c'-symb.induct*)

using *conn-inj-not(2)* **by** *blast+*

}

thus *?thesis* **by** *auto*

qed

lemma *c-in-c'-symb-step-exists*:

fixes $\varphi :: 'v \ propo$

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

shows $\psi \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi \implies \exists \psi'. \ push\text{-conn-inside } c \ c' \ \psi \ \psi'$

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

apply *auto[2]*

proof –

```

fix  $\psi1 \ \psi2 \ \varphi':: 'v \text{ propo}$ 
assume  $IH\psi1: \psi1 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi1 \implies Ex \ (push\text{-conn-}inside \ c \ c' \ \psi1)$ 
and  $IH\psi2: \psi2 \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi2 \implies Ex \ (push\text{-conn-}inside \ c \ c' \ \psi2)$ 
and  $\varphi': \varphi' = FAnd \ \psi1 \ \psi2 \vee \varphi' = FOr \ \psi1 \ \psi2 \vee \varphi' = FImp \ \psi1 \ \psi2 \vee \varphi' = FEq \ \psi1 \ \psi2$ 
and  $in\varphi: \varphi' \preceq \varphi$  and  $n0: \neg c\text{-in-}c'\text{-symb } c \ c' \ \varphi'$ 
hence  $n: not\text{-}c\text{-in-}c'\text{-symb } c \ c' \ \varphi'$  by auto
{
  assume  $\varphi': \varphi' = conn \ c \ [\psi1, \psi2]$ 
  obtain  $a \ b$  where  $\psi1 = conn \ c' \ [a, b] \vee \psi2 = conn \ c' \ [a, b]$ 
  using  $n \ \varphi'$  apply (induct rule: not-c-in-c'-symb.induct)
  using  $c$  by force+
  hence  $Ex \ (push\text{-conn-}inside \ c \ c' \ \varphi')$ 
  unfolding  $\varphi'$  apply auto
  using  $push\text{-conn-}inside.intros(1) \ c \ c'$  apply blast
  using  $push\text{-conn-}inside.intros(2) \ c \ c'$  by blast
}
moreover {
  assume  $\varphi': \varphi' \neq conn \ c \ [\psi1, \psi2]$ 
  have  $\forall \varphi \ c \ ca. \exists \varphi1 \ \psi1 \ \psi2 \ \psi1' \ \psi2' \ \varphi2'. \ conn \ (c::'v \text{ connective}) \ [\varphi1, \ conn \ ca \ [\psi1, \psi2]] = \varphi$ 
     $\vee \ conn \ c \ [conn \ ca \ [\psi1', \psi2'], \varphi2'] = \varphi \vee c\text{-in-}c'\text{-symb } c \ ca \ \varphi$ 
  by (metis not-c-in-c'-symb.cases)
  hence  $Ex \ (push\text{-conn-}inside \ c \ c' \ \varphi')$ 
  by (metis (no-types)  $c \ c' \ n \ push\text{-conn-}inside\text{-l} \ push\text{-conn-}inside\text{-r}$ )
}
ultimately show  $Ex \ (push\text{-conn-}inside \ c \ c' \ \varphi')$  by blast
qed

```

lemma $c\text{-in-}c'\text{-symb-rew}$:

```

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  $\exists \psi \ \psi'. \psi \preceq \varphi \wedge push\text{-conn-}inside \ c \ c' \ \psi \ \psi'$ 
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  $\psi \preceq \varphi \implies \neg c\text{-in-}c'\text{-symb } c \ c' \ \psi \implies \exists \psi'. push\text{-conn-}inside \ c \ c' \ \psi \ \psi'$ 
    by (auto simp add: assms(2)  $c' \ c\text{-in-}c'\text{-symb-step-exists}$ )
  }
  ultimately show ?thesis using noTB no-test-symb-step-exists[of  $c\text{-in-}c'\text{-symb } c \ c'$ ]
  unfolding  $c\text{-in-}c'\text{-only-def}$  by metis
qed

```

lemma $push\text{-conn-}inside\text{-}c'\text{-symb-no-T-F}$:

```

fixes  $\varphi \ \psi :: 'v \text{ propo}$ 
shows  $propo\text{-rew-step} \ (push\text{-conn-}inside \ c \ c') \ \varphi \ \psi \implies no\text{-T-F } \varphi \implies no\text{-T-F } \psi$ 

```

```

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

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

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

```

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

lemma simple-propo-rew-step-inv-push-conn-inside-simple-not:
 fixes c c' :: 'v connective **and** $\varphi \psi$:: 'v propo
 shows propo-rew-step (push-conn-inside c c') $\varphi \psi \implies$ simple-not $\varphi \implies$ simple-not ψ
proof (induct rule: propo-rew-step.induct)
case (global-rel $\varphi \psi$)
thus ?case **by** (case-tac φ , auto simp add: push-conn-inside.simps)

```

next
case (propo-rew-one-step-lift  $\varphi$   $\varphi'$   $ca$   $\xi$   $\xi'$ )
thus ?case
proof (case-tac  $ca$  rule: connective-cases-arity, auto)
  fix  $\varphi$   $\varphi'$  :: 'v propo and  $c$  :: 'v connective and  $\xi$   $\xi'$  :: 'v propo list
  assume rel: propo-rew-step (push-conn-inside  $c$   $c'$ )  $\varphi$   $\varphi'$ 
  assume simple  $\varphi$ 
  thus simple  $\varphi'$  using rel simple-propo-rew-step-push-conn-inside-inv by blast
next
fix  $\varphi$   $\varphi'$  :: 'v propo and  $ca$  :: 'v connective and  $\xi$   $\xi'$  :: 'v propo list
assume rel: propo-rew-step (push-conn-inside  $c$   $c'$ )  $\varphi$   $\varphi'$ 
and IH: all-subformula-st simple-not-symb  $\varphi \implies$  all-subformula-st simple-not-symb  $\varphi'$ 
and wf: wf-conn  $ca$  ( $\xi @ \varphi \# \xi'$ )
and simple-not: all-subformula-st simple-not-symb (conn  $ca$  ( $\xi @ \varphi \# \xi'$ ))
and ca:  $ca \in$  binary-connectives

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

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

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

```

```

assume corr-ca: wf-conn ca ( $\chi s @ \varphi \# \chi s'$ )
and simple-not: simple (conn ca ( $\chi s @ \varphi \# \chi s'$ ))
hence False
proof (case-tac ca rule: connective-cases-arity)
  fix x :: 'v
  assume simple (conn ca ( $\chi s @ \varphi \# \chi s'$ )) and ca = CT  $\vee$  ca = CF  $\vee$  ca = CVar x
  hence  $\chi s @ \varphi \# \chi s' = []$  using corr-ca by auto
  thus False by auto
next
  assume simple: simple (conn ca ( $\chi s @ \varphi \# \chi s'$ ))
  and ca: ca  $\in$  binary-connectives
  obtain a b where ab:  $\chi s @ \varphi \# \chi s' = [a, b]$ 
    using corr-ca ca list-length2-decomp wf-conn-bin-list-length
    by (metis append-assoc length-Cons length-append length-append-singleton)
  thus False using simple ca ab conn.simps(5,6,7,8) unfolding binary-connectives-def by auto
next
  assume simple: simple (conn ca ( $\chi s @ \varphi \# \chi s'$ ))
  and ca: ca = CNot
  hence empty:  $\chi s = []$   $\chi s' = []$  using corr-ca by auto
  thus False using simple ca conn.simps(4) by auto
qed
thus simple (conn ca ( $\chi s @ \varphi' \# \chi s'$ )) by blast
qed

```

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

lemma *push-conn-inside-inv*:
fixes $\varphi \psi :: 'v$ *propo*
assumes *full* (*propo-rew-step* (*push-conn-inside* *c* *c'*)) $\varphi \psi$
and *no-equiv* 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  $\varphi \psi :: 'v$  propo
    have H: push-conn-inside c c'  $\varphi \psi \implies$  all-subformula-st simple-not-symb  $\varphi$ 
       $\implies$  all-subformula-st simple-not-symb  $\psi$ 
      by (induct  $\varphi \psi$  rule: push-conn-inside.induct, auto)
  } note H = this
}

```

```

fix  $\varphi \psi :: 'v$  propo
have H: propo-rew-step (push-conn-inside c c')  $\varphi \psi \implies$  all-subformula-st simple-not-symb  $\varphi$ 
   $\implies$  all-subformula-st simple-not-symb  $\psi$ 
apply (induct  $\varphi \psi$  rule: propo-rew-step.induct)
using H apply simp
proof (case-tac ca rule: connective-cases-arity)
  fix  $\varphi \varphi' :: 'v$  propo and c:: 'v connective and  $\xi \xi' :: 'v$  propo list
  and x:: 'v
  assume wf-conn c ( $\xi @ \varphi \# \xi'$ )
  and c = CT  $\vee$  c = CF  $\vee$  c = CVar x
  hence  $\xi @ \varphi \# \xi' = []$  by auto
  hence False by auto
  thus all-subformula-st simple-not-symb (conn c ( $\xi @ \varphi' \# \xi'$ )) by blast

```



```

next
  fix  $\varphi \varphi' :: 'v \text{ propo}$  and  $ca :: 'v \text{ connective}$  and  $\xi \xi' :: 'v \text{ propo list}$ 
  and  $x :: 'v$ 
  assume  $rel$ :  $\text{propo-rew-step (push-conn-inside } c \ c') \ \varphi \ \varphi'$ 
  and  $\varphi\text{-}\varphi'$ :  $\text{all-subformula-st simple-not-symb } \varphi \implies \text{all-subformula-st simple-not-symb } \varphi'$ 
  and  $corr$ :  $\text{wf-conn } ca \ (\xi \ @ \ \varphi \ \# \ \xi')$ 
  and  $n$ :  $\text{all-subformula-st simple-not-symb (conn } ca \ (\xi \ @ \ \varphi \ \# \ \xi'))$ 
  and  $c$ :  $ca = CNot$ 

  have  $empty$ :  $\xi = [] \ \xi' = []$  using  $c \ corr$  by auto
  hence  $\text{simple-not:all-subformula-st simple-not-symb (FNot } \varphi)$  using  $corr \ c \ n$  by auto
  hence  $\text{simple } \varphi$ 
    using  $\text{all-subformula-st-test-symb-true-phi simple-not-symb.simps(1)}$  by blast
  hence  $\text{simple } \varphi'$ 
    using  $rel \ \text{simple-propo-rew-step-push-conn-inside-inv}$  by blast
  thus  $\text{all-subformula-st simple-not-symb (conn } ca \ (\xi \ @ \ \varphi' \ \# \ \xi'))$  using  $c \ empty$ 
    by ( $\text{metis simple-not } \varphi\text{-}\varphi' \ \text{append-Nil conn.simps(4) all-subformula-st-decomp-explicit(3)}$ 
       $\text{simple-not-symb.simps(1)})$ 
next
  fix  $\varphi \varphi' :: 'v \text{ propo}$  and  $ca :: 'v \text{ connective}$  and  $\xi \xi' :: 'v \text{ propo list}$ 
  and  $x :: 'v$ 
  assume  $rel$ :  $\text{propo-rew-step (push-conn-inside } c \ c') \ \varphi \ \varphi'$ 
  and  $n\varphi$ :  $\text{all-subformula-st simple-not-symb } \varphi \implies \text{all-subformula-st simple-not-symb } \varphi'$ 
  and  $corr$ :  $\text{wf-conn } ca \ (\xi \ @ \ \varphi \ \# \ \xi')$ 
  and  $n$ :  $\text{all-subformula-st simple-not-symb (conn } ca \ (\xi \ @ \ \varphi \ \# \ \xi'))$ 
  and  $c$ :  $ca \in \text{binary-connectives}$ 

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

  moreover have  $\text{simple-not-symb (conn } ca \ (\xi \ @ \ \varphi' \ \# \ \xi'))$ 
    using  $ab \ \text{conn-inj-not(1) corr wf-conn-list-decomp(4) wf-conn-no-arity-change}$ 
     $\text{not-Cons-self2 self-append-conv2 simple-not-symb.elims(3)}$  by ( $\text{metis (no-types) } c$ 
       $\text{calculation(1) wf-conn-binary}$ )
  moreover have  $\text{wf-conn } ca \ (\xi \ @ \ \varphi' \ \# \ \xi')$  using  $c \ \text{calculation(1)}$  by auto
  ultimately show  $\text{all-subformula-st simple-not-symb (conn } ca \ (\xi \ @ \ \varphi' \ \# \ \xi'))$ 

```

```

    by (metis all-subformula-st-decomp-imp)
  qed
}
moreover {
  fix ca :: 'v connective and  $\xi \xi' :: 'v \text{ propo list}$  and  $\varphi \varphi' :: 'v \text{ propo}$ 
  have propo-rew-step (push-conn-inside c c')  $\varphi \varphi' \implies \text{wf-conn ca } (\xi @ \varphi \# \xi')$ 
     $\implies \text{simple-not-symb (conn ca } (\xi @ \varphi \# \xi')) \implies \text{simple-not-symb } \varphi'$ 
     $\implies \text{simple-not-symb (conn ca } (\xi @ \varphi' \# \xi'))$ 
  by (metis append-self-conv2 conn.simps(4) conn-inj-not(1) simple-not-symb.elims(3)
    simple-not-symb.simps(1) simple-propo-rew-step-push-conn-inside-inv
    wf-conn-no-arity-change-helper wf-conn-list-decomp(4) wf-conn-no-arity-change)
}
ultimately show simple-not  $\psi$ 
  using full-propo-rew-step-inv-stay'[of push-conn-inside c c' simple-not-symb] assms
  unfolding no-T-F-except-top-level-def simple-not-def full-unfold by metis
next
{
  fix  $\varphi \psi :: 'v \text{ propo}$ 
  have H: propo-rew-step (push-conn-inside c c')  $\varphi \psi \implies \text{no-T-F-except-top-level } \varphi$ 
     $\implies \text{no-T-F-except-top-level } \psi$ 
  proof -
    assume rel: propo-rew-step (push-conn-inside c c')  $\varphi \psi$ 
    and no-T-F-except-top-level  $\varphi$ 
    hence no-T-F  $\varphi \vee \varphi = FF \vee \varphi = FT$ 
      by (metis no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
    moreover {
      assume  $\varphi = FF \vee \varphi = FT$ 
      hence False using rel propo-rew-step-push-conn-inside by blast
      hence no-T-F-except-top-level  $\psi$  by blast
    }
    moreover {
      assume no-T-F  $\varphi \wedge \varphi \neq FF \wedge \varphi \neq FT$ 
      hence no-T-F  $\psi$  using rel push-conn-insidec-in-c'-symb-no-T-F by blast
      hence no-T-F-except-top-level  $\psi$  using no-T-F-no-T-F-except-top-level by blast
    }
    ultimately show no-T-F-except-top-level  $\psi$  by blast
  qed
}
moreover {
  fix ca :: 'v connective and  $\xi \xi' :: 'v \text{ propo list}$  and  $\varphi \varphi' :: 'v \text{ propo}$ 
  assume rel: propo-rew-step (push-conn-inside c c')  $\varphi \varphi'$ 
  assume corr: wf-conn ca  $(\xi @ \varphi \# \xi')$ 
  hence c:  $ca \neq CT \wedge ca \neq CF$  by auto
  assume no-T-F: no-T-F-symb-except-toplevel (conn ca  $(\xi @ \varphi \# \xi')$ )
  have no-T-F-symb-except-toplevel (conn ca  $(\xi @ \varphi' \# \xi')$ )
  proof
    have c:  $ca \neq CT \wedge ca \neq CF$  using corr by auto
    have  $\zeta: \forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \zeta \neq FT \wedge \zeta \neq FF$ 
      using corr no-T-F no-T-F-symb-except-toplevel-if-is-a-true-false by blast
    hence  $\varphi \neq FT \wedge \varphi \neq FF$  by auto
    from rel this have  $\varphi' \neq FT \wedge \varphi' \neq FF$ 
    apply (induct rule: propo-rew-step.induct)
    by (metis append-is-Nil-conv conn.simps(2) conn-inj list.distinct(1)
      wf-conn-helper-facts(3) wf-conn-list(1) wf-conn-no-arity-change
      wf-conn-no-arity-change-helper push-conn-inside-not-true-false)+
  qed
}

```

```

    hence  $\forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \zeta \neq FT \wedge \zeta \neq FF$  using  $\zeta$  by auto
    moreover have wf-conn ca  $(\xi @ \varphi' \# \xi')$ 
      using corr wf-conn-no-arity-change by (metis wf-conn-no-arity-change-helper)
    ultimately show no-T-F-symb (conn ca  $(\xi @ \varphi' \# \xi')$ ) using no-T-F-symb.intros c by metis
  qed
}
ultimately show no-T-F-except-top-level  $\psi$ 
  using full-propo-rew-step-inv-stay'[of push-conn-inside c c' no-T-F-symb-except-toplevel] assms
  unfolding no-T-F-except-top-level-def full-unfold by metis

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

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

```

lemma *push-conn-inside-full-propo-rew-step*:

```

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

```

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

inductive *only-c-inside-symb* $:: 'v \text{ connective} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **for** $c :: 'v \text{ connective}$ **where**

```

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

```

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

```

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

```

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

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

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

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

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

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

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

lemma *only-c-inside-decomp*:

only-c-inside $c \varphi \longleftrightarrow$
 $(\forall \psi. \psi \preceq \varphi \longrightarrow (\text{simple } \psi \vee (\exists \varphi'. \psi = \text{FNot } \varphi' \wedge \text{simple } \varphi')$
 $\vee (\exists l. \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l)))$
unfolding *only-c-inside-def* **by** (*auto simp add: 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
 $\{$
 $\text{fix } \varphi'$
 $\text{have } ? \psi \neq \text{FNot } \varphi' \text{ using } c' \text{ conn-inj-not(1) wf by blast}$
 $\}$
ultimately obtain $l :: 'v \text{ propo list}$ **where** $? \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l$ **by** *metis*
hence $c = c'$ **using** *conn-inj wf* **by** *metis*
thus *False* **using** cc' **by** *auto*

qed

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

assumes $\delta: c \neq c'$ **and** $c: c = \text{CAnd} \vee c = \text{COr}$ **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 propo list and c c' ca :: 'v connective
  shows wf-conn ca l  $\implies$  ca  $\neq$  c  $\implies$  c-in-c'-symb c c' (conn ca l)
proof -
  have not-c-in-c'-symb c c' (conn ca l)  $\implies$  wf-conn ca l  $\implies$  ca = c
    by (induct conn ca l rule: not-c-in-c'-symb.induct, auto simp add: conn-inj)
  thus wf-conn ca l  $\implies$  ca  $\neq$  c  $\implies$  c-in-c'-symb c c' (conn ca l) by blast
qed

lemma only-c-inside-implies-c-in-c'-only:
  assumes  $\delta$ : c  $\neq$  c' and c: c = CAnd  $\vee$  c = COr and c': c' = CAnd  $\vee$  c' = COr
  shows only-c-inside c  $\varphi \implies$  c-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 = CAnd  $\vee$  c = COr c' = CAnd  $\vee$  c' = COr c  $\neq$  c' and wf: wf-conn c [ $\varphi$ ,  $\psi$ ]
  and inv: no-equiv (conn c l) no-imp (conn c l) simple-not (conn c l)
  shows wf-conn c l  $\implies$  c-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)
  thus ?case by (auto simp add: wf-conn-list assms)
next
  case (unary  $\varphi$  la)
  hence c = CNot  $\wedge$  la = [ $\varphi$ ] by (metis (no-types) wf-conn-list(8))
  thus ?case using assms(2) assms(1) by blast
next
  case (binary  $\varphi_1$   $\varphi_2$ )
  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)
  hence 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
  hence 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
hence ( $\varphi$ 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
  hence only-c-inside c  $\varphi$ 1
    by (metis IH $\varphi$ 1  $\varphi$ 1 all-subformula-st-decomp-imp inc $\varphi$ 1 no-equiv no-equiv-def no-imp no-imp-def
        c-in-only $\varphi$ 1 only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
        subformula-all-subformula-st)
}
moreover {
  assume  $\exists \psi$ 1.  $\varphi$ 1 = FNot  $\psi$ 1
  then obtain  $\psi$ 1 where  $\varphi$ 1 = FNot  $\psi$ 1 by metis
  hence only-c-inside c  $\varphi$ 1
    by (metis all-subformula-st-def assms(1) connective.distinct(37,39) inc $\varphi$ 1
        only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
}
moreover {
  assume simple  $\varphi$ 1
  hence only-c-inside c  $\varphi$ 1
    by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
        only-c-inside-decomp-not only-c-inside-def)
}
ultimately have only-c-inside $\varphi$ 1: only-c-inside c  $\varphi$ 1 by metis

have c-in-only $\varphi$ 2: c-in-c'-only c c' (conn c2 l2)
  using only l  $\varphi$ 2 wf $\varphi$ 2 assms unfolding c-in-c'-only-def by auto
have c2c: c2  $\neq$  c'
proof
  assume c2c: c2 = c'
  then obtain  $\xi$ 1  $\xi$ 2 where l2: l2 = [ $\xi$ 1,  $\xi$ 2]
    by (metis assms(2) wf $\varphi$ 2 wf-conn.simps connective.distinct(7,9,19,21,29,31,37,39))
  hence c-in-c'-symb c c' (conn c [ $\varphi$ 1, conn c' l2])
    using c2c l only  $\varphi$ 2 all-subformula-st-test-symb-true-phi unfolding c-in-c'-only-def by auto
  moreover have not-c-in-c'-symb c c' (conn c [ $\varphi$ 1, conn c' l2])
    using assms(1) c2c l2 not-c-in-c'-symb-r wf $\varphi$ 2 wf-conn-helper-facts(5,6) by metis
  ultimately show False by auto
qed
hence ( $\varphi$ 2 = conn c l2  $\wedge$  wf-conn c l2)  $\vee$  ( $\exists \psi$ 2.  $\varphi$ 2 = FNot  $\psi$ 2)  $\vee$  simple  $\varphi$ 2
  using c2-eq by (metis  $\varphi$ 2 assms(1-3) c2-eq c2-imp simple.elims(3) wf $\varphi$ 2 wf-conn-list(4-7))
moreover {
  assume  $\varphi$ 2 = conn c l2  $\wedge$  wf-conn c l2
  hence only-c-inside c  $\varphi$ 2

```

```

    by (metis IH $\varphi 2$   $\varphi 2$  all-subformula-st-decomp inc $\varphi 2$  no-equiv no-equiv-def no-imp no-imp-def
        c-in-only $\varphi 2$  only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
        subformula-all-subformula-st)
  }
  moreover {
    assume  $\exists \psi 2. \varphi 2 = FNot \psi 2$ 
    then obtain  $\psi 2$  where  $\varphi 2 = FNot \psi 2$  by metis
    hence only-c-inside c  $\varphi 2$ 
    by (metis all-subformula-st-def assms(1-3) connective.distinct(38,40) inc $\varphi 2$ 
        only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
  }
  moreover {
    assume simple  $\varphi 2$ 
    hence only-c-inside c  $\varphi 2$ 
    by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
        only-c-inside-decomp-not only-c-inside-def)
  }
  ultimately have only-c-inside $\varphi 2$ : only-c-inside c  $\varphi 2$  by metis
  show ?case using l only-c-inside $\varphi 1$  only-c-inside $\varphi 2$  by auto
qed

```

8.5.2 Push Conjunction

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

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

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

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

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

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

8.5.3 Push Disjunction

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

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

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

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

lemma *not-or-in-and-only-or-and[simp]*:
 $\sim \text{or-in-and-only } (FOr \ (FAnd \ \psi1 \ \psi2) \ \varphi')$
unfolding *or-in-and-only-def*
by (*metis all-subformula-st-test-symb-true-phi conn.simps(5-6) not-c-in-c'-symb-l wf-conn-helper-facts(5) wf-conn-helper-facts(6)*)

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

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

9 The full transformations

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

9.1.1 Definition

The normal is a super group of groups

inductive *grouped-by* :: *'a connective \Rightarrow 'a propo \Rightarrow bool* **for** *c* **where**
simple-is-grouped[simp]: simple $\varphi \Rightarrow$ grouped-by c φ |
simple-not-is-grouped[simp]: simple $\varphi \Rightarrow$ grouped-by c (FNot φ) |
*connected-is-group[simp]: grouped-by c $\varphi \Rightarrow$ grouped-by c $\psi \Rightarrow$ wf-conn c [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' [$\varphi 1$, $\varphi 2$]) $\implies c' = CAnd \vee c' = COr \implies wf\text{-}conn$ c' [$\varphi 1$, $\varphi 2$]
 $\implies c' = c$
by (*induct conn* c' [$\varphi 1$, $\varphi 2$] *rule: only-c-inside-symb.induct, auto simp add: conn-inj*)

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

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

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

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

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

case (*nullary* φ x)

thus $?G$ φ **by** *auto*

next

case (*unary* ψ)

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

next

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

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

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

proof –

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

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

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

have *only-c-inside-symb* c (*conn* c'' [$\varphi 1$, $\varphi 2$])

using *only all-subformula-st-test-symb-true-phi*

unfolding *only-c-inside-def* $\varphi\text{-}c''$ l'' **by** *metis*

hence $c = c''$

by (*metis* $\varphi\text{-}c''$ *conn-inj conn-inj-not(2) l'' list.distinct(1) list.inject wf*
only-c-inside-symb.cases simple.simps(5–8))

thus $\varphi = conn$ c [$\varphi 1$, $\varphi 2$] **and** $wf\text{-}conn$ c [$\varphi 1$, $\varphi 2$] **using** $\varphi\text{-}c''$ wf l'' **by** *auto*

qed

have *grouped-by* c $\varphi 1$ **using** wf $IH\varphi 1$ $IH\varphi 2$ $\varphi\text{-}conn$ $only$ φ **unfolding** *only-c-inside-def* **by** *auto*

moreover **have** *grouped-by* c $\varphi 2$

using wf φ $IH\varphi 1$ $IH\varphi 2$ $\varphi\text{-}conn$ $only$ **unfolding** *only-c-inside-def* **by** *auto*

ultimately show $?G$ φ **using** $\varphi\text{-}conn$ *connected-is-group local.wf* **by** *blast*

qed

lemma *grouped-by-false*:

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

apply (*induct conn* c' [φ , ψ] *rule: grouped-by.induct*)

apply (*auto simp add: simple-decomp wf-conn-list, auto simp add: conn-inj*)

by (*metis list.distinct(1) list.sel(3) wf-conn-list(8)*)**+**

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

inductive *super-grouped-by*:: '*a* *connective* \Rightarrow '*a* *connective* \Rightarrow '*a* *propo* \Rightarrow *bool* **for** c c' **where**

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

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

```

    by (simp add: only-c-inside-into-only-c-inside all-subformula-st-decomp)
  hence grouped-by c  $\varphi$  using  $\varphi$  only-c-inside-imp-grouped-by c by blast
  hence ?S  $\varphi$  using super-grouped-by.intros(1) by metis
}
ultimately show ?S  $\varphi$  by (metis  $\varphi'$  c c' cc' conn.simps(5,6) wf-conn-helper-facts(5,6))
qed

```

9.2 Conjunctive Normal Form

definition *is-conj-with-TF* **where** *is-conj-with-TF* == *super-grouped-by COr CAnd*

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

```

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

```

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

9.2.1 Full CNF transformation

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

definition *cnf-rew* **where** *cnf-rew* =
 (full (propo-rew-step elim-equiv)) OO
 (full (propo-rew-step elim-imp)) OO
 (full (propo-rew-step elimTB)) OO
 (full (propo-rew-step pushNeg)) OO
 (full (propo-rew-step pushDisj))

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

```

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

```

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

```

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

```

proof –

```

  fix  $\varphi \varphi_{Eq} \varphi_{Imp} \varphi_{TB} \varphi_{Neg} \varphi_{Disj} :: 'v$  propo
  assume Eq: full (propo-rew-step elim-equiv)  $\varphi \varphi_{Eq}$ 
  hence no-equiv: no-equiv  $\varphi_{Eq}$  using no-equiv-full-propo-rew-step-elim-equiv by blast

```

```

  assume Imp: full (propo-rew-step elim-imp)  $\varphi_{Eq} \varphi_{Imp}$ 
  hence no-imp: no-imp  $\varphi_{Imp}$  using no-imp-full-propo-rew-step-elim-imp by blast
  have no-imp-inv: no-equiv  $\varphi_{Imp}$  using no-equiv Imp elim-imp-inv by blast

```

```

  assume TB: full (propo-rew-step elimTB)  $\varphi_{Imp} \varphi_{TB}$ 
  hence noTB: no-T-F-except-top-level  $\varphi_{TB}$ 
    using no-imp-inv no-imp elimTB-full-propo-rew-step by blast
  have noTB-inv: no-equiv  $\varphi_{TB}$  no-imp  $\varphi_{TB}$  using elimTB-inv TB no-imp no-imp-inv by blast+

```

```

  assume Neg: full (propo-rew-step pushNeg)  $\varphi_{TB} \varphi_{Neg}$ 
  hence noNeg: simple-not  $\varphi_{Neg}$ 
    using noTB-inv noTB pushNeg-full-propo-rew-step by blast
  have noNeg-inv: no-equiv  $\varphi_{Neg}$  no-imp  $\varphi_{Neg}$  no-T-F-except-top-level  $\varphi_{Neg}$ 

```

using *pushNeg-inv Neg noTB noTB-inv* **by** *blast+*
assume *Disj: full (propo-rew-step pushDisj) φ Neg φ Disj*
hence *no-Disj: or-in-and-only φ Disj*
using *noNeg-inv noNeg pushDisj-full-propo-rew-step* **by** *blast*
have *noDisj-inv: no-equiv φ Disj no-imp φ Disj no-T-F-except-top-level φ Disj*
simple-not φ Disj
using *pushDisj-inv Disj noNeg noNeg-inv* **by** *blast+*
moreover have *is-conj-with-TF φ Disj*
using *or-in-and-only-conjunction-in-disj noDisj-inv no-Disj* **by** *blast*
ultimately show *is-cnff φ Disj unfolding is-cnff-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 $\varphi \longleftrightarrow$ is-disj-with-TF $\varphi \wedge$ no-T-F-except-top-level φ

9.3.1 Full DNF transform

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

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

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

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

theorem *dnf-transformation-correction:*

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

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

10.1 Transformation

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

inductive *elimTBFull* **where**

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

ElimTBFull2[simp]: *elimTBFull* (*FAnd* 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 elimTBFull  $\varphi \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$ 
    by (induct-tac rule: elimTBFull.inducts, auto)
}
thus ?thesis using preserves-un-sat-def by auto
qed
```

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

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

thus *Ex* (*elimTBFull* φ') **by** *blast*

```

next
  case (unary  $\psi$ )
  hence  $\psi = FF \vee \psi = FT$  using no-T-F-symb-except-toplevel-not-decom by blast
  thus  $Ex (elimTBFULL (FNot \psi))$  using elimTBFULL5 elimTBFULL5' by blast
next
  case (binary  $\varphi' \psi_1 \psi_2$ )
  hence  $\psi_1 = FT \vee \psi_2 = FT \vee \psi_1 = FF \vee \psi_2 = FF$ 
    by (metis binary-connectives-def conn.simps(5-8) insertI1 insert-commute
      no-T-F-symb-except-toplevel-bin-decom binary.hyps(3))
  thus  $Ex (elimTBFULL \varphi')$  using elimTBFULL.intros binary.hyps(3) by blast
qed

```

The same applies here. We do not need the assumption, but the deep link between $\neg no-T-F-except-top-level$ 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:  $\neg no-T-F-except-top-level \varphi$ 
  shows  $\exists \psi \psi'. \psi \preceq \varphi \wedge elimTBFULL \psi \psi'$ 
proof -
  have test-symb-false-nullary:
     $\forall x. no-T-F-symb-except-toplevel (FF :: 'v \text{ propo}) \wedge no-T-F-symb-except-toplevel FT$ 
     $\wedge no-T-F-symb-except-toplevel (FVar (x :: 'v))$ 
  by auto
  moreover {
    fix  $c :: 'v \text{ connective}$  and  $l :: 'v \text{ propo list}$  and  $\psi :: 'v \text{ propo}$ 
    have  $H: elimTBFULL (conn c l) \psi \implies \neg no-T-F-symb-except-toplevel (conn c l)$ 
      by (case-tac (conn c l) rule: elimTBFULL.cases, simp-all)
  }
  ultimately show ?thesis
    using no-test-symb-step-exists[of no-T-F-symb-except-toplevel \varphi elimTBFULL] noTB
    no-T-F-symb-except-toplevel-step-exists' unfolding no-T-F-except-top-level-def by metis
qed

```

```

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

```

10.2 More invariants

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

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

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

```

  fix  $\varphi' :: 'v \text{ propo}$  and  $\psi' :: 'v \text{ propo}$ 
  assume a1: no-T-F  $\varphi'$ 
  assume a2: elim-equiv  $\varphi' \psi'$ 
  have  $\forall x_0 x_1. (\neg \text{elim-equiv } (x_1 :: 'v \text{ propo}) x_0 \vee (\exists v_2 v_3 v_4 v_5 v_6 v_7. x_1 = FEq v_2 v_3$ 
     $\wedge x_0 = FAnd (FImp v_4 v_5) (FImp v_6 v_7) \wedge v_2 = v_4 \wedge v_4 = v_7 \wedge v_3 = v_5 \wedge v_3 = v_6))$ 
     $= (\neg \text{elim-equiv } x_1 x_0 \vee (\exists v_2 v_3 v_4 v_5 v_6 v_7. x_1 = FEq v_2 v_3$ 

```

```

 $\wedge x0 = FAnd (FImp v4 v5) (FImp v6 v7) \wedge v2 = v4 \wedge v4 = v7 \wedge v3 = v5 \wedge v3 = v6))$ 
by meson
hence  $\forall p pa. \neg elim-equiv (p :: 'v propo) pa \vee (\exists pb pc pd pe pf pg. p = FEq pb pc$ 
 $\wedge pa = FAnd (FImp pd pe) (FImp pf pg) \wedge pb = pd \wedge pd = pg \wedge pc = pe \wedge pc = pf)$ 
using elim-equiv.cases by force
thus no-T-F  $\psi'$  using a1 a2 by fastforce
next
fix  $\varphi \varphi' :: 'v propo$  and  $\xi \xi' :: 'v propo list$  and  $c :: 'v connective$ 
assume rel: propo-rew-step elim-equiv  $\varphi \varphi'$ 
and IH: no-T-F  $\varphi \implies$  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  $c: c = CNot$ 
  hence empty:  $\xi = [] \xi' = []$  using corr by auto
  hence no-T-F  $\varphi$  using no-T-F  $c$  no-T-F-decomp-not by auto
  hence no-T-F  $(conn c (\xi @ \varphi' \# \xi'))$  using c empty no-T-F-comp-not IH by auto
}
moreover {
  assume  $c: c \in binary-connectives$ 
  obtain  $a b$  where  $ab: \xi @ \varphi \# \xi' = [a, b]$ 
  using corr c list-length2-decomp wf-conn-bin-list-length by metis
  hence  $\varphi: \varphi = a \vee \varphi = b$ 
  by (metis append.simps(1) append-is-Nil-conv list.distinct(1) list.sel(3) nth-Cons-0
  tl-append2)
  have  $\zeta: \forall \zeta \in set (\xi @ \varphi \# \xi'). no-T-F \zeta$ 
  using no-T-F unfolding no-T-F-def using corr all-subformula-st-decomp by blast

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

lemma elim-equiv-inv':
fixes  $\varphi \psi :: 'v propo$ 

```

```

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

```



```

lemma propo-rew-step-ElimImp-no-T-F: propo-rew-step elim-imp  $\varphi \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$ 
proof (induct rule: propo-rew-step.induct)
  case (global-rel  $\varphi' \psi'$ )
  thus no-T-F  $\psi'$ 
    using elim-imp.cases no-T-F-comp-not no-T-F-decomp(1,2)
    by (metis no-T-F-comp-expanded-explicit(2))
next
  case (propo-rew-one-step-lift  $\varphi \varphi' c \xi \xi'$ )
  note rel = this(1) and IH = this(2) and corr = this(3) and no-T-F = this(4)
  {
    assume c: c = CNot
    hence empty:  $\xi = [] \ \xi' = []$  using corr by auto
    hence no-T-F  $\varphi$  using no-T-F c no-T-F-decomp-not by auto
    hence no-T-F (conn c ( $\xi @ \varphi' \# \xi'$ )) using c empty no-T-F-comp-not IH by auto
  }
  moreover {
    assume c: c  $\in$  binary-connectives
    then obtain a b where ab:  $\xi @ \varphi \# \xi' = [a, b]$ 
      using corr list-length2-decomp wf-conn-bin-list-length by metis
    hence  $\varphi$ :  $\varphi = a \vee \varphi = b$ 
      by (metis append-self-conv2 wf-conn-list-decomp(4) wf-conn-unary list.discI list.sel(3)
        nth-Cons-0 tl-append2)
    have  $\zeta$ :  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \text{no-T-F } \zeta$  using ab c propo-rew-one-step-lift.prems by auto

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

```

```

lemma elim-imp-inv':
  fixes  $\varphi \psi :: 'v \text{ propo}$ 
  assumes full (propo-rew-step elim-imp)  $\varphi \psi$  and no-T-F-except-top-level  $\varphi$ 
  shows no-T-F-except-top-level  $\psi$ 
proof –
  {

```

```

{
  fix  $\varphi \psi :: 'v \text{ propo}$ 
  have  $H: \text{elim-imp } \varphi \psi \implies \text{no-T-F-except-top-level } \varphi \implies \text{no-T-F-except-top-level } \psi$ 
    by (induct  $\varphi \psi$  rule: elim-imp.induct, auto)
} note  $H = \text{this}$ 
fix  $\varphi \psi :: 'v \text{ propo}$ 
have  $\text{propo-rew-step elim-imp } \varphi \psi \implies \text{no-T-F-except-top-level } \varphi \implies \text{no-T-F-except-top-level } \psi$ 
  proof -
    assume rel:  $\text{propo-rew-step elim-imp } \varphi \psi$ 
    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)
      by (cases rule: elim-imp.cases, auto simp add: wf-conn-list(1,2))
      hence  $\text{no-T-F-except-top-level } \psi$  by blast
    }
    moreover {
      assume  $\varphi \neq FT \wedge \varphi \neq FF$ 
      hence  $\text{no-T-F } \varphi$  by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
      hence  $\text{no-T-F } \psi$  using rel propo-rew-step-ElimImp-no-T-F by blast
      hence  $\text{no-T-F-except-top-level } \psi$  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:  $\text{propo-rew-step elim-imp } \zeta \zeta'$ 
  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  $\text{no-T-F-symb-except-toplevel } (\text{conn } c (\xi @ \zeta' \# \xi'))$ 
  proof
    have p:  $\text{no-T-F-symb } (\text{conn } c (\xi @ \zeta \# \xi'))$ 
      by (simp add: corr no-T-F no-T-F-symb-except-toplevel-no-T-F-symb wf-conn-list(1,2))

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

```

qed

10.3 The new CNF and DNF transformation

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

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

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

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

Given all the lemmas before the CNF transformation is easy to prove:

definition *cnf-rew'* :: 'a propo \Rightarrow 'a propo \Rightarrow bool **where** *cnf-rew'* \equiv
 (full (propo-rew-step elimTBFULL)) OO
 (full (propo-rew-step elim-equiv)) OO
 (full (propo-rew-step elim-imp)) OO
 (full (propo-rew-step pushNeg)) OO
 (full (propo-rew-step pushDisj))

lemma *cnf-rew'-consistent: preserves-un-sat cnf-rew'*
by (simp add: cnf-rew'-def elimEqv-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**
 $\text{consistent-interp } I = (\forall L. \neg(L \in I \wedge \neg L \in I))$

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

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

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

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

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

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

11.2.2 Atoms

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

lemma atms-of-msmultiset[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* = *atm-of* ' *C*

lemma *atms-of-ms-empty-set*[*simp*]:
atms-of-ms {} = {}
unfolding *atms-of-ms-def* **by** *auto*

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

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

lemma *atms-of-ms-finite*[*simp*]:
finite ψ s \Rightarrow *finite* (*atms-of-ms* ψ s)
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-union*[*simp*]:
atms-of-ms (ψ s \cup χ s) = *atms-of-ms* ψ s \cup *atms-of-ms* χ s
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-ms-insert*[*simp*]:
atms-of-ms (*insert* ψ s χ s) = *atms-of* ψ s \cup *atms-of-ms* χ s
unfolding *atms-of-ms-def* **by** *auto*

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

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

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

lemma *atms-of-ms-remove-incl*:
shows *atms-of-ms* (*Set.remove* *a* ψ) \subseteq *atms-of-ms* ψ
unfolding *atms-of-ms-def* **by** *auto*

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

lemma *finite-atms-of-ms-remove-subset*[*simp*]:
finite (*atms-of-ms* *A*) \Rightarrow *finite* (*atms-of-ms* (*A* - *C*))
using *atms-of-ms-remove-subset*[*of* *A* *C*] *finite-subset* **by** *blast*

lemma *atms-of-ms-empty-iff*:
atms-of-ms *A* = {} \longleftrightarrow *A* = {{#}} \vee *A* = {}
apply (*rule iffI*)

```

apply (metis (no-types, lifting) atms-empty-iff-empty atms-of-atms-of-ms-mono insert-absorb
  singleton-iff singleton-insert-inj-eq' subsetI subset-empty)
apply auto[]
done

```

lemma *in-implies-atm-of-on-atms-of-ms*:

```

assumes  $L \in \# C$  and  $C \in N$ 
shows  $\text{atm-of } L \in \text{atms-of-ms } N$ 
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  $\{\#A\} + D \in \psi_s$ 
shows  $\text{atm-of } A \in \text{atms-of-ms } \psi_s$ 
using assms by (auto dest: atms-of-atms-of-ms-mono)

```

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

```

 $\text{atms-of-s } (Ia \cup Ib) = \text{atms-of-s } Ia \cup \text{atms-of-s } Ib$ 
unfolding atms-of-s-def by auto

```

lemma *atms-of-s-single[simp]*:

```

 $\text{atms-of-s } \{L\} = \{\text{atm-of } L\}$ 
unfolding atms-of-s-def by auto

```

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

```

 $\text{atms-of-s } (\text{insert } L Ib) = \{\text{atm-of } L\} \cup \text{atms-of-s } Ib$ 
unfolding atms-of-s-def by auto

```

lemma *in-atms-of-s-decomp[iff]*:

```

 $P \in \text{atms-of-s } I \iff (\text{Pos } P \in I \vee \text{Neg } P \in I) \text{ (is } ?P \iff ?Q)$ 

```

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  $I S = (\forall l \in S. \text{Pos } l \in I \vee \text{Neg } l \in I)$ 

```

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]*:

$total-over-set\ I\ \{\}$
unfolding $total-over-set-def$ **by** *auto*

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

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

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

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

lemma $total-over-m-subset$:
 $A \subseteq B \implies total-over-m\ I\ B \implies total-over-m\ I\ A$
using $atms-of-ms-mono[of\ A]$ **unfolding** $total-over-m-def\ total-over-set-def$ **by** *auto*

lemma $total-over-m-sum[iff]$:
shows $total-over-m\ I\ \{C + D\} \longleftrightarrow (total-over-m\ I\ \{C\} \wedge total-over-m\ I\ \{D\})$
using $assms$ **unfolding** $total-over-m-def\ total-over-set-def$ **by** *auto*

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

lemma $total-over-m-insert[iff]$:
 $total-over-m\ I\ (insert\ a\ A) \longleftrightarrow (total-over-set\ I\ (atms-of\ a) \wedge total-over-m\ I\ A)$
unfolding $total-over-m-def\ total-over-set-def$ **by** *fastforce*

lemma $total-over-m-extension$:
fixes $I :: 'v\ literal\ set$ **and** $A :: 'v\ clauses$
assumes $total$: $total-over-m\ I\ A$
shows $\exists I'. total-over-m\ (I \cup I')\ (A \cup B)$
 $\wedge (\forall x \in I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A)$
proof –
let $?I' = \{Pos\ v \mid v. v \in atms-of-ms\ B \wedge v \notin atms-of-ms\ A\}$
have $(\forall x \in ?I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A)$ **by** *auto*
moreover have $total-over-m\ (I \cup ?I')\ (A \cup B)$
using $total$ **unfolding** $total-over-m-def\ total-over-set-def$ **by** *auto*
ultimately show $?thesis$ **by** *blast*
qed

lemma $total-over-m-consistent-extension$:
fixes $I :: 'v\ literal\ set$ **and** $A :: 'v\ clauses$
assumes $total$: $total-over-m\ I\ A$
and $cons$: $consistent-interp\ I$
shows $\exists I'. total-over-m\ (I \cup I')\ (A \cup B)$
 $\wedge (\forall x \in I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A) \wedge consistent-interp\ (I \cup I')$
proof –

let $?I' = \{Pos\ v \mid v. v \in \text{atms-of-ms } B \wedge v \notin \text{atms-of-ms } A \wedge Pos\ v \notin I \wedge Neg\ v \notin I\}$
 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** $(intro\ allI)\ (case-tac\ L,\ auto)$
 ultimately show *?thesis* **by** *blast*
qed

lemma *total-over-set-atms-of[simp]*:
total-over-set Ia (atms-of-s Ia)
unfolding *total-over-set-def atms-of-s-def* **by** $(metis\ image-iff\ literal.exhaust-sel)$

lemma *total-over-set-literal-defined*:
assumes $\{\#A\# \} + D \in \psi_s$
and *total-over-set I (atms-of-ms ψ_s)*
shows $A \in I \vee -A \in I$
using *assms unfolding total-over-set-def* **by** $(metis\ (no-types)\ Neg-atm-of-iff\ in-m-in-literals\ literal.collapse(1)\ uminus-Neg\ uminus-Pos)$

lemma *tot-over-m-remove*:
assumes *total-over-m* $(I \cup \{L\})\ \{\psi\}$
and $L: \neg L \in \# \psi \neg L \notin \# \psi$
shows *total-over-m I* $\{\psi\}$
unfolding *total-over-m-def total-over-set-def*

proof

fix l
assume $l: l \in \text{atms-of-ms } \{\psi\}$
then have $Pos\ l \in I \vee Neg\ l \in I \vee l = \text{atm-of } L$
 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 $Pos\ (\text{atm-of } L) \in \# \psi \vee Neg\ (\text{atm-of } L) \in \# \psi$
 using *atm-imp-pos-or-neg-lit* **by** *metis*
 then have $L \in \# \psi \vee -L \in \# \psi$ **by** $(case-tac\ L)\ auto$
 then show *False* **using** L **by** *auto*
 qed
ultimately show $Pos\ l \in I \vee Neg\ l \in I$ **using** l **by** *metis*
qed

lemma *total-union*:
assumes *total-over-m I ψ*
shows *total-over-m* $(I \cup I')\ \psi$
using *assms unfolding total-over-m-def total-over-set-def* **by** *auto*

lemma *total-union-2*:
assumes *total-over-m I ψ*
and *total-over-m I' ψ'*
shows *total-over-m* $(I \cup I')\ (\psi \cup \psi')$
using *assms unfolding total-over-m-def total-over-set-def* **by** *auto*

11.2.4 Interpretations

definition *true-cls* :: $'a\ interp \Rightarrow 'a\ clause \Rightarrow bool$ (*infix* $\models 50$) **where**

$I \models C \longleftrightarrow (\exists L \in \# C. I \models_l L)$

lemma *true-cls-empty*[*iff*]: $\neg I \models \{\#\}$
unfolding *true-cls-def* **by** *auto*

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

lemma *true-cls-union*[*iff*]: $I \models C + D \longleftrightarrow I \models C \vee I \models D$
unfolding *true-cls-def* **by** *auto*

lemma *true-cls-mono-set-mset*: $\text{set-mset } C \subseteq \text{set-mset } D \implies I \models C \implies I \models D$
unfolding *true-cls-def subset-eq Bex-mset-def* **by** (*metis mem-set-mset-iff*)

lemma *true-cls-mono-leD*[*dest*]: $A \subseteq \# B \implies I \models A \implies I \models B$
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* :: $'a \text{ interp} \Rightarrow 'a \text{ clauses} \Rightarrow \text{bool}$ (**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-clss-def* **by** *auto*

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

lemma *true-clss-insert*[*iff*]: $I \models_s \text{insert } C \ DD \longleftrightarrow I \models C \wedge I \models_s DD$
unfolding *true-clss-def* **by** *blast*

lemma *true-clss-mono*: $DD \subseteq CC \implies I \models_s CC \implies I \models_s DD$
unfolding *true-clss-def* **by** *blast*

lemma *true-clss-union-increase*[*simp*]:
assumes $I \models_s \psi$
shows $I \cup I' \models_s \psi$
using *assms* **unfolding** *true-clss-def* **by** *auto*

lemma *true-clss-union-increase'*[*simp*]:
assumes $I' \models_s \psi$
shows $I \cup I' \models_s \psi$
using *assms* **by** (*auto simp add: true-clss-def*)

lemma *true-clss-commute-l*:
 $(I \cup I' \models_s \psi) \longleftrightarrow (I' \cup I \models_s \psi)$
by (*simp add: Un-commute*)

lemma *model-remove*[*simp*]: $I \models_s N \implies I \models_s \text{Set.remove } a \ N$
by (*simp add: true-clss-def*)

lemma *model-remove-minus*[*simp*]: $I \models_s N \implies I \models_s N - A$
by (*simp add: true-clss-def*)

lemma *notin-vars-union-true-clss-true-clss*:
assumes $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-ms } A$
and $\text{atms-of } L \subseteq \text{atms-of-ms } A$
and $I \cup I' \models L$
shows $I \models L$
using *assms* **unfolding** *true-clss-def true-lit-def Bex-mset-def*
by (*metis Un-iff atm-of-lit-in-atms-of contra-subsetD*)

lemma *notin-vars-union-true-clss-true-clss*:
assumes $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-ms } A$
and $\text{atms-of-ms } L \subseteq \text{atms-of-ms } A$
and $I \cup I' \models_s L$
shows $I \models_s L$
using *assms* **unfolding** *true-clss-def true-lit-def Ball-def*
by (*meson atms-of-atms-of-ms-mono notin-vars-union-true-clss-true-clss subset-trans*)

11.2.5 Satisfiability

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

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

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

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

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

proof

assume *?B* **then show** *?sat* **by** (*auto simp add: satisfiable-def*)

next

assume *?sat*

then obtain *I* **where**

I-CC: $I \models_s CC$ **and**

cons: *consistent-interp* *I* **and**

tot: *total-over-m* *I* *CC*

unfolding *satisfiable-def* **by** *auto*

let *?I* = $\{P. P \in I \wedge \text{atm-of } P \in \text{atms-of-ms } CC\}$

have *I-CC*: $?I \models_s CC$

using *I-CC* **unfolding** *true-clss-def Ball-def true-cls-def Bex-mset-def true-lit-def*

by (*smt atm-of-lit-in-atms-of atms-of-atms-of-ms-mono mem-Collect-eq subset-eq*)

moreover have *cons*: *consistent-interp* *?I*

using *cons* **unfolding** *consistent-interp-def* **by** *auto*

moreover have *total-over-m* *?I* *CC*

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

moreover

have *atms-CC-incl*: $\text{atms-of-ms } CC \subseteq \text{atm-of } I$

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: split-if-asm*)

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

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

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

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

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

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

theorem *true-clss-remove-unused*:
assumes $I \models_s \psi$
shows $\{v \in I. atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\} \models_s \psi$
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 $\{v \in I. atm\text{-}of\ v \in atms\text{-}of\ x\} \models x$
by (*simp only: true-cls-remove-unused[of I]*)
moreover have $\{v \in I. atm\text{-}of\ v \in atms\text{-}of\ x\} \subseteq \{v \in I. atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\}$
using $\langle x \in \psi \rangle$ **by** (*auto simp add: atms-of-ms-def*)
ultimately show $\{v \in I. atm\text{-}of\ v \in atms\text{-}of\text{-}ms\ \psi\} \models x$
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 $H: \forall v \in I'. atm\text{-}of\ v \notin atms\text{-}of\ \psi$
shows $I \models \psi$

proof –
let $?I = \{v \in I \cup I'. atm\text{-}of\ v \in atms\text{-}of\ \psi\}$
have $?I \models \psi$ **using** *true-cls-remove-unused II'* **by** *blast*
moreover have $?I \subseteq I$ **using** H **by** *auto*
ultimately show *?thesis* **using** *true-cls-mono-set-mset-l* **by** *blast*
qed

lemma *multiset-not-empty*:
assumes $M \neq \{\#\}$
and $x \in \# M$
shows $\exists A. x = Pos\ A \vee x = Neg\ A$
using *assms literal.exhaust-sel* **by** *blast*

lemma *atms-of-ms-empty*:
fixes $\psi :: 'v\ \text{clauses}$
assumes $atms\text{-}of\text{-}ms\ \psi = \{\}$

shows $\psi = \{\} \vee \psi = \{\{\#\}\}$
using *assms* **by** (*auto simp add: atms-of-ms-def*)

lemma *consistent-interp-disjoint*:
assumes *consI*: *consistent-interp I*
and *disj*: *atms-of-s A* \cap *atms-of-s I* = $\{\}$
and *consA*: *consistent-interp A*
shows *consistent-interp (A \cup I)*
proof (*rule ccontr*)
assume \neg *?thesis*
moreover have $\bigwedge L. \neg (L \in A \wedge \neg L \in I)$
using *disj unfolding atms-of-s-def* **by** (*auto simp add: rev-image-eqI*)
ultimately show *False*
using *consA consI unfolding consistent-interp-def* **by** (*metis (full-types) Un-iff literal.exhaust-sel uminus-Neg uminus-Pos*)
qed

lemma *total-remove-unused*:
assumes *total-over-m I* ψ
shows *total-over-m* $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\}$ ψ
using *assms unfolding total-over-m-def total-over-set-def*
by (*metis (lifting) literal.sel(1,2) mem-Collect-eq*)

lemma *true-cls-remove-hd-if-notin-vars*:
assumes *insert a M' \models D*
and *atm-of a \notin atms-of D*
shows *M' \models D*
using *assms* **by** (*auto simp add: atm-of-lit-in-atms-of true-cls-def*)

lemma *total-over-set-atm-of*:
fixes *I* :: '*v* *interp* and *K* :: '*v* *set*
shows *total-over-set I K* $\longleftrightarrow (\forall l \in K. l \in (\text{atm-of } I))$
unfolding *total-over-set-def* **by** (*metis atms-of-s-def in-atms-of-s-decomp*)

11.2.7 Tautologies

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

lemma *tautology-Pos-Neg[intro]*:
assumes *Pos* $p \in \# A$ **and** *Neg* $p \in \# A$
shows *tautology A*
using *assms unfolding tautology-def total-over-set-def true-cls-def Bex-mset-def*
by (*meson atm-iff-pos-or-neg-lit true-lit-def*)

lemma *tautology-minus[simp]*:
assumes *L* $\in \# A$ **and** $\neg L \in \# A$
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-mset-def* **apply** *clarify*
using *p* **by** (*case-tac L*) *fastforce+*
ultimately show *False* **using** *assms* **unfolding** *tautology-def* **by** *auto*
qed

lemma *tautology-decomp*:
 $\text{tautology } \psi \longleftrightarrow (\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi)$
using *tautology-exists-Pos-Neg* **by** *auto*

lemma *tautology-false[simp]*: $\neg \text{tautology } \{\#\}$
unfolding *tautology-def* **by** *auto*

lemma *tautology-add-single*:
 $\text{tautology } (\{\#a\} + L) \longleftrightarrow \text{tautology } L \vee -a \in \# L$
unfolding *tautology-decomp* **by** (*cases a*) *auto*

lemma *minus-interp-tautology*:
assumes $\{-L \mid L. L \in \# \chi\} \models \chi$
shows *tautology* 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* (*atms-of* χ')
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**
 $\psi \models_f \chi \longleftrightarrow (\forall I. \text{total-over-m } I (\{\psi\} \cup \{\chi\}) \longrightarrow \text{consistent-interp } I \longrightarrow I \models \psi \longrightarrow I \models \chi)$

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

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

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

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

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

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

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

lemma *true-clss-clss-true-clss-cl[iff]*:
 $N \models_{ps} \{\psi\} \longleftrightarrow N \models_p \psi$
unfolding *true-clss-clss-def true-clss-cl-def* **by** *auto*

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

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

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

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

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

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

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

```

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

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

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

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

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

lemma true-clss-clss-union-and[iff]:
   $A \models_{ps} C \cup D \longleftrightarrow (A \models_{ps} C \wedge A \models_{ps} D)$ 
proof
{
  fix  $A \ C \ D :: 'a \ \text{clauses}$ 
  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  $\text{tot}AC: \text{total-over-m } I \ (A \cup C)$ 
      and  $\text{cons}: \text{consistent-interp } I$ 
      and  $I: I \models_s A$ 
      then have  $\text{tot}: \text{total-over-m } I \ A$  and  $\text{tot}': \text{total-over-m } I \ C$  by auto
      obtain  $I'$  where  $\text{tot}': \text{total-over-m } (I \cup I') \ (A \cup C \cup D)$ 
      and  $\text{cons}': \text{consistent-interp } (I \cup I')$ 
      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 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  $H = \text{this}$ 
  assume  $A \models_{ps} C \cup D$ 
  then show  $A \models_{ps} C \wedge A \models_{ps} D$  using  $H[\text{of } A]$  Un-commute[of C D] by metis
next
  assume  $A \models_{ps} C \wedge A \models_{ps} D$ 
  then show  $A \models_{ps} C \cup D$ 
    unfolding true-clss-clss-def by auto
qed

lemma true-clss-clss-insert[iff]:
   $A \models_{ps} \text{insert } L \ Ls \longleftrightarrow (A \models_p L \wedge A \models_{ps} Ls)$ 

```


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  $D \langle I \models_s N \rangle\ tot\ \langle consistent-interp\ I \rangle$ 
    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  $I \models D + C$  using  $\langle consistent-interp\ I \rangle\ consistent-interp-def$  by fastforce
}
moreover {
  assume  $L: L \notin I \wedge -L \notin I$ 
  let  $?I' = I \cup \{L\}$ 
  have  $consistent-interp\ ?I'$  using  $L\ \langle consistent-interp\ I \rangle$  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  $?I' \models N$  using  $\langle I \models_s N \rangle$  using  $true-clss-union-increase$  by blast
  ultimately have  $?I' \models D + \{\#- L\#\}$ 
    using  $D$  unfolding  $true-clss-cls-def$  by blast
  then have  $?I' \models D$  using  $L$  by auto
  moreover
    have  $total-over-set\ I\ (atms-of\ (D + C))$  using  $tot$  by auto
    then have  $L \notin \# D \wedge -L \notin \# D$ 
      using  $L$  unfolding  $total-over-set-def\ atms-of-def$  by (cases  $L$ ) force+
  ultimately have  $I \models D + C$  unfolding  $true-cls-def$  by auto
}
ultimately show  $I \models D + C$  by blast
qed

```

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

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

unfolding $atms-of-def$ by (auto simp: $max-def\ split: split-if-asm$)

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

unfolding $true-cls-def$ by (force simp: $max-def\ Bex-mset-def\ split: split-if-asm$)

lemma $true-clss-cls-union-mset-true-clss-cls-or-not-true-clss-cls-or$:

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

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

shows $N \models_p D \# \cup C$

unfolding $true-clss-cls-def$

proof (intro $allI\ impI$)

fix I

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

and $consistent-interp\ I$

and $I \models_s N$

{

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

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

```

    using tot by (cases L) auto
  then have  $I \models D + \{\#- L\# \}$  using  $D \langle I \models s N \rangle$  tot  $\langle \text{consistent-interp } I \rangle$ 
    unfolding true-clss-cls-def by auto
  moreover
    have total-over-m  $I \{C + \{\#L\# \}\}$ 
      using L tot by (cases L) auto
    then have  $I \models C + \{\#L\# \}$ 
      using  $C \langle I \models s N \rangle$  tot  $\langle \text{consistent-interp } I \rangle$  unfolding true-clss-cls-def by auto
  ultimately have  $I \models D \# \cup C$  using  $\langle \text{consistent-interp } I \rangle$  unfolding consistent-interp-def
  by auto
}
moreover {
  assume L:  $L \notin I \wedge -L \notin I$ 
  let  $?I' = I \cup \{L\}$ 
  have consistent-interp  $?I'$  using L  $\langle \text{consistent-interp } I \rangle$  by auto
  moreover have total-over-m  $?I' \{D + \{\#- L\# \}\}$ 
    using tot unfolding total-over-m-def total-over-set-def by (auto simp add: atms-of-def)
  moreover have total-over-m  $?I' N$  using tot using total-union by blast
  moreover have  $?I' \models s N$  using  $\langle I \models s N \rangle$  using true-clss-union-increase by blast
  ultimately have  $?I' \models D + \{\#- L\# \}$ 
    using D unfolding true-clss-cls-def by blast
  then have  $?I' \models D$  using L by auto
  moreover
    have total-over-set  $I$  (atms-of  $(D + C)$ ) using tot by auto
    then have  $L \notin \# D \wedge -L \notin \# D$ 
      using L unfolding total-over-set-def atms-of-def by (cases L) force+
    ultimately have  $I \models D \# \cup C$  unfolding true-clss-def by auto
  }
  ultimately show  $I \models D \# \cup C$  by blast
qed

```

lemma *satisfiable-carac*[iff]:

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

proof

assume $?S$

then show $\exists I. ?Q I$ unfolding satisfiable-def by auto

next

assume $\exists I. ?Q I$

then obtain I where cons: consistent-interp I and $I: I \models s \varphi$ by metis

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

have consistent-interp $(I \cup ?I')$

using cons unfolding consistent-interp-def by (intro allI) (case-tac L, auto)

moreover have total-over-m $(I \cup ?I') \varphi$

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

moreover have $I \cup ?I' \models s \varphi$

using I unfolding Ball-def true-clss-def true-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 $\text{total-over-}m \ I \ \{B\} \implies \text{total-over-}m \ I \ \{A\}$
using *assms unfolding subset-mset-def total-over-m-def total-over-set-def*
by (*auto simp add: mset-le-exists-conv*)

lemma *atm-of-eq-atm-of*:
 $\text{atm-of } L = \text{atm-of } L' \iff (L = L' \vee L = -L')$
by (*cases L; cases L'*) *auto*

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 (*auto split: split-if-asm simp add: atm-of-eq-atm-of atms-of-def*)

lemma *subsumption-chained*:
assumes $\forall I. \text{total-over-}m \ I \ \{D\} \longrightarrow I \models D \longrightarrow I \models \varphi$
and $C \subseteq\# D$
shows $(\forall I. \text{total-over-}m \ I \ \{C\} \longrightarrow I \models C \longrightarrow I \models \varphi) \vee \text{tautology } \varphi$
using *assms*

proof (*induct card {Pos v | v. v ∈ atms-of D ∧ v ∉ atms-of C} arbitrary: D*
rule: nat-less-induct-case)

case 0 **note** $n = \text{this}(1)$ **and** $H = \text{this}(2)$ **and** $\text{incl} = \text{this}(3)$
then have $\text{atms-of } D \subseteq \text{atms-of } C$ **by** *auto*
then have $\forall I. \text{total-over-}m \ I \ \{C\} \longrightarrow \text{total-over-}m \ I \ \{D\}$
unfolding *total-over-m-def total-over-set-def* **by** *auto*
moreover have $\forall I. I \models C \longrightarrow I \models D$ **using** *incl true-cls-mono-leD* **by** *blast*
ultimately show *?case* **using** H **by** *auto*

next

case (*Suc n D*) **note** $IH = \text{this}(1)$ **and** $\text{card} = \text{this}(2)$ **and** $H = \text{this}(3)$ **and** $\text{incl} = \text{this}(4)$
let $\text{?atms} = \{\text{Pos } v \mid v. v \in \text{atms-of } D \wedge v \notin \text{atms-of } C\}$
have *finite ?atms* **by** *auto*
then obtain L **where** $L: L \in \text{?atms}$
using *card* **by** (*metis (no-types, lifting) Collect-empty-eq card-0-eq mem-Collect-eq*
nat.simps(3))
let $\text{?D}' = D - \text{replicate-mset } (\text{count } D \ L) \ L - \text{replicate-mset } (\text{count } D \ (-L)) \ (-L)$
have $\text{atms-of-}D: \text{atms-of-}ms \ \{D\} \subseteq \text{atms-of-}ms \ \{\text{?D}'\} \cup \{\text{atm-of } L\}$ **by** *auto*

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

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

moreover

have $\text{tot': total-over-}m \ (I \cup \{-L\}) \ \{D\}$
using *tot* **unfolding** *total-over-m-def total-over-set-def* **by** *auto*
have $I \cup \{-L\} \models D$ **using** IDL **unfolding** *true-cls-def* **by** *force*
then have $I \cup \{-L\} \models \varphi$ **using** $H \text{ tot'}$ **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  $C$ -in- $D'$ :  $C \subseteq \# ?D'$  using  $\langle C \subseteq \# D \rangle$  by (auto simp add: subseteq-mset-def)
have card  $\{Pos\ v \mid v. v \in \text{atms-of } ?D' \wedge v \notin \text{atms-of } C\} <$ 
  card  $\{Pos\ v \mid v. v \in \text{atms-of } D \wedge v \notin \text{atms-of } C\}$ 
  using  $L$  by (auto intro!: psubset-card-mono)
then show ?case
  using IH  $C$ -in- $D'$   $H'$  unfolding card[symmetric] by blast
qed

```

11.4 Removing Duplicates

```

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

```

```

lemma atms-of-remdups-mset[simp]: 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

A simple clause contains no duplicate and is not tautology.

```

function build-all-simple-clss :: 'v :: linorder set  $\Rightarrow$  'v clause set where
  build-all-simple-clss vars =
    (if  $\neg \text{finite vars} \vee \text{vars} = \{\}$ 
     then  $\{\{\#\}\}$ 
     else
       let  $cls' = \text{build-all-simple-clss } (\text{vars} - \{\text{Min vars}\})$  in
        $\{\{\#Pos\ (\text{Min vars})\#\} + \chi \mid \chi. \chi \in cls'\} \cup$ 
        $\{\{\#Neg\ (\text{Min vars})\#\} + \chi \mid \chi. \chi \in cls'\} \cup$ 
        $cls')$ 
  by auto
termination by (relation measure card) (auto simp add: card-gt-0-iff)

```

To avoid infinite simplifier loops:

```

declare build-all-simple-clss.simps[simp del]

```

```

lemma build-all-simple-clss-simps-if[simp]:
   $\neg \text{finite vars} \vee \text{vars} = \{\} \implies \text{build-all-simple-clss vars} = \{\{\#\}\}$ 
  by (simp add: build-all-simple-clss.simps)

```

```

lemma build-all-simple-clss-simps-else[simp]:
  fixes vars::'v :: linorder set
  defines  $cls \equiv \text{build-all-simple-clss } (\text{vars} - \{\text{Min vars}\})$ 
  shows
     $\text{finite vars} \wedge \text{vars} \neq \{\} \implies \text{build-all-simple-clss } (\text{vars}::'v :: \text{linorder set}) =$ 
     $\{\{\#Pos\ (\text{Min vars})\#\} + \chi \mid \chi. \chi \in cls\}$ 
     $\cup \{\{\#Neg\ (\text{Min vars})\#\} + \chi \mid \chi. \chi \in cls\}$ 
     $\cup cls$ 
  using build-all-simple-clss.simps[of vars] unfolding Let-def cls-def by metis

```

```

lemma build-all-simple-clss-finite:
  fixes atms :: 'v::linorder set
  shows finite (build-all-simple-clss atms)
proof (induct card atms arbitrary: atms rule: nat-less-induct)
  case (1 atms) note IH = this
  {
    assume atms = {}  $\vee$   $\neg$ finite atms
    then have finite (build-all-simple-clss atms) by auto
  }
  moreover {
    assume atms: atms  $\neq$  {} and fin: finite atms
    then have Min atms  $\in$  atms using Min-in by auto
    then have card (atms - {Min atms}) < card atms using fin atms by (meson card-Diff1-less)
    then have finite (build-all-simple-clss (atms - {Min atms})) using IH by auto
    then have finite (build-all-simple-clss atms) by (simp add: atms fin)
  }
  ultimately show finite (build-all-simple-clss atms) by blast
qed

```

```

lemma build-all-simple-clssE:
  assumes
     $x \in$  build-all-simple-clss atms and
    finite atms
  shows atms-of  $x \subseteq$  atms  $\wedge$   $\neg$ tautology  $x \wedge$  distinct-mset  $x$ 
  using assms
proof (induct card atms arbitrary: atms x)
  case (0 atms)
  then show ?case by auto
next
  case (Suc n) note IH = this(1) and card = this(2) and  $x =$  this(3) and finite = this(4)
  obtain v where  $v \in$  atms and  $v$ :  $v =$  Min atms
    using Min-in card local.finite by fastforce

  let ?atms' = atms - {v}
  have build-all-simple-clss atms
    = {{#Pos v#} +  $\chi$  |  $\chi \in$  build-all-simple-clss (?atms')}
       $\cup$  {{#Neg v#} +  $\chi$  |  $\chi \in$  build-all-simple-clss (?atms')}
       $\cup$  build-all-simple-clss (?atms')
    using build-all-simple-clss-simps-else[of atms] finite  $\langle v \in$  atms  $\rangle$  unfolding v
    by (metis emptyE)
  then consider
    (Pos)  $\chi$   $\varphi$  where  $x =$  {# $\varphi$ #} +  $\chi$  and  $\chi \in$  build-all-simple-clss (?atms') and
     $\varphi =$  Pos v  $\vee$   $\varphi =$  Neg v
    | (In)  $x \in$  build-all-simple-clss (?atms')
    using x by auto
  then show ?case
  proof cases
    case In
    then show ?thesis using card finite IH[of ?atms']  $\langle v \in$  atms  $\rangle$  by fastforce
  next
    case Pos note  $x$ - $\chi =$  this(1) and  $\chi =$  this(2) and  $\varphi =$  this(3)
    have
      atms-of  $\chi \subseteq$  atms - {v} and
       $\neg$  tautology  $\chi$  and

```

```

distinct-mset  $\chi$ 
  using card finite IH[of ?atms'  $\chi$ ]  $\langle v \in \text{atms} \rangle x\text{-}\chi$   $\chi$  by auto
moreover then have count  $\chi$  (Neg  $v$ ) = 0
  using  $\langle v \in \text{atms} \rangle$  unfolding  $x\text{-}\chi$  by (metis Diff-insert-absorb Set.set-insert
    atm-iff-pos-or-neg-lit gr0I subset-iff)
moreover have count  $\chi$  (Pos  $v$ ) = 0
  using  $\langle \text{atms-of } \chi \subseteq \text{atms} - \{v\} \rangle$  by (meson Diff-iff atm-iff-pos-or-neg-lit
    contra-subsetD insertI1 not-gr0)
ultimately show ?thesis
  using  $\langle v \in \text{atms} \rangle \varphi$  unfolding  $x\text{-}\chi$ 
  by (auto simp add: tautology-add-single distinct-mset-add-single)
qed
qed

lemma cls-in-build-all-simple-clss:
  shows  $\{\#\} \in \text{build-all-simple-clss } s$ 
  by (induct  $s$  rule: build-all-simple-clss.induct)
  (metis (no-types, lifting) UnCI build-all-simple-clss.simps insertI1)

lemma build-all-simple-clss-card:
  fixes  $\text{atms} :: 'v :: \text{linorder set}$ 
  assumes finite  $\text{atms}$ 
  shows card (build-all-simple-clss  $\text{atms}$ )  $\leq 3^{\wedge}(\text{card } \text{atms})$ 
  using assms
proof (induct card  $\text{atms}$  arbitrary:  $\text{atms}$  rule: nat-less-induct)
  case (1  $\text{atms}$ ) note IH = this(1) and finite = this(2)
  {
    assume  $\text{atms} = \{\}$ 
    then have card (build-all-simple-clss  $\text{atms}$ )  $\leq 3^{\wedge}(\text{card } \text{atms})$  by auto
  }
  moreover {
    let ?P =  $\{\{\#\text{Pos } (\text{Min } \text{atms})\#\} + \chi \mid \chi \in \text{build-all-simple-clss } (\text{atms} - \{\text{Min } \text{atms}\})\}$ 
    let ?N =  $\{\{\#\text{Neg } (\text{Min } \text{atms})\#\} + \chi \mid \chi \in \text{build-all-simple-clss } (\text{atms} - \{\text{Min } \text{atms}\})\}$ 
    let ?Z = build-all-simple-clss  $(\text{atms} - \{\text{Min } \text{atms}\})$ 
    assume  $\text{atms} \neq \{\}$ 
    then have min:  $\text{Min } \text{atms} \in \text{atms}$  using Min-in finite by auto
    then have card-atms-1: card  $\text{atms} \geq 1$  by (simp add: Suc-leI  $\text{atms}$  card-gt-0-iff local.finite)
    have card (build-all-simple-clss  $\text{atms}$ ) = card (?P  $\cup$  ?N  $\cup$  ?Z) using  $\text{atms}$  finite by simp
    moreover
      have  $\bigwedge M \text{ Ma. card } ((M :: 'v \text{ literal multiset set}) \cup \text{Ma}) \leq \text{card } \text{Ma} + \text{card } M$ 
      by (simp add: add commute card-Un-le)
    then have card (?P  $\cup$  ?N  $\cup$  ?Z)  $\leq$  card ?Z + (card ?P + card ?N)
      by (meson Nat.le-trans card-Un-le nat-add-left-cancel-le)
    then have card (?P  $\cup$  ?N  $\cup$  ?Z)  $\leq$  card ?P + card ?N + card ?Z

    by presburger
  }
  also
    have PZ: card ?P  $\leq$  card ?Z
      by (simp add: Setcompr-eq-image build-all-simple-clss-finite card-image-le)
    have NZ: card ?N  $\leq$  card ?Z
      by (simp add: Setcompr-eq-image build-all-simple-clss-finite card-image-le)
    have card ?P + card ?N + card ?Z  $\leq$  card ?Z + card ?Z + card ?Z
      using PZ NZ by linarith
  finally have card (build-all-simple-clss  $\text{atms}$ )  $\leq$  card ?Z + card ?Z + card ?Z .
  moreover

```

```

have finite': finite (atms - {Min atms}) and
  card: card (atms - {Min atms}) = card atms - 1
  using finite min by auto
have card-inf: card (atms - {Min atms}) < card atms
  using card (card atms ≥ 1) min by auto
then have card ?Z ≤ 3 ^ (card atms - 1) using IH finite' card by metis
moreover
  have (3::nat) ^ (card atms - 1) + 3 ^ (card atms - 1) + 3 ^ (card atms - 1)
    = 3 * 3 ^ (card atms - 1) by simp
  then have (3::nat) ^ (card atms - 1) + 3 ^ (card atms - 1) + 3 ^ (card atms - 1)
    = 3 ^ (card atms) by (metis card card-Suc-Diff1 local.finite min power-Suc)
  ultimately have card (build-all-simple-clss atms) ≤ 3 ^ (card atms) by linarith
}
ultimately show card (build-all-simple-clss atms) ≤ 3 ^ (card atms) by metis
qed

lemma build-all-simple-clss-mono-disj:
  assumes atms ∩ atms' = {} and finite atms and finite atms'
  shows build-all-simple-clss atms ⊆ build-all-simple-clss (atms ∪ atms')
  using assms
proof (induct card (atms ∪ atms') arbitrary: atms atms')
  case (0 atms' atms)
  then show ?case by auto
next
  case (Suc n atms atms') note IH = this(1) and c = this(2) and disj = this(3) and finite = this(4)
  and finite' = this(5)
  let ?min = Min (atms ∪ atms')
  have m: ?min ∈ atms ∨ ?min ∈ atms' by (metis Min-in Un-iff c card-eq-0-iff nat.distinct(1))
  moreover {
    assume min: ?min ∈ atms'
    then have min': ?min ∉ atms using disj by auto
    then have atms = atms - {?min} by fastforce
    then have n = card (atms ∪ (atms' - {?min}))
      using c min finite finite' by (metis Min-in Un-Diff card-Diff-singleton-if diff-Suc-1
        finite-UnI sup-eq-bot-iff)
    moreover have atms ∩ (atms' - {?min}) = {} using disj by auto
    moreover have finite (atms' - {?min}) using finite' by auto
    ultimately have build-all-simple-clss atms ⊆ build-all-simple-clss (atms ∪ (atms' - {?min}))
      using IH[of atms atms' - {?min}] finite by metis
    moreover have atms ∪ (atms' - {?min}) = (atms ∪ atms') - {?min} using min min' by auto
    ultimately have ?case by (metis (no-types, lifting) build-all-simple-clss.simps c card-0-eq
      finite' finite-UnI le-supI2 local.finite nat.distinct(1))
  }
  moreover {
    let ?atms' = atms - {Min atms}
    assume min: ?min ∈ atms
    moreover have min': ?min ∉ atms' using disj min by auto
    moreover have atms' - {?min} = atms'
      using (?min ∉ atms') by fastforce
    ultimately have n = card (atms - {?min} ∪ atms')
      by (metis Min-in Un-Diff c card-0-eq card-Diff-singleton-if diff-Suc-1 finite' finite-Un
        finite nat.distinct(1))
    moreover have finite (atms - {?min}) using finite by auto
    moreover have (atms - {?min}) ∩ atms' = {} using disj by auto
    ultimately have build-all-simple-clss (atms - {?min})

```



```

    ⊆ build-all-simple-clss ((atms - {?min}) ∪ atms')
    using IH[of atms - {?min} atms'] finite' by metis
  moreover have build-all-simple-clss atms
    = {{#Pos (Min atms)#} + χ | χ. χ ∈ build-all-simple-clss (?atms')}
      ∪ {{#Neg (Min atms)#} + χ | χ. χ ∈ build-all-simple-clss (?atms')}
      ∪ build-all-simple-clss (?atms')
    using build-all-simple-clss-simps-else[of atms] finite min by (metis emptyE)
  moreover
    let ?mcls = build-all-simple-clss (atms ∪ atms' - {?min})
    have build-all-simple-clss (atms ∪ atms')
      = {{#Pos (?min)#} + χ | χ. χ ∈ ?mcls} ∪ {{#Neg (?min)#} + χ | χ. χ ∈ ?mcls} ∪ ?mcls
    using build-all-simple-clss-simps-else[of atms ∪ atms'] finite' min
    by (metis c card-eq-0-iff nat.distinct(1))
  moreover have atms ∪ atms' - {?min} = atms - {?min} ∪ atms'
    using min min' by (simp add: Un-Diff)
  moreover have Min atms = ?min using min min' by (simp add: Min-eqI finite' local.finite)
  ultimately have ?case by auto
}
ultimately show ?case by metis
qed

```

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

```

  assumes finite: finite atms' and incl: atms ⊆ atms'
  shows build-all-simple-clss atms ⊆ build-all-simple-clss atms'
proof -
  have atms' = atms ∪ (atms' - atms) using incl by auto
  moreover have finite (atms' - atms) using finite by auto
  moreover have atms ∩ (atms' - atms) = {} by auto
  ultimately show ?thesis
    using rev-finite-subset[OF assms] build-all-simple-clss-mono-disj by (metis (no-types))
qed

```

lemma *distinct-mset-not-tautology-implies-in-build-all-simple-clss*:

```

  assumes distinct-mset χ and ¬tautology χ
  shows χ ∈ build-all-simple-clss (atms-of χ)
  using assms
proof (induct card (atms-of χ) arbitrary: χ)
  case 0
  then show ?case by simp
next
  case (Suc n) note IH = this(1) and simp = this(3) and c = this(2) and no-dup = this(4)
  have finite: finite (atms-of χ) by simp

```

with *no-dup atm-iff-pos-or-neg-lit* **obtain** *L* **where**

```

  Lχ: L ∈# χ and
  L-min: atm-of L = Min (atms-of χ) and
  mLχ: ¬ -L ∈# χ
  by (metis Min-in c card-0-eq literal.sel(1,2) nat.distinct(1) tautology-minus)
  then have χL: χ = (χ - {#L#}) + {#L#} by auto
  have atmχ: atms-of χ = atms-of (χ - {#L#}) ∪ {atm-of L}
    using arg-cong[OF χL, of atms-of] by simp

```

have *aχ*: *atms-of* (χ - {#L#}) = (*atms-of* χ) - {*atm-of* L}

proof (*standard, standard*)

fix *v*

```

assume  $a: v \in \text{atms-of } (\chi - \{\#L\# \})$ 
then obtain  $l$  where  $l: v = \text{atm-of } l$  and  $l': l \in \# \chi - \{\#L\# \}$ 
  unfolding atms-of-def by auto
moreover {
  assume  $v = \text{atm-of } L$ 
  then have  $L \in \# \chi - \{\#L\# \} \vee -L \in \# \chi - \{\#L\# \}$ 
    using  $l' l$  by (auto simp add: atm-of-eq-atm-of)
  moreover have  $L \notin \# \chi - \{\#L\# \}$  using  $\langle L \in \# \chi \rangle \text{ simp}$  unfolding distinct-mset-def by auto
  ultimately have False using  $mL\chi$  by auto
}
ultimately show  $v \in \text{atms-of } \chi - \{\text{atm-of } L\}$ 
  by (auto dest: atm-of-lit-in-atms-of split: split-if-asm)
next
  show  $\text{atms-of } \chi - \{\text{atm-of } L\} \subseteq \text{atms-of } (\chi - \{\#L\# \})$  using  $\text{atm}\chi$  by auto
qed

let  $?s' = \text{build-all-simple-clss } (\text{atms-of } (\chi - \{\#L\# \}))$ 
have  $\text{card } (\text{atms-of } (\chi - \{\#L\# \})) = n$ 
  using  $c \text{ finite } a\chi$  by (simp add: L\chi atm-of-lit-in-atms-of)
moreover have distinct-mset  $(\chi - \{\#L\# \})$  using simp by auto
moreover have  $\neg \text{tautology } (\chi - \{\#L\# \})$ 
  by (meson Multiset.diff-le-self mset-leD no-dup tautology-decomp)
ultimately have  $\chi \text{in: } \chi - \{\#L\# \} \in \text{build-all-simple-clss } (\text{atms-of } (\chi - \{\#L\# \}))$ 
  using IH by simp
have  $\chi = \{\#L\# \} + (\chi - \{\#L\# \})$  using  $\chi L$  by (simp add: add.commute)
then show ?case
  using  $\chi \text{in } L\text{-min } a\chi$ 
  by (cases L)
  (auto simp add: build-all-simple-clss.simps[of atms-of \chi] Let-def)
qed

lemma simplified-in-build-all:
  assumes finite  $\psi$  and distinct-mset-set  $\psi$  and  $\forall \chi \in \psi. \neg \text{tautology } \chi$ 
  shows  $\psi \subseteq \text{build-all-simple-clss } (\text{atms-of-ms } \psi)$ 
  using assms
proof (induct rule: finite.induct)
  case emptyI
  then show ?case by simp
next
  case (insertI  $\psi \chi$ ) note finite = this(1) and IH = this(2) and simp = this(3) and tauto = this(4)
  have distinct-mset  $\chi$  and  $\neg \text{tautology } \chi$ 
    using simp tauto unfolding distinct-mset-set-def by auto
  from distinct-mset-not-tautology-implies-in-build-all-simple-clss[OF this]
  have  $\chi: \chi \in \text{build-all-simple-clss } (\text{atms-of } \chi)$  .
  then have  $\psi \subseteq \text{build-all-simple-clss } (\text{atms-of-ms } \psi)$  using IH simp tauto by auto
  moreover
    have  $\text{atms-of-ms } \psi \subseteq \text{atms-of-ms } (\text{insert } \chi \psi)$  unfolding atms-of-ms-def atm-of-def by force
  ultimately
    have  $\psi \subseteq \text{build-all-simple-clss } (\text{atms-of-ms } (\text{insert } \chi \psi))$ 
      by (meson atms-of-ms-finite build-all-simple-clss-mono dual-order.trans finite.insertI local.finite)
  moreover
    have  $\chi \in \text{build-all-simple-clss } (\text{atms-of-ms } (\text{insert } \chi \psi))$ 
      using  $\chi \text{ finite build-all-simple-clss-mono[of atms-of-ms } (\text{insert } \chi \psi)]$  by auto
  ultimately show ?case by auto

```

qed

11.6 Experiment: Expressing the Entailments as Locales

locale *entail* =

fixes *entail* :: 'a set \Rightarrow 'b \Rightarrow bool (**infix** \models_e 50)

assumes *entail-insert[simp]*: $I \neq \{\} \implies \text{insert } L \ I \models_e x \longleftrightarrow \{L\} \models_e x \vee I \models_e x$

assumes *entail-union[simp]*: $I \models_e A \implies I \cup I' \models_e A$

begin

definition *entails* :: 'a set \Rightarrow 'b set \Rightarrow bool (**infix** \models_{es} 50) **where**

$I \models_{es} A \longleftrightarrow (\forall a \in A. I \models_e a)$

lemma *entails-empty[simp]*:

$I \models_{es} \{\}$

unfolding *entails-def* **by** *auto*

lemma *entails-single[iff]*:

$I \models_{es} \{a\} \longleftrightarrow I \models_e a$

unfolding *entails-def* **by** *auto*

lemma *entails-insert-l[simp]*:

$M \models_{es} A \implies \text{insert } L \ M \models_{es} A$

unfolding *entails-def* **by** (*metis Un-commute entail-union insert-is-Un*)

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

unfolding *entails-def* **by** *blast*

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

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

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

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

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

unfolding *true-clss-ext-def* **by** (*metis sup.orderE true-clss-union-increase'*)

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

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

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

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 $?J = J \cup \{ \text{Pos } (\text{atm-of } P) \mid P. P \in \# C \wedge \text{atm-of } P \notin \text{atm-of } J \}$

have $I \subseteq ?J$ **using** $\langle I \subseteq J \rangle$ **by** *auto*

moreover **have** *consistent-interp* $?J$

using *cons* **unfolding** *consistent-interp-def* **apply** $-$

apply (*rule allI*) **by** (*case-tac L*) (*fastforce simp add: image-iff*) $+$

moreover

have *ex-or-eq*: $\bigwedge l R J. \exists P. (l = P \vee l = -P) \wedge P \in \# C \wedge P \notin J \wedge -P \notin J$

$\iff (l \in \# C \wedge l \notin J \wedge -l \notin J) \vee (-l \in \# C \wedge l \notin J \wedge -l \notin J)$

by (*metis uminus-of-uminus-id*)

have *total-over-m* $?J \ N$

using *tot* **unfolding** *total-over-m-def* *total-over-set-def* *atms-of-ms-def*

apply (*auto simp add: atms-of-def*)

apply (*case-tac a* $\in N - \{C\}$)

apply *auto* \square

using *atms-of-s-def* *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set* **by** *fastforce* $+$

ultimately **have** $?J \models_s N$

using *assms* **unfolding** *true-clss-ext-def* **by** *blast*

then **have** $?J \models_s N - \{C\}$ **by** *auto*

have $\{v \in ?J. \text{atm-of } v \in \text{atms-of-ms } (N - \{C\})\} \subseteq J$

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

by (*auto intro!: rev-image-eqI*)

then **show** $J \models_s N - \{C\}$

using *true-clss-remove-unused*[*OF* $\langle ?J \models_s N - \{C\} \rangle$] **unfolding** *true-clss-def*

by (*meson true-clss-mono-set-mset-l*)

qed

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

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

shows *satisfiable* A

by (*metis Un-empty-left assms satisfiable-carac subset-Un-eq sup.left-idem*

total-over-m-consistent-extension *total-over-m-empty* *true-clss-ext-def*)

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

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

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

```

  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:

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

condensation:

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

subsumption:

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

lemma *simplify-preserves-un-sat*:

fixes *N N'* :: '*v* clauses

assumes *simplify N N'*

and *total-over-m I N*

shows $I \models_s N' \longrightarrow I \models_s N$

using *assms*

proof (*induct rule: simplify.induct*)

case (*tautology-deletion A P*)

then have $I \models A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}$

by (*metis total-over-m-def total-over-set-literal-defined true-clss-singleton true-clss-union true-lit-def uminus-Neg union-commute*)

then show ?*case* **by** (*metis Un-Diff-cancel2 true-clss-singleton true-clss-union*)

next

case (*condensation A P*)

then show ?*case* **by** (*metis Diff-insert-absorb Set.set-insert insertE true-clss-union true-clss-def true-clss-singleton true-clss-union*)

next

case (*subsumption A B*)

have $A \neq B$ **using** *subsumption.hyps(2)* **by** *auto*

then have $I \models_s N - \{B\} \Longrightarrow I \models A$ **using** $\langle A \in N \rangle$ **by** (*simp add: true-clss-def*)

moreover have $I \models A \Longrightarrow I \models B$ **using** $\langle A \subset\# B \rangle$ **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  $N N' :: 'v \text{ clauses}$ 
assumes simplify  $N N'$ 
and total-over-m  $I N$ 
shows  $I \models_s N \longleftrightarrow I \models_s N'$ 
using simplify-preserves-un-sat simplify-preserves-un-sat' assms by blast
```

lemma *simplify-preserves-finite*:

```

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

lemma *rtranclp-simplify-preserves-finite*:

```

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

lemma *simplify-atms-of-ms*:

```

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

lemma *rtranclp-simplify-atms-of-ms*:

```

assumes rtranclp simplify  $\psi \psi'$ 
shows atms-of-ms  $\psi' \subseteq$  atms-of-ms  $\psi$ 
using assms apply (induct rule: rtranclp-induct)
  apply (fastforce intro: simplify-atms-of-ms)
using simplify-atms-of-ms by blast
```

lemma *factoring-imp-simplify*:

```

assumes  $\{\#L\# \} + \{\#L\# \} + C \in N$ 
shows  $\exists N'. \text{simplify } N N'$ 
```

proof –

```

have  $C + \{\#L\# \} + \{\#L\# \} \in N$  using assms by (simp add: add.commute union-lcomm)
from condensation[OF this] show ?thesis by blast
qed
```

12.2 Unconstrained Resolution

type-synonym $'v \text{ uncon-state} = 'v \text{ clauses}$

inductive *uncon-res* :: 'v uncon-state \Rightarrow 'v uncon-state \Rightarrow bool **where**

resolution:

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

$\implies uncon-res\ (N)\ (N \cup \{C + D\}) \mid$

factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \implies uncon-res\ N\ (N \cup \{C + \{\#L\#\}\})$

lemma *uncon-res-increasing*:

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

shows $\psi \in S'$

using *assms* **by** (*induct rule: uncon-res.induct*) *auto*

lemma *rtrancpl-uncon-inference-increasing*:

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

shows $\psi \in S'$

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

12.2.1 Subsumption

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

subsumes $\chi\ \chi' \iff$

$(\forall I. total-over-m\ I\ \{\chi'\} \longrightarrow total-over-m\ I\ \{\chi\})$

$\wedge (\forall I. total-over-m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi')$

lemma *subsumes-refl[simp]*:

subsumes $\chi\ \chi$

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

lemma *subsumes-subsumption*:

assumes *subsumes* *D* χ

and $C \subset\# D$ **and** $\neg tautology\ \chi$

shows *subsumes* *C* χ **unfolding** *subsumes-def*

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

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

lemma *subsumes-tautology*:

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

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

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

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

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

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

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

abbreviation *already-used-inv*

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

already-used-inv *state* \equiv

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

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

assumes *inference-clause* *S S'*

and *already-used-inv* *S*

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

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

by *fastforce*+

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

assumes *inference* *S S'*

and *already-used-inv* *S*

shows *already-used-inv* *S'*

using *assms*

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

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

then show ?*case*

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

qed

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

assumes *rtranclp* *inference* *S S'*

and *already-used-inv* *S*

shows *already-used-inv* *S'*

using *assms* **apply** (*induct* rule: *rtranclp-induct*, *simp*)

using *inference-preserves-already-used-inv* **unfolding** *tautology-def* **by** *fast*

lemma *subsumes-condensation*:

assumes *subsumes* (*C* + {*#L*}) + {*#L*}) *D*

shows *subsumes* (*C* + {*#L*}) *D*

using *assms* **unfolding** *subsumes-def* **by** *simp*

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

assumes *simplify* *N N'*

and *already-used-inv* (*N*, *already-used*)

shows *already-used-inv* (*N'*, *already-used*)

using *assms*

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

case (*condensation* *C L*)

then show ?*case*

using *subsumes-condensation* **by** *simp* *fast*

next

{

fix *a* :: '*a* **and** *A* :: '*a* *set* **and** *P*

have $(\exists x \in \text{Set.remove } a \ A. P \ x) \longleftrightarrow (\exists x \in A. x \neq a \wedge P \ x)$ **by** *auto*

} **note** *ex-member-remove* = *this*

{

fix *a* *a0* :: '*v* *clause* **and** *A* :: '*v* *clauses* **and** *y*


```

assume  $a \in A$  and  $a0 \subset\# a$ 
then have  $(\exists x \in A. \text{subsumes } x y) \longleftrightarrow (\text{subsumes } a y \vee (\exists x \in A. x \neq a \wedge \text{subsumes } x y))$ 
by auto
} note  $tt2 = \text{this}$ 
case  $(\text{subsumption } A B)$  note  $A = \text{this}(1)$  and  $AB = \text{this}(2)$  and  $B = \text{this}(3)$  and  $\text{inv} = \text{this}(4)$ 
show  $?case$ 
proof  $(\text{standard}, \text{standard})$ 
  fix  $x a b$ 
  assume  $x: x \in \text{snd } (N - \{B\}, \text{already-used})$  and  $[simp]: x = (a, b)$ 
  obtain  $p$  where  $p: \text{Pos } p \in\# a \wedge \text{Neg } p \in\# b$  and
     $q: (\exists \chi \in N. \text{subsumes } \chi (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})))$ 
     $\vee \text{tautology } (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \}))$ 
  using  $\text{inv } x$  by  $\text{fastforce}$ 
  consider  $(\text{taut}) \text{tautology } (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})) \mid$ 
     $(\chi) \chi$  where  $\chi \in N$   $\text{subsumes } \chi (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \}))$ 
     $\neg \text{tautology } (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \}))$ 
  using  $q$  by  $\text{auto}$ 
  then show
     $\exists p. \text{Pos } p \in\# a \wedge \text{Neg } p \in\# b$ 
     $\wedge ((\exists \chi \in \text{fst } (N - \{B\}, \text{already-used}). \text{subsumes } \chi (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})))$ 
     $\vee \text{tautology } (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})))$ 
  proof  $\text{cases}$ 
    case  $\text{taut}$ 
    then show  $?thesis$  using  $p$  by  $\text{auto}$ 
  next
    case  $\chi$  note  $H = \text{this}$ 
    show  $?thesis$  using  $p A AB B$   $\text{subsumes-subsumption}[OF - AB H(3)] H(1,2)$  by  $\text{auto}$ 
  qed
qed
next
case  $(\text{tautology-deletion } C P)$ 
then show  $?case$  apply  $\text{clarify}$ 
proof  $-$ 
  fix  $a b$ 
  assume  $C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \} \in N$ 
  assume  $\text{already-used-inv } (N, \text{already-used})$ 
  and  $(a, b) \in \text{snd } (N - \{C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\}, \text{already-used})$ 
  then obtain  $p$  where
     $\text{Pos } p \in\# a \wedge \text{Neg } p \in\# b \wedge$ 
     $((\exists \chi \in \text{fst } (N \cup \{C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\}, \text{already-used}).$ 
     $\text{subsumes } \chi (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})))$ 
     $\vee \text{tautology } (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})))$ 
  by  $\text{fastforce}$ 
  moreover have  $\text{tautology } (C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \})$  by  $\text{auto}$ 
  ultimately show
     $\exists p. \text{Pos } p \in\# a \wedge \text{Neg } p \in\# b$ 
     $\wedge ((\exists \chi \in \text{fst } (N - \{C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\}, \text{already-used}).$ 
     $\text{subsumes } \chi (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})))$ 
     $\vee \text{tautology } (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \})))$ 
  by  $(\text{metis } (\text{no-types}) \text{Diff-iff Un-insert-right empty-iff fst-conv insertE subsumes-tautology sup-bot.right-neutral})$ 
qed
qed

```

lemma

factoring-satisfiable: $I \models \{\#L\# \} + \{\#L\# \} + C \longleftrightarrow I \models \{\#L\# \} + C$ **and**

resolution-satisfiable:

consistent-interp $I \implies I \models \{\#Pos\ p\# \} + C \implies I \models \{\#Neg\ p\# \} + D \implies I \models C + D$ **and**

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

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

lemma *inference-increasing*:

assumes *inference* $S\ S'$ **and** $\psi \in fst\ S$

shows $\psi \in fst\ S'$

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

lemma *rtranclp-inference-increasing*:

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

shows $\psi \in fst\ S'$

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

lemma *inference-clause-already-used-increasing*:

assumes *inference-clause* $S\ S'$

shows $snd\ S \subseteq snd\ S'$

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

lemma *inference-already-used-increasing*:

assumes *inference* $S\ S'$

shows $snd\ S \subseteq snd\ S'$

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 $I \models_s fst\ N \longleftrightarrow I \models_s fst\ N'$
using *assms* **apply** (*induct rule: rtranclp-induct*)
apply (*simp add: inference-preserves-un-sat*)
using *inference-preserves-un-sat rtranclp-inference-preserves-total* **by** *blast*

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

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

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

lemma *rtranclp-inference-preserves-finite*:
assumes *rtranclp inference* $\psi\ \psi'$ **and** *finite* $(fst\ \psi)$
shows *finite* $(fst\ \psi')$
using *assms* **by** (*induct rule: rtranclp-induct*)
(auto simp add: simplify-preserves-finite inference-preserves-finite)

```

lemma consistent-interp-insert:
  assumes consistent-interp I
  and atm-of P  $\notin$  atm-of ' I
  shows consistent-interp (insert P I)
proof -
  have P: insert P I = I  $\cup$  {P} by auto
  show ?thesis unfolding P
  apply (rule consistent-interp-disjoint)
  using assms by (auto simp add: atms-of-s-def)
qed

lemma simplify-clause-preserves-sat:
  assumes simp: simplify  $\psi$   $\psi'$ 
  and satisfiable  $\psi'$ 
  shows satisfiable  $\psi$ 
  using assms
proof induction
  case (tautology-deletion A P) note AP = this(1) and sat = this(2)
  let ?A' = A + {#Pos P#} + {#Neg P#}
  let ? $\psi'$  =  $\psi$  - {?A'}
  obtain I where
    I: I  $\models$  s ? $\psi'$  and
    cons: consistent-interp I and
    tot: total-over-m I ? $\psi'$ 
  using sat unfolding satisfiable-def by auto
  { assume Pos P  $\in$  I  $\vee$  Neg P  $\in$  I
    then have I  $\models$  ?A' by auto
    then have I  $\models$  s  $\psi$  using I by (metis insert-Diff tautology-deletion.hyps true-clss-insert)
    then have ?case using cons tot by auto
  }
  moreover {
    assume Pos: Pos P  $\notin$  I and Neg: Neg P  $\notin$  I
    then have consistent-interp (I  $\cup$  {Pos P}) using cons by simp
    moreover have I'A: I  $\cup$  {Pos P}  $\models$  ?A' by auto
    have {Pos P}  $\cup$  I  $\models$  s  $\psi$  - {A + {#Pos P#} + {#Neg P#}}
      using {I  $\models$  s  $\psi$  - {A + {#Pos P#} + {#Neg P#}}} true-clss-union-increase' by blast
    then have I  $\cup$  {Pos P}  $\models$  s  $\psi$ 
      by (metis (no-types) Un-empty-right Un-insert-left Un-insert-right I'A insert-Diff
        sup-bot.left-neutral tautology-deletion.hyps true-clss-insert)
    ultimately have ?case using satisfiable-carac' by blast
  }
  ultimately show ?case by blast
next
  case (condensation A L) note AL = this(1) and sat = this(2)
  have f3: simplify  $\psi$  ( $\psi$  - {A + {#L#} + {#L#}}  $\cup$  {A + {#L#}})
    using AL simplify.condensation by blast
  obtain LL :: 'a literal multiset set  $\Rightarrow$  'a literal set where
    f4: LL ( $\psi$  - {A + {#L#} + {#L#}}  $\cup$  {A + {#L#}})  $\models$  s  $\psi$  - {A + {#L#} + {#L#}}  $\cup$  {A
  + {#L#}}
     $\wedge$  consistent-interp (LL ( $\psi$  - {A + {#L#} + {#L#}}  $\cup$  {A + {#L#}}))
     $\wedge$  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: insert (A + {#L#} + {#L#}) ( $\psi$  - {A + {#L#} + {#L#}}) =  $\psi$ 

```

```

    using AL by fastforce
  have atms-of (A + {#L#} + {#L#}) = atms-of ({#L#} + A)
    by simp
  then show ?case
    using f5 f4 f3 by (metis (no-types) add.commute satisfiable-def simplify-preserves-un-sat'
      total-over-m-insert total-over-m-union)
next
case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and sat = this(4)
let ?ψ' = ψ - {B}
obtain I where I: I ⊨s ?ψ' and cons: consistent-interp I and tot: total-over-m I ?ψ'
  using sat unfolding satisfiable-def by auto
have I ⊨ A using A I by (metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def)
then have I ⊨ B using AB subset-mset.less-imp-le true-clss-mono-leD by blast
then have I ⊨s ψ using I by (metis insert-Diff-single true-clss-insert)
then show ?case using cons satisfiable-carac' by blast
qed

```

```

lemma simplify-preserves-unsat:
  assumes inference ψ ψ'
  shows satisfiable (fst ψ') ⟶ satisfiable (fst ψ)
  using assms apply (induct rule: inference.induct)
  using satisfiable-decreasing by (metis fst-conv)+

```

```

lemma inference-preserves-unsat:
  assumes inference** S S'
  shows satisfiable (fst S') ⟶ satisfiable (fst S)
  using assms apply (induct rule: rtranclp-induct)
  apply simp-all
  using simplify-preserves-unsat by blast

```

```

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

```

```

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

```

```

lemma sem-tree-size[case-names bigger]:
  (⋀xs:: 'v sem-tree. (⋀ys:: 'v sem-tree. sem-tree-size ys < sem-tree-size xs ⟹ P ys) ⟹ P xs)
  ⟹ P xs
  by (fact Nat.measure-induct-rule)

```

```

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

```

```

lemma simplify-preserve-partial-leaf:
  simplify N N' ⟹ partial-interps Leaf I N ⟹ partial-interps Leaf I N'
  apply (induct rule: simplify.induct)
  using union-lcomm apply auto[1]
  apply (simp, metis atms-of-plus total-over-set-union true-clss-union)
  apply simp
  by (metis atms-of-ms-singleton mset-le-exists-conv subset-mset-def true-clss-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* $t\ I\ (\text{fst } S)$
shows *partial-interps* $t\ I\ (\text{fst } S')$
using *assms* **apply** (*induct* t *arbitrary*: I , *simp-all*)
by (*meson inference-increasing*)

lemma *rtranclp-inference-preserve-partial-tree*:
assumes *rtranclp inference* $N\ N'$
and *partial-interps* $t\ I\ (\text{fst } N)$
shows *partial-interps* $t\ I\ (\text{fst } N')$
using *assms* **apply** (*induct rule*: *rtranclp-induct*, *auto*)
using *inference-preserve-partial-tree* **by** *force*

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

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

lemma *partial-interps-build-sem-tree-atms-general*:
fixes $\psi :: 'v :: \text{linorder clauses}$ **and** $p :: 'v \text{ literal list}$
assumes *unsat*: *unsatisfiable* ψ **and** *finite* ψ **and** *consistent-interp* I
and *finite* atms
and *atms-of-ms* $\psi = \text{atms} \cup \text{atms-of-s } I$ **and** $\text{atms} \cap \text{atms-of-s } I = \{\}$
shows *partial-interps* (*build-sem-tree* $\text{atms } \psi$) $I\ \psi$
using *assms*
proof (*induct arbitrary*: I *rule*: *build-sem-tree.induct*)
case ($1\ \text{atms } \psi\ I_a$) **note** $IH1 = \text{this}(1)$ **and** $IH2 = \text{this}(2)$ **and** $\text{unsat} = \text{this}(3)$ **and** $\text{finite} = \text{this}(4)$
and $\text{cons} = \text{this}(5)$ **and** $f = \text{this}(6)$ **and** $\text{un} = \text{this}(7)$ **and** $\text{disj} = \text{this}(8)$

```

{
  assume atms: atms = {}
  then have atmsIa: atms-of-ms  $\psi$  = atms-of-s Ia using un by auto
  then have total-over-m Ia  $\psi$  unfolding total-over-m-def atmsIa by auto
  then have  $\chi$ :  $\exists \chi \in \psi. \neg Ia \models \chi$ 
    using unsat cons unfolding true-clss-def satisfiable-def by auto
  then have build-sem-tree atms  $\psi$  = Leaf using atms by auto
  moreover
    have tot:  $\bigwedge \chi. \chi \in \psi \implies \text{total-over-m Ia } \{\chi\}$ 
    unfolding total-over-m-def total-over-set-def atms-of-ms-def atms-of-s-def
    using atmsIa atms-of-ms-def by fastforce
  have partial-interps Leaf Ia  $\psi$ 
    using  $\chi$  tot by (auto simp add: total-over-m-def total-over-set-def atms-of-ms-def)

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

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

  moreover have disj': Set.remove (Min atms) atms  $\cap$  atms-of-s (Ia  $\cup \{Neg (Min atms)\})$  = {}
    using disj by auto
  moreover have finite (Set.remove (Min atms) atms) using f by (simp add: remove-def)
  ultimately have subtree2: partial-interps (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (Ia  $\cup \{Neg (Min atms)\})$   $\psi$ 
    using IH2[of Ia  $\cup \{Neg (Min (atms))\}] atms f unsat finite$  by metis

  then have ?case
    using IH1 subtree1 subtree2 f local.finite unsat atms by simp
}
ultimately show ?case by metis
qed

```

lemma *partial-interps-build-sem-tree-atms*:
fixes $\psi :: 'v :: \text{linorder clauses}$ **and** $p :: 'v \text{ literal list}$
assumes *unsat*: *unsatisfiable* ψ **and** *finite*: *finite* ψ
shows *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) $\{\}$ ψ
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* ψ) **unfolding** *atms-of-ms-def* **using** *finite* **by** *simp*
ultimately **show** *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) $\{\}$ ψ
using *partial-interps-build-sem-tree-atms-general*[*of* $\psi \{\}$ *atms-of-ms* ψ] *assms* **by** *metis*
qed

lemma *can-decrease-count*:
fixes $\psi'' :: 'v \text{ clauses} \times ('v \text{ clause} \times 'v \text{ clause} \times 'v) \text{ set}$
assumes *count* $\chi \ L = n$
and $L \in \# \chi$ **and** $\chi \in \text{fst } \psi$
shows $\exists \psi' \chi'. \text{inference}^{**} \psi \psi' \wedge \chi' \in \text{fst } \psi' \wedge (\forall L. L \in \# \chi \longleftrightarrow L \in \# \chi')$
 $\wedge \text{count } \chi' \ L = 1$
 $\wedge (\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi')$
 $\wedge (I \models \chi \longleftrightarrow I \models \chi')$
 $\wedge (\forall I'. \text{total-over-m } I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi'\})$

using *assms*

proof (*induct n arbitrary: $\chi \psi$*)

case 0

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

next

case (*Suc n χ*)

note *IH* = *this*(1) **and** *count* = *this*(2) **and** *L* = *this*(3) **and** χ = *this*(4)

{

assume $n = 0$

then **have** *inference*^{**} $\psi \psi$

and $\chi \in \text{fst } \psi$

and $\forall L. (L \in \# \chi) \longleftrightarrow (L \in \# \chi)$

and *count* $\chi \ L = (1::\text{nat})$

and $\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi$

by (*auto simp add: count L χ*)

then **have** ?*case* **by** *metis*

}

moreover {

assume $n > 0$

then **have** $\exists C. \chi = C + \{\#L, L\# \}$

by (*metis L One-nat-def add-diff-cancel-right' count-diff count-single diff-Suc-Suc diff-zero*
local.count multi-member-split union-assoc)

then **obtain** *C* **where** $\chi = C + \{\#L, L\# \}$ **by** *metis*

let $? \chi' = C + \{\#L\# \}$

let $? \psi' = (\text{fst } \psi \cup \{? \chi'\}, \text{snd } \psi)$

have $\varphi: \forall \varphi \in \text{fst } \psi. (\varphi \in \text{fst } \psi \vee \varphi \neq ? \chi') \longleftrightarrow \varphi \in \text{fst } ? \psi'$ **unfolding** *C* **by** *auto*

have *inf*: *inference* $\psi ? \psi'$

using *C factoring χ prod.collapse union-commute inference-step* **by** *metis*

moreover **have** *count'*: *count* $? \chi' \ L = n$ **using** *C count* **by** *auto*

moreover **have** $L \chi'$: $L : \# ? \chi'$ **by** *auto*

moreover **have** $\chi' \psi'$: $? \chi' \in \text{fst } ? \psi'$ **by** *auto*

ultimately obtain ψ'' and χ''
where
*inference*** $? \psi' \psi''$ and
 $\alpha: \chi'' \in \text{fst } \psi''$ and
 $\forall La. (La \in \# ? \chi') \longleftrightarrow (La \in \# \chi'')$ and
 $\beta: \text{count } \chi'' L = (1::\text{nat})$ and
 $\varphi': \forall \varphi. \varphi \in \text{fst } ? \psi' \longrightarrow \varphi \in \text{fst } \psi''$ and
 $I\chi: I \models ? \chi' \longleftrightarrow I \models \chi''$ and
 $\text{tot}: \forall I'. \text{total-over-m } I' \{? \chi'\} \longrightarrow \text{total-over-m } I' \{\chi''\}$
using *IH*[*of* $? \chi' ? \psi'$ *count'* $L\chi' \chi' \psi'$ *by* *blast*

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

}

ultimately show $? \text{case}$ *by* *auto*

qed

lemma *can-decrease-tree-size*:
fixes $\psi :: 'v \text{ state}$ and *tree* $:: 'v \text{ sem-tree}$
assumes *finite* (*fst* ψ) and *already-used-inv* ψ
and *partial-interps tree I* (*fst* ψ)
shows $\exists (\text{tree}' :: 'v \text{ sem-tree}) \psi'. \text{inference** } \psi \psi' \wedge \text{partial-interps tree}' I (\text{fst } \psi')$
 $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0)$
using *assms*

proof (*induct arbitrary: I rule: sem-tree-size*)
case (*bigger xs I*) *note* $\text{IH} = \text{this}(1)$ and *finite* = *this*(2) and *a-u-i* = *this*(3) and *part* = *this*(4)

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

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

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

```

{
  assume Negχ: ¬Neg v ∈# χ
  have ¬ I ⊨ χ using χ Posv unfolding true-cls-def true-lit-def by auto
  moreover have total-over-m I {χ}
    using Posv Negχ atm-imp-pos-or-neg-lit totχ unfolding total-over-m-def total-over-set-def
    by fastforce
  ultimately have partial-interps Leaf I (fst ψ)
  and sem-tree-size Leaf < sem-tree-size xs
  and inference** ψ ψ
    unfolding xs by (auto simp add: χψ)
}
moreover {
  assume Posχ: ¬Pos v ∈# χ'
  then have Iχ: ¬ I ⊨ χ' using χ' Posv unfolding true-cls-def true-lit-def by auto
  moreover have total-over-m I {χ'}
    using Negv Posχ atm-imp-pos-or-neg-lit totχ' unfolding total-over-m-def total-over-set-def by fastforce
  ultimately have partial-interps Leaf I (fst ψ) and
    sem-tree-size Leaf < sem-tree-size xs and
    inference** ψ ψ
    using χ'ψ Iχ unfolding xs by auto
}
moreover {
  assume neg: Neg v ∈# χ and pos: Pos v ∈# χ'
  then obtain ψ' χ2 where inf: rtrnclp inference ψ ψ' and χ2incl: χ2 ∈ fst ψ'
    and χχ2incl: ∀ L. L :# χ ↔ L :# χ2
    and countχ2: count χ2 (Neg v) = 1
    and φ: ∀ φ::'v literal multiset. φ ∈ fst ψ → φ ∈ fst ψ'
    and Iχ: I ⊨ χ ↔ I ⊨ χ2
    and tot-impχ: ∀ I'. total-over-m I' {χ} → total-over-m I' {χ2}
    using can-decrease-count[of χ Neg v count χ (Neg v) ψ I] χψ χ'ψ by auto

  have χ' ∈ fst ψ' by (simp add: χ'ψ φ)
  with pos
  obtain ψ'' χ2' where
    inf': inference** ψ' ψ''
    and χ2'-incl: χ2' ∈ fst ψ''
    and χ'χ2'-incl: ∀ L::'v literal. (L ∈# χ') = (L ∈# χ2')
    and countχ2': count χ2' (Pos v) = (1::nat)
    and φ': ∀ φ::'v literal multiset. φ ∈ fst ψ' → φ ∈ fst ψ''
    and Iχ': I ⊨ χ' ↔ I ⊨ χ2'
    and tot-impχ': ∀ I'. total-over-m I' {χ'} → total-over-m I' {χ2'}
    using can-decrease-count[of χ' Pos v count χ' (Pos v) ψ' I] by auto

  obtain C where χ2: χ2 = C + {#Neg v#} and negC: Neg v ∉# C and posC: Pos v ∉# C
    by (metis (no-types, lifting) One-nat-def Posv Suc-inject Suc-pred χχ2incl countχ2
      count-diff count-single grOI insert-DiffM insert-DiffM2 multi-member-skip
      old.nat.distinct(2))

  obtain C' where
    χ2': χ2' = C' + {#Pos v#} and
    posC': Pos v ∉# C' and
    negC': Neg v ∉# C'
  proof -
    assume a1: ∧ C'. [χ2' = C' + {#Pos v#}; Pos v ∉# C'; Neg v ∉# C'] ⇒ thesis

```

```

have f2:  $\bigwedge n. (n::nat) - n = 0$ 
  by simp
have Neg v  $\notin \# \chi^{2'} - \{\#Pos v\# \}$ 
  using Negv  $\chi' \chi^{2'} \text{-incl}$  by auto
then show ?thesis
  using f2 a1 by (metis add.commute count $\chi^{2'}$  count-diff count-single insert-DiffM
    less-nat-zero-code zero-less-one)
qed

have already-used-inv  $\psi'$ 
  using 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 totC: total-over-m I {C}
  using tot-imp $\chi$  tot $\chi$  total-over-m-remove[of I Pos v C] negC posC unfolding  $\chi^2$ 
  by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
have totC': total-over-m I {C'}
  using tot-imp $\chi'$  tot $\chi'$  total-over-m-sum total-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 Bex-mset-def
  by (metis add-gr-0 count-union true-cls-singleton true-cls-union-increase)
then have part-I- $\psi'''$ : partial-interps Leaf I (fst  $\psi'' \cup \{C + C'\}$ )
  using 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  $\text{Pos } p \in \# \ C' \wedge \text{Neg } p \in \# \ C$  using  $p \ n$  by force
  then have ?thesis by (metis add-gr-0 count-union tautology-Pos-Neg)
}
moreover {
  assume  $p = v$ 
  then have ?thesis using decomp by (metis add.commute add-diff-cancel-left')
}
ultimately show ?thesis by auto
qed
moreover {
  assume  $\exists \chi \in \text{fst } \psi'' . (\forall I . \text{total-over-m } I \ \{C+C'\} \longrightarrow \text{total-over-m } I \ \{\chi\})$ 
     $\wedge (\forall I . \text{total-over-m } I \ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ 
  then obtain  $\vartheta$  where  $\vartheta: \vartheta \in \text{fst } \psi''$  and
     $\text{tot-}\vartheta\text{-}CC': \forall I . \text{total-over-m } I \ \{C+C'\} \longrightarrow \text{total-over-m } I \ \{\vartheta\}$  and
     $\vartheta\text{-inv}: \forall I . \text{total-over-m } I \ \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$  by blast
  have partial-interps Leaf  $I$  (fst  $\psi''$ )
    using  $\text{tot-}\vartheta\text{-}CC' \ \vartheta \ \vartheta\text{-inv} \ \text{tot}C \ \text{tot}C' \ \langle \neg I \models C + C' \rangle \text{total-over-m-sum}$  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  $\text{tot}C \ \text{tot}C' \ \text{total-over-m-sum}$  by auto
  then have  $\neg \text{tautology} \ (C' + C)$ 
    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  $ag \ (I \cup \{\text{Pos } v\})$  (fst  $\psi$ )
    and partad: partial-interps  $ad \ (I \cup \{\text{Neg } v\})$  (fst  $\psi$ )
    using part partial-interps.simps(2) unfolding  $xs$  by metis+
  moreover have sem-tree-size  $ag <$  sem-tree-size  $xs \longrightarrow \text{finite} \ (\text{fst } \psi) \longrightarrow \text{already-used-inv } \psi$ 
     $\longrightarrow (\text{partial-interps } ag \ (I \cup \{\text{Pos } v\}) \ (\text{fst } \psi) \longrightarrow$ 
     $(\exists \text{tree}' \ \psi' . \text{inference}^{**} \ \psi \ \psi' \wedge \text{partial-interps } \text{tree}' \ (I \cup \{\text{Pos } v\}) \ (\text{fst } \psi'))$ 
     $\wedge (\text{sem-tree-size } \text{tree}' < \text{sem-tree-size } ag \vee \text{sem-tree-size } ag = 0)))$ 
    using IH by auto
  ultimately obtain  $\psi' :: 'v \text{ state}$  and  $\text{tree}' :: 'v \text{ sem-tree}$  where
    inf: inference**  $\psi \ \psi'$ 
    and part: partial-interps  $\text{tree}' \ (I \cup \{\text{Pos } v\})$  (fst  $\psi'$ )
    and size: sem-tree-size  $\text{tree}' <$  sem-tree-size  $ag \vee \text{sem-tree-size } ag = 0$ 
    using finite part rtranclp.rtrancl-refl a-u-i by blast

  have partial-interps  $ad \ (I \cup \{\text{Neg } v\})$  (fst  $\psi'$ )
    using rtranclp-inference-preserve-partial-tree inf partad by metis
  then have partial-interps  $(\text{Node } v \ \text{tree}' \ ad) \ I$  (fst  $\psi'$ ) using part by auto
}

```

```

    then have ?case using inf size size-ag part unfolding xs by fastforce
  }
  moreover {
    assume size-ad: sem-tree-size ad > 0
    have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
    moreover have partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
      partial-interps ad (I ∪ {Neg v}) (fst ψ)
      using part partial-interps.simps(2) unfolding xs by metis+
    moreover have sem-tree-size ad < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
      ⟶ ( partial-interps ad (I ∪ {Neg v}) (fst ψ)
        ⟶ (∃ tree' ψ'. inference** ψ ψ' ∧ partial-interps tree' (I ∪ {Neg v}) (fst ψ')
          ∧ (sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0)))
      using IH by auto
    ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
      inf: inference** ψ ψ'
      and part: partial-interps tree' (I ∪ {Neg v}) (fst ψ')
      and size: sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0
      using finite part rtranclp.rtrancl-refl a-u-i by blast

    have partial-interps ag (I ∪ {Pos v}) (fst ψ')
      using rtranclp-inference-preserve-partial-tree inf partag by metis
    then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
    then have ?case using inf size size-ad unfolding xs by fastforce
  }
  ultimately have ?case by auto
}
ultimately show ?case by auto
qed

```

lemma *inference-completeness-inv*:

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

assumes

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

finite: *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** *totχ: total-over-m {} {χ}* **and** $\chi\psi: \chi \in \text{fst } \psi$

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

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

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

moreover have *inference** ψ ψ* **by** *auto*

ultimately have *?case* **by** *metis*

}

moreover {

fix *v tree1 tree2*

```

assume tree: tree = Node v tree1 tree2
obtain
  tree'  $\psi'$  where inf: inference**  $\psi \psi'$  and
  part': partial-interps tree' {} (fst  $\psi'$ ) and
  decrease: sem-tree-size tree' < sem-tree-size tree  $\vee$  sem-tree-size tree = 0
  using can-decrease-tree-size[of  $\psi$ ] H(2,4,5) unfolding tautology-def by meson
have sem-tree-size tree' < sem-tree-size tree using decrease unfolding tree by auto
moreover have finite (fst  $\psi'$ ) using rtranclp-inference-preserves-finite inf H(4) by metis
moreover have unsatisfiable (fst  $\psi'$ )
  using inference-preserves-unsat inf bigger.premis(2) by blast
moreover have already-used-inv  $\psi'$ 
  using H(5) inf rtranclp-inference-preserves-already-used-inv[of  $\psi \psi'$ ] by auto
ultimately have ?case using inf rtranclp-trans part' H(1) by fastforce
}
ultimately show ?case by (case-tac tree, auto)
qed
qed

```

```

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

```

```

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

```

```

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

```

12.4 Lemma about the simplified state

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

```

lemma simplified-count:
  assumes simp: simplified  $\psi$  and  $\chi: \chi \in \psi$ 
  shows count  $\chi$  L  $\leq 1$ 
proof –
  {
    let ? $\chi' = \chi - \{\#L, L\# \}$ 
    assume count  $\chi$  L  $\geq 2$ 
    then have f1: count  $(\chi - \{\#L, L\# \} + \{\#L, L\# \})$  L = count  $\chi$  L
    by simp
  }

```

```

then have  $L \in \# \chi - \{\#L\# \}$ 
  by simp
then have  $\chi': ?\chi' + \{\#L\# \} + \{\#L\# \} = \chi$ 
  using f1 by (metis (no-types) diff-diff-add diff-single-eq-union union-assoc
    union-single-eq-member)
have  $\exists \psi'. \text{simplify } \psi \ \psi'$ 
  by (metis (no-types, hide-lams)  $\chi \ \chi'$  add.commute factoring-imp-simplify union-assoc)
then have False using simp by auto
}
then show ?thesis by arith
qed

lemma simplified-no-both:
  assumes simp: simplified  $\psi$  and  $\chi: \chi \in \psi$ 
  shows  $\neg (L \in \# \chi \wedge \neg L \in \# \chi)$ 
proof (rule ccontr)
  assume  $\neg \neg (L \in \# \chi \wedge \neg L \in \# \chi)$ 
  then have  $L \in \# \chi \wedge \neg L \in \# \chi$  by metis
  then obtain  $\chi'$  where  $\chi = \chi' + \{\#Pos \text{ (atm-of } L)\# \} + \{\#Neg \text{ (atm-of } L)\# \}$ 
    by (metis Neg-atm-of-iff Pos-atm-of-iff diff-union-swap insert-DiffM2 uminus-Neg uminus-Pos)
  then show False using  $\chi$  simp tautology-deletion by fastforce
qed

lemma simplified-not-tautology:
  assumes simplified  $\{\psi\}$ 
  shows  $\sim \text{tautology } \psi$ 
proof (rule ccontr)
  assume  $\sim ?thesis$ 
  then obtain  $p$  where  $Pos \ p \in \# \psi \wedge Neg \ p \in \# \psi$  using tautology-decomp by metis
  then obtain  $\chi$  where  $\psi = \chi + \{\#Pos \ p\# \} + \{\#Neg \ p\# \}$ 
    by (metis insert-noteq-member literal.distinct(1) multi-member-split)
  then have  $\sim \text{simplified } \{\psi\}$  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:  $\neg \text{simplified } \{\psi - \{\#l\# \}\}$ 
  {
    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  $l': \text{simplify } \{\psi - \{\#l\# \}\} \ l'$  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
    qed
  }

```

```

      by (metis simplify.tautology-deletion[of A+{#l#} P {ψ}] add commute)
next
  case (condensation A L)
  have A + {#L#} + {#L#} + {#l#} ∈ {ψ}
    using A condensation.hyps by blast
  then have {#L, L#} + (A + {#l#}) ∈ {ψ}
    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 ψ and incl: ψ' ⊆ ψ
  shows simplified ψ'
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain ψ'' where simplify ψ' ψ'' by metis
  then have ∃ l'. simplify ψ l'
  proof (induction rule: simplify.induct)
    case (tautology-deletion A P)
    then show ?thesis using simplify.tautology-deletion[of A P ψ] incl by blast
  next
    case (condensation A L)
    then show ?case using simplify.condensation[of A L ψ] incl by blast
  next
    case (subsumption A B)
    then show ?case using simplify.subsumption[of A ψ B] incl by auto
  qed
  then show False using assms(1) by blast
qed

```

lemma *simplified-in*:

```

  assumes simplified ψ
  and N ∈ ψ
  shows simplified {N}
  using assms by (metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono)

```

lemma *subsumes-imp-formula*:

```

  assumes ψ ≤# φ
  shows {ψ} ⊨p φ
  unfolding true-clss-clf-def apply auto
  using assms true-clf-mono-leD by blast

```

lemma *simplified-imp-distinct-mset-tauto*:

```

  assumes simp: simplified ψ'
  shows distinct-mset-set ψ' and ∀ χ ∈ ψ'. ¬tautology χ
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  $\text{count } \chi \ L \geq 2$ 
      unfolding distinct-mset-def by (metis gr-implies-not0 le-antisym less-one not-le simp
        simplified-count)
    then show False by (metis Suc-1  $\langle \chi \in \psi' \rangle$  not-less-eq-eq simp simplified-count)
  qed
qed

```

```

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

```

12.5 Resolution and Invariants

```

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

```

12.5.1 Invariants

```

lemma resolution-finite:
  assumes resolution  $\psi \ \psi'$  and finite (fst  $\psi$ )
  shows finite (fst  $\psi'$ )
  using assms by (induct rule: resolution.induct)
    (auto simp add: full1-def full-def rtranclp-simplify-preserves-finite
      dest: tranclp-into-rtranclp inference-preserves-finite)

```

```

lemma rtranclp-resolution-finite:
  assumes resolution**  $\psi \ \psi'$  and finite (fst  $\psi$ )
  shows finite (fst  $\psi'$ )
  using assms by (induct rule: rtranclp-induct, auto simp add: resolution-finite)

```

```

lemma resolution-finite-snd:
  assumes resolution  $\psi \ \psi'$  and finite (snd  $\psi$ )
  shows finite (snd  $\psi'$ )
  using assms apply (induct rule: resolution.induct, auto simp add: inference-preserves-finite-snd)
  using inference-preserves-finite-snd snd-conv by metis

```

```

lemma rtranclp-resolution-finite-snd:
  assumes resolution**  $\psi \ \psi'$  and finite (snd  $\psi$ )
  shows finite (snd  $\psi'$ )
  using assms by (induct rule: rtranclp-induct, auto simp add: resolution-finite-snd)

```

```

lemma resolution-always-simplified:
  assumes resolution  $\psi \ \psi'$ 
  shows simplified (fst  $\psi'$ )
  using assms by (induct rule: resolution.induct)
    (auto simp add: full1-def full-def)

```

lemma *tranclp-resolution-always-simplified*:

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

shows *simplified (fst 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 $\psi \psi'$ and finite: finite (fst ψ)*

shows *fst $\psi' \subseteq$ build-all-simple-clss (atms-of-ms (fst ψ))*

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 *fst $\psi' \subseteq$ build-all-simple-clss (atms-of-ms (fst ψ'))*

using *simplified-in-build-all finite' simplified-imp-distinct-mset-tauto[of fst ψ'] by auto*

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

using *res finite resolution-atms-of[of $\psi \psi'$] by auto*

ultimately show *?thesis by (meson atms-of-ms-finite local.finite order.trans rev-finite-subset build-all-simple-clss-mono)*

qed

lemma *rtranclp-resolution-include*:

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

shows *fst $\psi' \subseteq$ build-all-simple-clss (atms-of-ms (fst ψ))*

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

apply (*simp add: resolution-include*)

by (*meson atms-of-ms-finite build-all-simple-clss-finite build-all-simple-clss-mono finite-subset resolution-include rtranclp-resolution-atms-of set-rev-mp subsetI tranclp-into-rtranclp*)

abbreviation *already-used-all-simple*

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

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

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

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

assumes *vars \subseteq vars'*

shows *already-used-all-simple a vars \implies already-used-all-simple a vars'*

using *assms by fast*

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

assumes *inference-clause $S S'$*

and *already-used-all-simple (snd S) vars*

```

and simplified (fst S)
and atms-of-ms (fst S)  $\subseteq$  vars
shows already-used-all-simple (snd (fst S  $\cup$  {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)  $\in$  snd (fst (N, already-used)
 $\cup$  {fst (C + D, already-used  $\cup$  {({#Pos P#} + C, {#Neg P#} + D))},
snd (C + D, already-used  $\cup$  {({#Pos P#} + C, {#Neg P#} + D))})
then have (A, B)  $\in$  already-used  $\vee$  (A, B) = ({#Pos P#} + C, {#Neg P#} + D) by auto
moreover {
assume (A, B)  $\in$  already-used
then have simplified {A}  $\wedge$  simplified {B}  $\wedge$  atms-of A  $\subseteq$  vars  $\wedge$  atms-of B  $\subseteq$  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) 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 rtrancplp-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: rtrancplp-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 rtrancplp-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 trancplpD)
  by (metis inference-simplified-already-used-subset fst-conv snd-conv)

lemma trancplp-resolution-simplified-already-used-subset:

```

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

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

lemma *already-used-all-simple-in-already-used-top:*

assumes *already-used-all-simple s vars and finite vars*
shows *s \subseteq already-used-top vars*

proof

fix *x*
assume *x-s: x \in s*
obtain *A B where x: x = (A, B) by (case-tac x, auto)*
then have *simplified {A} and atms-of A \subseteq vars using assms(1) x-s by fastforce+*
then have *A: A \in build-all-simple-clss vars*
 using *build-all-simple-clss-mono[of vars atms-of A] x assms(2)*
 simplified-imp-distinct-mset-tauto[of {A}]
 distinct-mset-not-tautology-implies-in-build-all-simple-clss by fast
moreover have *simplified {B} and atms-of B \subseteq vars using assms(1) x-s x by fast+*
then have *B: B \in build-all-simple-clss vars*
 using *simplified-imp-distinct-mset-tauto[of {B}]*
 distinct-mset-not-tautology-implies-in-build-all-simple-clss
 build-all-simple-clss-mono[of vars atms-of B] x assms(2) by fast
ultimately show *x \in build-all-simple-clss vars \times build-all-simple-clss vars*
 unfolding *x by auto*

qed

lemma *already-used-top-finite:*

assumes *finite vars*
shows *finite (already-used-top vars)*
using *build-all-simple-clss-finite assms by auto*

lemma *already-used-top-increasing:*

assumes *var \subseteq var' and finite var'*
shows *already-used-top var \subseteq already-used-top var'*
using *assms build-all-simple-clss-mono by auto*

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

fixes *s :: ('a::linorder literal multiset \times 'a literal multiset) set and vars :: 'a 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 $\psi \equiv$ card (already-used-top vars $- \psi$)*

lemma *resolution-card-simple-decreasing:*

assumes *res: resolution $\psi \psi'$*
and *a-u-s: already-used-all-simple (snd ψ) vars*
and *finite-v: finite vars*
and *finite-fst: finite (fst ψ)*

and *finite-snd*: *finite* (*snd* ψ)
and *simp*: *simplified* (*fst* ψ)
and *atms-of-ms* (*fst* ψ) \subseteq *vars*
shows *card-simple vars* (*snd* ψ') < *card-simple vars* (*snd* ψ)
proof –
let *?vars* = *vars*
let *?top* = *build-all-simple-clss* *?vars* \times *build-all-simple-clss* *?vars*
have 1: *card-simple vars* (*snd* ψ) = *card* *?top* – *card* (*snd* ψ)
using *card-Diff-subset* *finite-snd* *already-used-all-simple-in-already-used-top*[*OF* *a-u-s*]
finite-v **by** *metis*
have *a-u-s'*: *already-used-all-simple* (*snd* ψ') *vars*
using *already-used-all-simple-inv* *res* *a-u-s* *assms*(7) **by** *blast*
have *f*: *finite* (*snd* ψ') **using** *already-used-all-simple-finite* *a-u-s'* *finite-v* **by** *auto*
have 2: *card-simple vars* (*snd* ψ') = *card* *?top* – *card* (*snd* ψ')
using *card-Diff-subset*[*OF* *f*] *already-used-all-simple-in-already-used-top*[*OF* *a-u-s'* *finite-v*]
by *auto*
have *card* (*already-used-top vars*) \geq *card* (*snd* ψ')
using *already-used-all-simple-in-already-used-top*[*OF* *a-u-s'* *finite-v*]
card-mono[*of* *already-used-top vars* *snd* ψ'] *already-used-top-finite*[*OF* *finite-v*] **by** *metis*
then show *?thesis*
using *psubset-card-mono*[*OF* *f* *resolution-simplified-already-used-subset*[*OF* *res simp*]]
unfolding 1 2 **by** *linarith*
qed

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

assumes *tranclp resolution* ψ ψ' **and** *finite-fst*: *finite* (*fst* ψ)
and *already-used-all-simple* (*snd* ψ) *vars*
and *atms-of-ms* (*fst* ψ) \subseteq *vars*
and *finite-v*: *finite* *vars*
and *finite-snd*: *finite* (*snd* ψ)
and *simplified* (*fst* ψ)
shows *card-simple vars* (*snd* ψ') < *card-simple vars* (*snd* ψ)
using *assms*
proof (*induct rule*: *tranclp.induct*)
case (*r-into-trancl* ψ ψ')
then show *?case* **by** (*simp add*: *resolution-card-simple-decreasing*)
next
case (*trancl-into-trancl* ψ ψ' ψ'') **note** *res* = *this*(1) **and** *res'* = *this*(3) **and** *a-u-s* = *this*(5) **and**
atms = *this*(6) **and** *f-v* = *this*(7) **and** *f-fst* = *this*(4) **and** *H* = *this*
then have *card-simple vars* (*snd* ψ') < *card-simple vars* (*snd* ψ) **by** *auto*
moreover have *a-u-s'*: *already-used-all-simple* (*snd* ψ') *vars*
using *rtranclp-already-used-all-simple-inv*[*OF* *tranclp-into-rtranclp*[*OF* *res*] *a-u-s* *atms* *f-fst*] .
have *finite* (*fst* ψ')
by (*meson* *build-all-simple-clss-finite* *rev-finite-subset* *rtranclp-resolution-include*
trancl-into-trancl.hyps(1) *trancl-into-trancl.prem*s(1))
moreover have *finite* (*snd* ψ') **using** *already-used-all-simple-finite*[*OF* *a-u-s'* *f-v*] .
moreover have *simplified* (*fst* ψ') **using** *res* *tranclp-resolution-always-simplified* **by** *blast*
moreover have *atms-of-ms* (*fst* ψ') \subseteq *vars*
by (*meson* *atms* *f-fst* *order.trans* *res* *rtranclp-resolution-atms-of* *tranclp-into-rtranclp*)
ultimately show *?case*
using *resolution-card-simple-decreasing*[*OF* *res'* *a-u-s'* *f-v*] *f-v*
less-trans[*of* *card-simple vars* (*snd* ψ'') *card-simple vars* (*snd* ψ')
card-simple vars (*snd* ψ)]
by *blast*

qed

lemma *tranclp-resolution-card-simple-decreasing-2*:

assumes *tranclp resolution* $\psi \ \psi'$
and *finite-fst*: *finite* (*fst* ψ)
and *empty-snd*: *snd* $\psi = \{\}$
and *simplified* (*fst* ψ)
shows *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ') < *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ)

proof –

let *?vars* = (*atms-of-ms* (*fst* ψ))
have *already-used-all-simple* (*snd* ψ) *?vars* **unfolding** *empty-snd* **by** *auto*
moreover **have** *atms-of-ms* (*fst* ψ) \subseteq *?vars* **by** *auto*
moreover **have** *finite-v*: *finite* *?vars* **using** *finite-fst* **by** *auto*
moreover **have** *finite-snd*: *finite* (*snd* ψ) **unfolding** *empty-snd* **by** *auto*
ultimately **show** *?thesis*
using *assms*(1,2,4) *tranclp-resolution-card-simple-decreasing*[*of* $\psi \ \psi'$] **by** *presburger*

qed

12.5.2 well-foundness if the relation

lemma *wf-simplified-resolution*:

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

proof –

{
fix *a b* :: '*v::linorder state*
assume $(b, a) \in \{(y, x). (\text{atms-of-ms } (\text{fst } x) \subseteq \text{vars} \wedge \text{simplified } (\text{fst } x) \wedge \text{finite } (\text{snd } x) \wedge \text{finite } (\text{fst } x) \wedge \text{already-used-all-simple } (\text{snd } x) \text{ vars}) \wedge \text{resolution } x \ y\}$
then have
atms-of-ms (*fst* *a*) \subseteq *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* \subseteq *already-used-top vars* **by** *auto*
moreover **have** *snd b* \subseteq *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*) \subseteq *vars*) *f-vars*
by *presburger*
moreover **have** *snd a* \subset *snd b* **using** *resolution-simplified-already-used-subset*[*OF res simp*] .
ultimately **have** *finite* (*already-used-top vars*) \wedge *already-used-top vars* \subseteq *already-used-top vars*
 \wedge *snd b* \subseteq *already-used-top vars* \wedge *snd a* \subset *snd b* **by** *metis*
}

then show *?thesis* **using** *wf-bounded-set*[*of* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (\text{fst } x) \subseteq \text{vars} \wedge \text{simplified } (\text{fst } x) \wedge \text{finite } (\text{snd } x) \wedge \text{finite } (\text{fst } x) \wedge \text{already-used-all-simple } (\text{snd } x) \text{ vars}) \wedge \text{resolution } x \ y\}$ $\lambda\cdot$. *already-used-top vars* *snd*]] **by** *auto*

qed

lemma *wf-simplified-resolution'*:

assumes *f-vars*: *finite vars*
shows *wf* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (\text{fst } x) \subseteq \text{vars} \wedge \neg \text{simplified } (\text{fst } x)$

```

     $\wedge$  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  $\cup$  ?S)
    using wf-simplified-resolution[OF f-vars] wf-simplified-resolution'[OF f-vars] wf-Un[of ?R ?S]
    by fast
qed

lemma rtrancp-simplify-already-used-inv:
  assumes simplify** S S'
  and already-used-inv (S, N)
  shows already-used-inv (S', N)
  using assms apply induction
  using simplify-preserves-already-used-inv by fast+

lemma full1-simplify-already-used-inv:
  assumes full1 simplify S S'
  and already-used-inv (S, N)
  shows already-used-inv (S', N)
  using assms trancp-into-rtrancp[of simplify S S'] rtrancp-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 rtrancp-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 rtrancp-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*^{**} ψ ψ'
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
using *assms* **apply** *induction*
using *simplify-clause-preserves-sat* **by** *blast+*

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* ψ ψ'
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
using *assms* *rtanclp-simplify-preserves-unsat*[*of* ψ ψ'] *tranclp-into-rtranclp*
unfolding *full1-def* **by** *metis*

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* ψ ψ'
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
using *assms* *rtanclp-simplify-preserves-unsat*[*of* ψ ψ'] **unfolding** *full-def* **by** *metis*

lemma *resolution-preserves-unsat*:
assumes *resolution* ψ ψ'
shows *satisfiable* (*fst* ψ') \longrightarrow *satisfiable* (*fst* ψ)
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*^{**} ψ ψ'
shows *satisfiable* (*fst* ψ') \longrightarrow *satisfiable* (*fst* ψ)
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 rtrancp-resolution-preserve-partial-tree:
  assumes resolution** S S'
  and partial-interps t I (fst S)
  shows partial-interps t I (fst S')
  using assms apply induction
  using resolution-preserve-partial-tree by fast+
  thm nat-less-induct nat.induct

```

```

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

```

```

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

```

```

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

```

```

translations
   $\{\# e. x. \text{setof } M \#\} == \text{CONST set-mset (CONST image-mset (\%x. e) M)}$ 

```

value $\{\# a. a : \text{setof } \{\#1,1,2::\text{int}\#\}\} = \{1,2\}$

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

interpretation *sum-count-ge-2*:

folding ($\lambda\varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L\#\}) 0$)

rewrites

folding.F ($\lambda\varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L\#\}) 0$) = *sum-count-ge-2*

proof –

show *folding* ($\lambda\varphi. \text{op} + (\text{msetsum } (\text{image-mset } (\text{count } \varphi) \{\# L : \# \varphi. 2 \leq \text{count } \varphi L\#\}))$)
by *standard auto*

then interpret *sum-count-ge-2*:

folding ($\lambda\varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L\#\}) 0$) .

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

qed

lemma *finite-incl-le-setsum*:

finite (*B*::'a multiset set) $\Longrightarrow A \subseteq B \Longrightarrow \Xi A \leq \Xi B$

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

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

then show ?*thesis*

by (*metis* (*no-types*) *Nat.le-iff-add* *aA* *aF* *add.assoc* *finite.insertI* *finite-subset*
insert.prem *local.finite* *sum-count-ge-2.insert* *sum-count-ge-2.remove*)

qed

qed

qed

lemma *mset-condensation1*:

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

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

by (*auto intro: multiset-eqI*)

lemma *mset-condensation2*:

$\{\# La : \# A + \{\# L\# \} + \{\# L\# \}. 2 \leq \text{count } (A + \{\# L\# \} + \{\# L\# \}) La\# \} = \{\# La : \# A. La \neq L \wedge 2 \leq \text{count } A La\# \} \# \cup (\text{replicate-mset } (\text{count } A L + 2) L)$
by (auto intro: multiset-eqI)

lemma *msetsum-disjoint*:

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

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

by (metis assms diff-zero empty-sup image-mset-union msetsum.union multiset-inter-commute multiset-union-diff-commute sup-subset-mset-def zero-diff)

lemma *msetsum-linear[simp]*:

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

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

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

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

by (induction A) auto

lemma *filter-equality-in-mset*:

$\text{filter-mset } (op = L) A = \text{replicate-mset } (\text{count } A L) L$

by (auto simp: multiset-eq-iff)

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

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

by (auto simp: multiset-eq-iff)

lemma *simplify-finite-measure-decrease*:

$\text{simplify } N N' \implies \text{finite } N \implies \text{card } N' + \Xi N' < \text{card } N + \Xi N$

proof (induction rule: simplify.induct)

case (tautology-deletion A P) **note** $an = \text{this}(1)$ **and** $fin = \text{this}(2)$

let $?N' = N - \{A + \{\# \text{Pos } P\# \} + \{\# \text{Neg } P\# \}\}$

have $\text{card } ?N' < \text{card } N$

by (meson card-Diff1-less tautology-deletion.hyps tautology-deletion.prem)

moreover **have** $?N' \subseteq N$ **by** auto

then **have** $\text{sum-count-ge-2 } ?N' \leq \text{sum-count-ge-2 } N$ **using** *finite-incl-le-setsum[OF fin]* **by** blast

ultimately **show** $?case$ **by** linarith

next

case (condensation A L) **note** $AN = \text{this}(1)$ **and** $fin = \text{this}(2)$

let $?C' = A + \{\# L\# \}$

let $?C = A + \{\# L\# \} + \{\# L\# \}$

let $?N' = N - \{?C\} \cup \{?C'\}$

have $\text{card } ?N' \leq \text{card } N$

using AN **by** (metis (no-types, lifting) Diff-subset Un-empty-right Un-insert-right card.remove card-insert-if card-mono fin finite-Diff order-refl)

moreover **have** $\Xi \{?C'\} < \Xi \{?C\}$

proof –

have *mset-decomp*:

$\{\# La \in \# A. (L = La \longrightarrow \text{Suc } 0 \leq \text{count } A La) \wedge (L \neq La \longrightarrow 2 \leq \text{count } A La)\# \}$

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

$\{\# La \in \# A. L = La \wedge \text{Suc } 0 \leq \text{count } A L\# \}$

```

    by (auto simp: multiset-eq-iff ac-simps)
  have mset-decomp2:  $\{\# \text{ La} \in \# A. L \neq \text{La} \longrightarrow 2 \leq \text{count } A \text{ La}\# \} =$ 
     $\{\# \text{ La} \in \# A. L \neq \text{La} \wedge 2 \leq \text{count } A \text{ La}\# \} + \text{replicate-mset } (\text{count } A L) L$ 
    by (auto simp: multiset-eq-iff)
  show ?thesis
    by (auto simp: mset-decomp mset-decomp2 filter-equality-in-mset ac-simps)
qed
have  $\exists N' < \exists N$ 
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:
       $\{\# \text{ La} \in \# A. (L = \text{La} \longrightarrow \text{Suc } 0 \leq \text{count } A \text{ La}) \wedge (L \neq \text{La} \longrightarrow 2 \leq \text{count } A \text{ La})\# \}$ 
       $= \{\# \text{ La} \in \# A. L \neq \text{La} \wedge 2 \leq \text{count } A \text{ La}\# \} +$ 
       $\{\# \text{ La} \in \# A. L = \text{La} \wedge \text{Suc } 0 \leq \text{count } A L\# \}$ 
      by (auto simp: multiset-eq-iff ac-simps)
    have mset-decomp2:  $\{\# \text{ La} \in \# A. L \neq \text{La} \longrightarrow 2 \leq \text{count } A \text{ La}\# \} =$ 
       $\{\# \text{ La} \in \# A. L \neq \text{La} \wedge 2 \leq \text{count } A \text{ La}\# \} + \text{replicate-mset } (\text{count } A L) L$ 
      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-equality-in-mset)
    qed
  ultimately show ?case by linarith
next
case (subsumption A B) note AN = this(1) and AB = this(2) and BN = this(3) and fin = this(4)
have  $\text{card } (N - \{B\}) < \text{card } N$  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*: $\text{wf } \{(N', N). \text{simplify } N \ N' \wedge \text{finite } N\}$

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 rtranclp-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 ψ) and already-used-inv ψ

and partial-interps tree I (fst ψ)

and simplified (fst ψ)

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 $IH = \text{this}(1)$ 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 $xs: xs = \text{Node } v \ ag \ ad$ using sn0 by (case-tac xs , auto)

{

assume sem-tree-size $ag = 0 \wedge \text{sem-tree-size } ad = 0$

then have $ag: ag = \text{Leaf}$ and $ad: ad = \text{Leaf}$ by (case-tac ag , auto, case-tac ad , auto)

then obtain $\chi \ \chi'$ where

$\chi: \neg I \cup \{\text{Pos } v\} \models \chi$ and

$\text{tot}\chi: \text{total-over-m } (I \cup \{\text{Pos } v\}) \ \{\chi\}$ and

$\chi\psi: \chi \in \text{fst } \psi$ and

$\chi': \neg I \cup \{\text{Neg } v\} \models \chi'$ and

$\text{tot}\chi': \text{total-over-m } (I \cup \{\text{Neg } v\}) \ \{\chi'\}$ and $\chi'\psi: \chi' \in \text{fst } \psi$

using part unfolding xs by auto

have $\text{Pos}v: \text{Pos } v \notin \# \chi$ using χ unfolding true-cls-def true-lit-def by auto

have $\text{Neg}v: \text{Neg } v \notin \# \chi'$ using χ' unfolding true-cls-def true-lit-def by auto

{

assume $\text{Neg}\chi: \neg \text{Neg } v \in \# \chi$

then have $\neg I \models \chi$ using $\chi \ \text{Pos}v$ unfolding true-cls-def true-lit-def by auto

moreover have $\text{total-over-m } I \ \{\chi\}$

using $\text{Pos}v \ \text{Neg}\chi \ \text{atm-imp-pos-or-neg-lit tot}\chi$ unfolding total-over-m-def total-over-set-def
by fastforce

ultimately have $\text{partial-interps Leaf } I (\text{fst } \psi)$

and $\text{sem-tree-size Leaf} < \text{sem-tree-size } xs$

and $\text{resolution}^{**} \psi \psi$

unfolding xs by (auto simp add: $\chi\psi$)

}

moreover {

assume $\text{Pos}\chi: \neg \text{Pos } v \in \# \chi'$

then have $I\chi: \neg I \models \chi'$ using $\chi' \ \text{Pos}v$ unfolding true-cls-def true-lit-def by auto

moreover have $\text{total-over-m } I \ \{\chi'\}$

using $\text{Neg}v \ \text{Pos}\chi \ \text{atm-imp-pos-or-neg-lit tot}\chi'$

```

    unfolding total-over-m-def total-over-set-def by fastforce
    ultimately have partial-interps Leaf I (fst  $\psi$ )
    and sem-tree-size Leaf < sem-tree-size xs
    and resolution**  $\psi$   $\psi$  using  $\chi'\psi$   $I\chi$  unfolding xs by auto
  }
  moreover {
    assume neg: Neg v  $\in\#$   $\chi$  and pos: Pos v  $\in\#$   $\chi'$ 
    have count  $\chi$  (Neg v) = 1
      using simplified-count[OF simp  $\chi\psi$ ] neg by (metis One-nat-def Suc-le-mono Suc-pred eq-iff le0)
    have count  $\chi'$  (Pos v) = 1
      using simplified-count[OF simp  $\chi'\psi$ ] pos by (metis One-nat-def Suc-le-mono Suc-pred eq-iff le0)
    obtain C where  $\chi C$ :  $\chi = C + \{\#Neg\ v\# \}$  and negC: Neg v  $\notin\#$  C and posC: Pos v  $\notin\#$  C
    proof -
      assume a1:  $\bigwedge C. [\chi = C + \{\#Neg\ v\# \}; Neg\ v \notin\# C; Pos\ v \notin\# C] \implies thesis$ 
      have f2:  $\bigwedge n. (0::nat) + n = n$ 
        by simp
      obtain mm :: 'v literal multiset  $\Rightarrow$  'v literal  $\Rightarrow$  'v literal multiset where
        f3:  $\{\#Neg\ v\# \} + mm\ \chi\ (Neg\ v) = \chi$ 
        by (metis (no-types)  $\langle count\ \chi\ (Neg\ v) = 1 \rangle$  add.commute multi-member-split zero-less-one)
      then have Pos v  $\notin\#$  mm  $\chi$  (Neg v)
        using f2 by (metis (no-types) Posv  $\langle count\ \chi\ (Neg\ v) = 1 \rangle$  add.right-neutral add-left-cancel count-single count-union less-nat-zero-code)
      then show ?thesis
        using f3 a1 by (metis (no-types)  $\langle count\ \chi\ (Neg\ v) = 1 \rangle$  add.commute add.right-neutral add-left-cancel count-single count-union less-nat-zero-code)
    qed
    obtain C' where
       $\chi C'$ :  $\chi' = C' + \{\#Pos\ v\# \}$  and
      posC': Pos v  $\notin\#$  C' and
      negC': Neg v  $\notin\#$  C'
    by (metis (no-types, hide-lams) Negv  $\langle count\ \chi'\ (Pos\ v) = 1 \rangle$  add.diff-cancel-right' cancel-comm-monoid-add-class.diff-cancel count-diff count-single less-nat-zero-code mset-leD mset-le-add-left multi-member-split zero-less-one)

    have totC: total-over-m I {C}
      using tot $\chi$  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  $\psi$  ( $N'$ ,  $\text{snd } \psi \cup \{(\chi', \chi)\}$ )
  using resolution.intros(2)[OF - simp full, of  $\text{snd } \psi$   $\text{snd } \psi \cup \{(\chi', \chi)\}$ ] 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  $a$ : ( $\{\#Pos\ v\# \} + C'$ ,  $\{\#Neg\ v\# \} + C$ )  $\in \text{snd } \psi$ 
  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  $p$ :  $Pos\ p \in \# (\{\#Pos\ v\# \} + C') \wedge Neg\ p \in \# (\{\#Neg\ v\# \} + C)$ 
       $\wedge ((\exists \chi \in \text{fst } \psi. (\forall I. \text{total-over-m } I \{(\{\#Pos\ v\# \} + C') - \{\#Pos\ p\# \} + ((\{\#Neg\ v\# \} + C) - \{\#Neg\ p\# \})) \longrightarrow \text{total-over-m } I \{\chi\})$ 
       $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models (\{\#Pos\ v\# \} + C') - \{\#Pos\ p\# \} + ((\{\#Neg\ v\# \} + C) - \{\#Neg\ p\# \}))) \vee \text{tautology } ((\{\#Pos\ v\# \} + C') - \{\#Pos\ p\# \} + ((\{\#Neg\ v\# \} + C) - \{\#Neg\ p\# \})))$ 
    using  $a$  by (blast intro: allE[OF a-u-i[unfolded subsumes-def Ball-def],
      of ( $\{\#Pos\ v\# \} + C'$ ,  $\{\#Neg\ v\# \} + C$ ))]
    { assume  $p \neq v$ 
      then have  $Pos\ p \in \# C' \wedge Neg\ p \in \# C$  using  $p$  by force
      then have ?thesis by (metis add-gr-0 count-union tautology-Pos-Neg)
    }
    moreover {
      assume  $p = v$ 
      then have ?thesis using  $p$  by (metis add.commute add-diff-cancel-left')
    }
    ultimately show ?thesis by auto
  qed
moreover {
  assume  $\exists \chi \in \text{fst } \psi. (\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\chi\})$ 
     $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ 
  then obtain  $\vartheta$  where
     $\vartheta$ :  $\vartheta \in \text{fst } \psi$  and
     $\text{tot-}\vartheta\text{-}CC'$ :  $\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\vartheta\}$  and
     $\vartheta\text{-inv}$ :  $\forall I. \text{total-over-m } I \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$  by blast
  have partial-interps Leaf  $I$  ( $\text{fst } \psi$ )
    using  $\text{tot-}\vartheta\text{-}CC'$   $\vartheta$   $\vartheta\text{-inv}$   $\text{tot}C$   $\text{tot}C'$   $\hookrightarrow I \models C + C'$   $\text{total-over-m-sum}$  by fastforce
  moreover have sem-tree-size Leaf < sem-tree-size  $xs$  unfolding  $xs$  by auto
  ultimately have ?case by blast
}
moreover {
  assume  $\text{taut}CC'$ :  $\text{tautology } (C' + C)$ 
  have  $\text{total-over-m } I \{C'+C\}$  using  $\text{tot}C$   $\text{tot}C'$   $\text{total-over-m-sum}$  by auto
  then have  $\neg \text{tautology } (C' + C)$ 
    using  $\hookrightarrow I \models C + C'$  unfolding add.commute[of  $C$   $C'$ ]  $\text{total-over-m-def}$ 
    unfolding  $\text{tautology-def}$  by auto
  then have False using  $\text{taut}CC'$  unfolding  $\text{tautology-def}$  by auto
}
ultimately have ?case by auto
}
ultimately have ?case by auto
}

```

```

    ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
  }
  moreover {
    assume size-ag: sem-tree-size ag > 0
    have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
    moreover have partial-interps ag (I ∪ {Pos v}) (fst ψ)
    and partad: partial-interps ad (I ∪ {Neg v}) (fst ψ)
      using part partial-interps.simps(2) unfolding xs by metis+
    moreover
      have sem-tree-size ag < sem-tree-size xs ⟹ finite (fst ψ) ⟹ already-used-inv ψ
        ⟹ partial-interps ag (I ∪ {Pos v}) (fst ψ) ⟹ simplified (fst ψ)
        ⟹ ∃ tree' ψ'. resolution** ψ ψ' ∧ partial-interps tree' (I ∪ {Pos v}) (fst ψ')
          ∧ (sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0)
      using IH[of ag I ∪ {Pos v}] by auto
    ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
      inf: resolution** ψ ψ'
      and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')
      and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
      using finite part rtranclp.rtrancl-refl a-u-i simp by blast

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

    have partial-interps ag (I ∪ {Pos v}) (fst ψ')
      using rtranclp-resolution-preserve-partial-tree inf partag by fast
    then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
    then have ?case using inf size size-ad unfolding xs by fastforce
  }
  ultimately have ?case by auto
}
ultimately show ?case by auto
qed

```

lemma resolution-completeness-inv:

```

fixes  $\psi :: 'v :: \text{linorder state}$ 
assumes
   $\text{unsat}: \neg \text{satisfiable (fst } \psi)$  and
   $\text{finite}: \text{finite (fst } \psi)$  and
   $\text{a-u-v}: \text{already-used-inv } \psi$ 
shows  $\exists \psi'. (\text{resolution}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$ 
proof –
  obtain  $\text{tree}$  where  $\text{partial-interps tree } \{\} (\text{fst } \psi)$ 
  using  $\text{partial-interps-build-sem-tree-atms assms}$  by  $\text{metis}$ 
then show  $?thesis$ 
  using  $\text{unsat finite a-u-v}$ 
  proof ( $\text{induct tree arbitrary: } \psi$   $\text{rule: sem-tree-size}$ )
    case ( $\text{bigger tree } \psi$ ) 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  $v \text{ tree1 tree2}$ 
    assume  $\text{tree: tree} = \text{Node } v \text{ tree1 tree2}$ 
    obtain  $\psi_0$  where  $\psi_0: \text{resolution}^{**} \psi \psi_0$  and  $\text{simp: simplified (fst } \psi_0)$ 
    proof –
      { assume  $\text{simplified (fst } \psi)$ 
        moreover have  $\text{resolution}^{**} \psi \psi$  by  $\text{auto}$ 
        ultimately have  $thesis$  using  $that$  by  $\text{blast}$ 
      }
    moreover {
      assume  $\neg \text{simplified (fst } \psi)$ 
      then have  $\exists \psi'. \text{full1 simplify (fst } \psi) \psi'$ 
      by ( $\text{metis Nitpick.rtranclp-unfold bigger.prem}(3) \text{full1-def rtranclp-simplify-terminates}$ )
      then obtain  $N$  where  $\text{full1 simplify (fst } \psi) N$  by  $\text{metis}$ 
      then have  $\text{resolution } \psi (N, \text{snd } \psi)$ 
      using  $\text{resolution.intros}(1)[\text{of fst } \psi N \text{snd } \psi]$  by  $\text{auto}$ 
      moreover have  $\text{simplified } N$ 
      using  $\langle \text{full1 simplify (fst } \psi) N \rangle$  unfolding  $\text{full1-def}$  by  $\text{blast}$ 
      ultimately have  $?thesis$  using  $that$  by  $\text{force}$ 
    }
    ultimately show  $?thesis$  by  $\text{auto}$ 
  }
qed

have  $p: \text{partial-interps tree } \{\} (\text{fst } \psi_0)$ 
and  $\text{uns}: \text{unsatisfiable (fst } \psi_0)$ 
and  $f: \text{finite (fst } \psi_0)$ 
and  $\text{a-u-v}: \text{already-used-inv } \psi_0$ 
  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  $\psi_0$  bigger.premis(3) rtrancp-resolution-finite apply blast
    using rtrancp-resolution-already-used-inv[OF  $\psi_0$  bigger.premis(4)] by blast
  obtain tree'  $\psi'$  where
    inf: resolution**  $\psi_0 \psi'$  and
    part': partial-interps tree' {} (fst  $\psi'$ ) and
    decrease: sem-tree-size tree' < sem-tree-size tree  $\vee$  sem-tree-size tree = 0
    using can-decrease-tree-size-resolution[OF f a-u-v p simp] unfolding tautology-def
    by meson
  have s: sem-tree-size tree' < sem-tree-size tree using decrease unfolding tree by auto
  have fin: finite (fst  $\psi'$ )
    using f inf rtrancp-resolution-finite by blast
  have unsat: unsatisfiable (fst  $\psi'$ )
    using rtrancp-resolution-preserves-unsat inf uns by metis
  have a-u-i': already-used-inv  $\psi'$ 
    using a-u-v inf rtrancp-resolution-already-used-inv[of  $\psi_0 \psi'$ ] by auto
  have ?case
    using inf rtrancp-trans[of resolution] H(1)[OF s part' unsat fin a-u-i']  $\psi_0$  by blast
}
ultimately show ?case by (case-tac tree, auto)
qed
qed

```

```

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

```

```

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

```

```

lemma resolution-completeness:
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes unsat:  $\neg$ 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 rtrancp-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  $\psi$ )
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  $\psi$ )
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  $\psi$ 
and  $\{\#\} \in \psi$ 
shows  $\psi = \{\{\#\}\}$ 
proof (rule ccontr)
assume H:  $\neg ?thesis$ 
then obtain  $\chi$  where  $\chi \in \psi$  and  $\chi \neq \{\#\}$  using assms(2) by blast
then have  $\{\#\} \subsetneq \chi$  by (simp add: mset-less-empty-nonempty)
then have simplify  $\psi (\psi - \{\chi\})$ 
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 rtrancpl-simplify-falsity-in-preserved:
  assumes simplify**  $\chi s \chi s'$ 
  and  $\{\#\} \in \chi s$ 
  shows  $\{\#\} \in \chi s'$ 
  using assms
  by induction (auto intro: simplify-falsity-in-preserved)

lemma resolution-falsity-get-falsity-alone:
  assumes finite (fst  $\psi$ )
  shows  $(\exists \psi'. (\text{resolution}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow (\exists a-u-v. \text{resolution}^{**} \psi (\{\{\#\}\}, a-u-v))$ 
  (is  $?A \longleftrightarrow ?B$ )
proof
  assume  $?B$ 
  then show  $?A$  by auto
next
  assume  $?A$ 
  then obtain  $\chi s$  a-u-v where  $\chi s: \text{resolution}^{**} \psi (\chi s, a-u-v)$  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  $\neg$  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 rtrancpl-resolution-finite)
    then have  $\{\#\} \in \chi s'$ 
      unfolding full1-def by (meson F rtrancpl-simplify-falsity-in-preserved
        trancpl-into-rtrancpl)
    then have  $?B$ 
      by (metis  $\chi s$  (full1 simplify  $\chi s \chi s'$ ) fst-conv full1-simp resolution-always-simplified
        rtrancpl.rtrancpl-into-rtrancpl 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, 'lvl, 'mark) *marked-lit* =
is-marked: *Marked* (lit-of: 'v literal) (level-of: 'lvl) |
is-proped: *Propagated* (lit-of: 'v literal) (mark-of: 'mark)

lemma *marked-lit-list-induct*[case-names nil marked proped]:
assumes $P \ []$ **and**
 $\bigwedge L \ l \ xs. P \ xs \implies P \ (\text{Marked } L \ l \ \# \ xs)$ **and**
 $\bigwedge L \ m \ xs. P \ xs \implies P \ (\text{Propagated } L \ m \ \# \ xs)$
shows $P \ xs$
using *assms* **apply** (*induction xs, simp*)
by (*case-tac a*) *auto*

lemma *is-marked-ex-Marked*:
 $\text{is-marked } L \implies \exists K \ lvl. L = \text{Marked } K \ lvl$
by (*cases L*) *auto*

type-synonym ('v, 'l, 'm) *marked-lits* = ('v, 'l, 'm) *marked-lit list*

definition *lits-of* :: ('a, 'b, 'c) *marked-lit list* \Rightarrow 'a *literal set* **where**
lits-of $Ls = \text{lit-of } \text{' (set } Ls)$

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

lemma *lits-of-cons*[*simp*]:
 $\text{lits-of } (L \ \# \ Ls) = \text{insert } (\text{lit-of } L) \ (\text{lits-of } Ls)$
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-append*[*simp*]:
 $\text{lits-of } (l \ @ \ l') = \text{lits-of } l \cup \text{lits-of } l'$
unfolding *lits-of-def* **by** *auto*

lemma *finite-lits-of-def*[*simp*]: *finite* (*lits-of* L)
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-rev*[*simp*]: *lits-of* (*rev* M) = *lits-of* M
unfolding *lits-of-def* **by** *auto*

lemma *set-map-lit-of-lits-of*[*simp*]:
 $\text{set } (\text{map } \text{lit-of } T) = \text{lits-of } T$
unfolding *lits-of-def* **by** *auto*

lemma *atms-of-ms-lambda-lit-of-is-atm-of-lit-of*[*simp*]:
 $\text{atms-of-ms } ((\lambda a. \{\# \text{lit-of } a \# \}) \ \text{' set } M') = \text{atm-of } \text{' lits-of } M'$
unfolding *atms-of-ms-def lits-of-def* **by** *auto*

lemma *lits-of-empty-is-empty*[iff]:
 $\text{lits-of } M = \{\} \longleftrightarrow M = []$
by (*induct* M) *auto*

13.1.2 Entailment

definition *true-annot* :: ('a, 'l, 'm) *marked-lits* \Rightarrow 'a *clause* \Rightarrow bool (**infix** \models_a 49) **where**
 $I \models_a C \longleftrightarrow (\text{lits-of } I) \models C$

definition *true-annots* :: ('a, 'l, 'm) *marked-lits* \Rightarrow 'a *clauses* \Rightarrow bool (**infix** \models_{as} 49) **where**
 $I \models_{as} CC \longleftrightarrow (\forall C \in CC. I \models_a C)$

lemma *true-annot-empty-model*[simp]:
 $\neg[] \models_a \psi$
unfolding *true-annot-def* *true-cls-def* **by** *simp*

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

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

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

lemma *true-annots-single-true-annot*[iff]:
 $I \models_{as} \{C\} \longleftrightarrow I \models_a C$
unfolding *true-annots-def* **by** *auto*

lemma *true-annot-insert-l*[simp]:
 $M \models_a A \Longrightarrow L \# M \models_a A$
unfolding *true-annot-def* **by** *auto*

lemma *true-annots-insert-l* [simp]:
 $M \models_{as} A \Longrightarrow L \# M \models_{as} A$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-union*[iff]:
 $M \models_{as} A \cup B \longleftrightarrow (M \models_{as} A \wedge M \models_{as} B)$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-insert*[iff]:
 $M \models_{as} \text{insert } a \ A \longleftrightarrow (M \models_a a \wedge M \models_{as} A)$
unfolding *true-annots-def* **by** *auto*

Link between \models_{as} and \models_s :

lemma *true-annots-true-cls*:
 $I \models_{as} CC \longleftrightarrow (\text{lits-of } I) \models_s CC$
unfolding *true-annots-def* *Ball-def* *true-annot-def* *true-clss-def* **by** *auto*

lemma *in-lit-of-true-annot*:
 $a \in \text{lits-of } M \longleftrightarrow M \models_a \{\#a\#\}$

unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annot-lit-of-notin-skip*:

$L \# M \models_a A \implies \text{lit-of } L \not\subseteq \# A \implies M \models_a A$

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

lemma *true-clss-singleton-lit-of-implies-incl*:

$I \models_s (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } MLs \implies \text{lits-of } MLs \subseteq I$

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

lemma *true-annot-true-clss-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]*:

$\text{map } (\lambda M. \text{Marked } M \ a) \ M \models_{as} N \longleftrightarrow \text{set } M \models_s N$

proof –

have *: $\text{lits-of } (\text{map } (\lambda M. \text{Marked } M \ a) \ M) = \text{set } M$ **unfolding** *lits-of-def* **by** *force*

show ?thesis **by** (*simp add: true-annots-true-clss **)

qed

lemma *true-annot-singleton[iff]*: $M \models_a \{\# L \# \} \longleftrightarrow L \in \text{lits-of } M$

unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annots-true-clss-clss*:

$A \models_{as} \Psi \implies (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } A \models_{ps} \Psi$

unfolding *true-clss-clss-def true-annots-def true-clss-def*

by (*auto*

dest!: true-clss-singleton-lit-of-implies-incl

simp add: lits-of-def true-annot-def true-clss-def)

lemma *true-annot-commute*:

$M @ M' \models_a D \longleftrightarrow M' @ M \models_a D$

unfolding *true-annot-def* **by** (*simp add: Un-commute*)

lemma *true-annots-commute*:

$M @ M' \models_{as} D \longleftrightarrow M' @ M \models_{as} D$

unfolding *true-annots-def* **by** (*auto simp add: true-annot-commute*)

lemma *true-annot-mono[dest]*:

$\text{set } I \subseteq \text{set } I' \implies I \models_a N \implies I' \models_a N$

using *true-clss-mono-set-mset-l* **unfolding** *true-annot-def lits-of-def*

by (*metis (no-types) Un-commute Un-upper1 image-Un sup.orderE*)

lemma *true-annots-mono*:

$\text{set } I \subseteq \text{set } I' \implies I \models_{as} N \implies I' \models_{as} N$

unfolding *true-annots-def* **by** *auto*

13.1.3 Defined and undefined literals

definition *defined-lit* :: ('a, 'l, 'm) marked-lit list \Rightarrow 'a literal \Rightarrow bool

where

defined-lit I L \longleftrightarrow ($\exists l. \text{Marked } L \ l \in \text{set } I$) \vee ($\exists P. \text{Propagated } L \ P \in \text{set } I$)
 \vee ($\exists l. \text{Marked } (-L) \ l \in \text{set } I$) \vee ($\exists P. \text{Propagated } (-L) \ P \in \text{set } I$)

abbreviation *undefined-lit* :: ('a, 'l, 'm) marked-lit list \Rightarrow 'a literal \Rightarrow bool

where *undefined-lit* I L $\equiv \neg \text{defined-lit } I \ L$

lemma *defined-lit-rev[simp]*:

defined-lit (rev M) L \longleftrightarrow *defined-lit* M L

unfolding *defined-lit-def* **by** *auto*

lemma *atm-imp-marked-or-proped*:

assumes $x \in \text{set } I$

shows

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

using *assms marked-lit.exhaust-sel* **by** *metis*

lemma *literal-is-lit-of-marked*:

assumes $L = \text{lit-of } x$

shows ($\exists l. x = \text{Marked } L \ l$) \vee ($\exists l'. x = \text{Propagated } L \ l'$)

using *assms* **by** (*case-tac x*) *auto*

lemma *true-annot-iff-marked-or-true-lit*:

defined-lit I L \longleftrightarrow ((*lits-of* I) $\models_l L \vee$ (*lits-of* 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* I) \Longrightarrow I \models_{as} N \Longrightarrow *satisfiable* N

by (*simp add: true-annots-true-cls*)

lemma *defined-lit-map*:

defined-lit Ls L \longleftrightarrow *atm-of* L \in ($\lambda l. \text{atm-of } (\text{lit-of } l)$) ' 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 *defined-lit* I L

unfolding *defined-lit-def* **by** *auto*

lemma *Marked-Propagated-in-iff-in-lits-of*:

defined-lit I L \longleftrightarrow ($L \in \text{lits-of } I \vee -L \in \text{lits-of } I$)

unfolding *lits-of-def* *defined-lit-def*

by (*auto simp add: rev-image-eqI*) (*case-tac x, auto*) $+$

lemma *consistent-add-undefined-lit-consistent[simp]*:

assumes

consistent-interp (*lits-of* Ls) **and**

undefined-lit Ls L

shows *consistent-interp* (*insert* L (*lits-of* Ls))

using *assms* **unfolding** *consistent-interp-def* **by** (*auto simp: Marked-Propagated-in-iff-in-lits-of*)

lemma *decided-empty[simp]*:
 $\neg \text{defined-lit } [] \ L$
unfolding *defined-lit-def* **by** *simp*

13.2 Backtracking

fun *backtrack-split* :: ('v, 'l, 'm) *marked-lits*
 \Rightarrow ('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*: $a \in \text{set } (\text{fst } (\text{backtrack-split } l)) \Rightarrow \neg \text{is-marked } a$
by (*induct l rule: marked-lit-list-induct*) *auto*

lemma *backtrack-split-snd-hd-marked*:
 $\text{snd } (\text{backtrack-split } l) \neq [] \Rightarrow \text{is-marked } (\text{hd } (\text{snd } (\text{backtrack-split } l)))$
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*:
 $\text{backtrack-split } M = (M'', []) \Rightarrow \forall l \in \text{set } M. \neg \text{is-marked } l$
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*:
 $\exists l \in \text{set } M. \text{is-marked } l \Rightarrow \exists M' L' M''. \text{backtrack-split } M = (M'', L' \# M')$
by (*metis backtrack-snd-empty-not-marked list.exhaust prod.collapse*)

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma *backtrack-split-takeWhile-dropWhile*:
 $\text{backtrack-split } M = (\text{takeWhile } (\text{Not } o \text{ is-marked}) M, \text{dropWhile } (\text{Not } o \text{ is-marked}) M)$
proof (*induct M*)
case *Nil* **show** ?*case* **by** *simp*
next
case (*Cons L M*) **thus** ?*case* **by** (*cases L*) *auto*
qed

13.3 Decomposition with respect to the marked literals

The pattern *get-all-marked-decomposition* [] = [([]), []] 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 False

by (*induct M, simp*) (*case-tac a, auto*)

lemma *get-all-marked-decomposition-never-empty-sym*[*iff*]:

[] = get-all-marked-decomposition M \longleftrightarrow False

using *get-all-marked-decomposition-never-empty*[*of M*] **by** *presburger*

lemma *get-all-marked-decomposition-decomp*:

hd (get-all-marked-decomposition S) = (a, c) \implies S = c @ a

proof (*induct S arbitrary: a c*)

case *Nil*

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

next

case (*Cons x A*)

thus ?*case* **by** (*cases x; cases hd (get-all-marked-decomposition A)*) *auto*

qed

lemma *get-all-marked-decomposition-backtrack-split*:

backtrack-split S = (M, M') \longleftrightarrow hd (get-all-marked-decomposition S) = (M', M)

proof (*induction S arbitrary: M M'*)

case *Nil*

thus ?*case* **by** *auto*

next

case (*Cons a S*)

thus ?*case* **using** *backtrack-split-takeWhile-dropWhile* **by** (*cases a*) *force+*

qed

lemma *get-all-marked-decomposition-nil-backtrack-split-snd-nil*:

get-all-marked-decomposition S = [([], A)] \implies snd (backtrack-split S) = []

by (*simp add: get-all-marked-decomposition-backtrack-split sndI*)

lemma *get-all-marked-decomposition-length-1-fst-empty-or-length-1*:

assumes *get-all-marked-decomposition M = (a, b) # []*

shows *a = [] \vee (length a = 1 \wedge is-marked (hd a) \wedge hd a \in set M)*

using *assms*

proof (*induct M arbitrary: a b*)

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

next

case (*Cons m M*)

show ?*case*

proof (*cases m*)

case (*Marked l mark*)

thus ?*thesis* **using** *Cons* **by** *simp*

next

case (*Propagated l mark*)

thus ?*thesis* **using** *Cons* **by** (*cases get-all-marked-decomposition M*) *force+*

qed

qed

lemma *get-all-marked-decomposition-fst-empty-or-hd-in-M:*
assumes *get-all-marked-decomposition* $M = (a, b) \# l$
shows $a = [] \vee (\text{is-marked } (\text{hd } a) \wedge \text{hd } a \in \text{set } M)$
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 (*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*

thus *?case* **by** *auto*

next

case (*Cons* L $M0$)

hence *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)

thus *?thesis* **using** *xy* **by** *auto*

next

case (*Propagated* l m)

thus *?thesis*

using *xy Cons.prem*s

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:*

$\text{set } M = \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-marked-decomposition } M)) \cup \{L \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$

(is ?M M = ?U M \cup ?Ls M)

proof (*induct* M *arbitrary*:)

case *Nil*

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

next

case (*Cons* L M)

show *?case*

```

proof (cases L)
  case (Marked a l) note L = this
  hence L ∈ ?Ls (L#M) by auto
  moreover have ?U (L#M) = ?U M unfolding L by auto
  moreover have ?M M = ?U M ∪ ?Ls M using Cons.hyps by auto
  ultimately show ?thesis by auto
next
  case (Propagated a P)
  thus ?thesis using Cons.hyps by (cases (get-all-marked-decomposition M)) auto
qed
qed

```

lemma in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend:

```

(a, b) ∈ set (get-all-marked-decomposition M') ⇒
  ∃ 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 (case-tac get-all-marked-decomposition (xs @ M')) auto

```

lemma get-all-marked-decomposition-remove-unmarked-length:

```

assumes ∀ l ∈ set M'. ¬is-marked l
shows length (get-all-marked-decomposition (M' @ M''))
  = length (get-all-marked-decomposition M'')
using assms by (induct M' arbitrary: M'' rule: marked-lit-list-induct) auto

```

lemma get-all-marked-decomposition-not-is-marked-length:

```

assumes ∀ l ∈ set M'. ¬is-marked l
shows 1 + length (get-all-marked-decomposition (Propagated (-L) P # M))
  = length (get-all-marked-decomposition (M' @ Marked L l # M))
using assms get-all-marked-decomposition-remove-unmarked-length by fastforce

```

lemma get-all-marked-decomposition-last-choice:

```

assumes tl (get-all-marked-decomposition (M' @ Marked L l # M)) ≠ []
and ∀ l ∈ set M'. ¬is-marked l
and hd (tl (get-all-marked-decomposition (M' @ Marked L l # M))) = (M0', M0)
shows hd (get-all-marked-decomposition (Propagated (-L) P # M)) = (M0', 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 ∀ l ∈ set M'. ¬is-marked l
shows tl (get-all-marked-decomposition (Propagated (-L) P # M))
  = tl (tl (get-all-marked-decomposition (M' @ Marked L l # M)))
using assms by (induct M' rule: marked-lit-list-induct) auto

```

lemma get-all-marked-decomposition-hd-hd:

```

assumes get-all-marked-decomposition Ls = (M, C) # (M0, M0') # l
shows tl M = M0' @ M0 ∧ is-marked (hd M)
using assms
proof (induct Ls arbitrary: M C M0 M0' l)
  case Nil
  thus ?case by simp
next
  case (Cons a Ls M C M0 M0' l) note IH = 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)
  hence tl M = M0' @ M0 ∧ 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 (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 (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

```

definition *all-decomposition-implies* :: 'a literal multiset set
 $\Rightarrow ((\text{'a}, \text{'l}, \text{'m}) \text{ marked-lit list} \times (\text{'a}, \text{'l}, \text{'m}) \text{ marked-lit list}) \text{ list} \Rightarrow \text{bool})$ **where**
all-decomposition-implies N S
 $\longleftrightarrow (\forall (Ls, seen) \in \text{set } S. (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } Ls \cup N \models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set seen})$

```

lemma all-decomposition-implies-empty[iff]:
  all-decomposition-implies N [] unfolding all-decomposition-implies-def by auto

```

```

lemma all-decomposition-implies-single[iff]:
  all-decomposition-implies N [(Ls, seen)]
 $\longleftrightarrow (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set } Ls \cup N \models_{ps} (\lambda a. \{\# \text{lit-of } a \# \}) \text{ ' set seen}$ 
  unfolding all-decomposition-implies-def by auto

```

```

lemma all-decomposition-implies-append[iff]:
  all-decomposition-implies  $N$  ( $S @ S'$ )
     $\longleftrightarrow$  (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  $N$  ( $(Ls, seen) \# S'$ )
     $\longleftrightarrow$  (all-decomposition-implies  $N$   $[(Ls, seen)] \wedge$  all-decomposition-implies  $N$   $S'$ )
  unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-cons-single[iff]:
  all-decomposition-implies  $N$  ( $l \# S'$ )  $\longleftrightarrow$ 
    ( $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } (fst\ l) \cup N \models ps (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } (snd\ l) \wedge$ 
     all-decomposition-implies  $N$   $S'$ )
  unfolding all-decomposition-implies-def by auto

lemma all-decomposition-implies-trail-is-implied:
  assumes all-decomposition-implies  $N$  (get-all-marked-decomposition  $M$ )
  shows  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
     $\models ps (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \bigcup (\text{set ' snd ' set } (\text{get-all-marked-decomposition } M))$ 
  using assms
proof (induct length (get-all-marked-decomposition M) arbitrary: M)
  case 0
  thus ?case by auto
next
case (Suc n) note  $IH = \text{this}(1)$  and  $\text{length} = \text{this}(2)$ 
  {
    assume  $\text{length } (\text{get-all-marked-decomposition } M) \leq 1$ 
    then obtain  $a\ b$  where  $g: \text{get-all-marked-decomposition } M = (a, b) \# []$ 
    by (case-tac get-all-marked-decomposition M) auto
    moreover {
      assume  $a = []$ 
      hence ?case using  $\text{Suc.prem } g$  by auto
    }
    moreover {
      assume  $l: \text{length } a = 1$  and  $m: \text{is-marked } (hd\ a)$  and  $hd: hd\ a \in \text{set } M$ 
      hence  $(\lambda a. \{\#lit\text{-of } a\# \}) (hd\ a) \in \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$  by auto
      hence  $H: (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup N \subseteq N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
      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  $\text{Suc.prem}$  unfolding all-decomposition-implies-def  $g$  by simp
      have ?case
      unfolding  $g$  apply (rule true-clss-clss-subset) using  $f1\ H$  by auto
    }
    ultimately have ?case using get-all-marked-decomposition-length-1-fst-empty-or-length-1 by blast
  }
  moreover {
    assume  $\text{length } (\text{get-all-marked-decomposition } M) > 1$ 
    then obtain  $Ls0\ seen0\ M'$  where
       $Ls0: \text{get-all-marked-decomposition } M = (Ls0, seen0) \# \text{get-all-marked-decomposition } M'$  and
       $\text{length}' : \text{length } (\text{get-all-marked-decomposition } M') = n$  and
       $M'\text{-in-}M: \text{set } M' \subseteq \text{set } M$ 
    using  $\text{length}$  apply (induct M)
    apply simp
  }

```



```

by (case-tac a, case-tac hd (get-all-marked-decomposition M))
  (auto simp add: subset-insertI2)
{
  assume n = 0
  hence get-all-marked-decomposition M' = [] using length' by auto
  hence ?case using Suc.premis unfolding all-decomposition-implies-def Ls0 by auto
}
moreover {
  assume n: n > 0
  then obtain Ls1 seen1 l where Ls1: get-all-marked-decomposition M' = (Ls1, seen1) # l
    using length' by (induct M', simp) (case-tac a, auto)

  have all-decomposition-implies N (get-all-marked-decomposition M')
    using Suc.premis unfolding Ls0 all-decomposition-implies-def by auto
  hence N: N  $\cup$  { $\{\#lit\text{-of } L\# \mid L. is\text{-marked } L \wedge L \in set\ M'\}$ }
     $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \bigcup (set \text{ ' } snd \text{ ' } set (get\text{-all-marked-decomposition } M'))$ 
    using IH length' by auto

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

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

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

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

```

hence $N \cup \{\{\#lit\text{-of } L\# \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}\}$
 $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set (tl Ls0)}$
using $M' \text{-in-} M \text{ true-clss-clss-subset}[OF - t,$
 $\text{ of } N \cup \{\{\#lit\text{-of } L\# \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}\}]$
by *auto*
hence $N \cup \{\{\#lit\text{-of } L\# \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}\} \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set Ls0}$
using *hd-Ls0* **by** (*case-tac Ls0, auto*)

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

hence *?case unfolding Ls0 using $\Psi \Psi N$ by simp*
}
ultimately have *?case by auto*
}
ultimately show *?case by arith*
qed

lemma *all-decomposition-implies-propagated-lits-are-implied:*
assumes *all-decomposition-implies N (get-all-marked-decomposition M)*
shows $N \cup \{\{\#lit\text{-of } L\# \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}\} \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set M}$
(is ?I \models_{ps} ?A)
proof –
have $?I \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \{L \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}$
by (*auto intro: all-in-true-clss-clss*)
moreover have $?I \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' } \bigcup (\text{set ' snd ' set (get-all-marked-decomposition M)})$
using *all-decomposition-implies-trail-is-implied assms* **by** *blast*
ultimately have $N \cup \{\{\#lit\text{-of } m\# \mid m. \text{ is-marked } m \wedge m \in \text{set } M\}\}$
 $\models_{ps} (\lambda m. \{\#lit\text{-of } m\# \}) \text{ ' } \bigcup (\text{set ' snd ' set (get-all-marked-decomposition M)})$
 $\cup (\lambda m. \{\#lit\text{-of } m\# \}) \text{ ' } \{m \mid m. \text{ is-marked } m \wedge m \in \text{set } M\}$
by *blast*
thus *?thesis*
by (*metis (no-types) get-all-marked-decomposition-snd-union[of M] image-Un*)
qed

lemma *all-decomposition-implies-insert-single:*
 $\text{all-decomposition-implies } N M \implies \text{all-decomposition-implies (insert } C N) M$
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 $\{\#L\# \in CNot \psi \longleftrightarrow -L \in \# \psi$

using *assms* **unfolding** *CNot-def* **by** *force*

lemma *CNot-singleton[simp]*: $CNot \{\#L\# \} = \{\{\#-L\#\}\}$ **unfolding** *CNot-def* **by** *auto*

lemma *CNot-empty[simp]*: $CNot \{\# \} = \{\}$ **unfolding** *CNot-def* **by** *auto*

lemma *CNot-plus[simp]*: $CNot (A + B) = CNot A \cup CNot B$ **unfolding** *CNot-def* **by** *auto*

lemma *CNot-eq-empty[iff]*:
 $CNot D = \{\} \longleftrightarrow D = \{\#\}$
unfolding *CNot-def* **by** (*auto simp add: multiset-eqI*)

lemma *in-CNot-implies-uminus*:
assumes $L \in \# D$
and $M \models_{as} CNot D$
shows $M \models_a \{\#-L\# \}$ **and** $-L \in lits_of M$
using *assms* **by** (*auto simp add: true-annot-def true-annot-def CNot-def*)

lemma *CNot-remdups-mset[simp]*:
 $CNot (remdups_mset A) = CNot A$
unfolding *CNot-def* **by** *auto*

lemma *Ball-CNot-Ball-mset[simp]* :
 $(\forall x \in CNot D. P x) \longleftrightarrow (\forall L \in \# D. P \{\#-L\# \})$
unfolding *CNot-def* **by** *auto*

lemma *consistent-CNot-not*:
assumes *consistent-interp I*
shows $I \models_s CNot \varphi \implies \neg I \models \varphi$
using *assms* **unfolding** *consistent-interp-def true-clss-def true-clss-def* **by** *auto*

lemma *total-not-true-clss-true-clss-CNot*:
assumes *total-over-m I* $\{\varphi\}$ **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-clss-def CNot-def*
apply *clarify*
by (*case-tac L*) (*force intro: pos-lit-in-atms-of neg-lit-in-atms-of*)**+**

lemma *total-not-CNot*:
assumes *total-over-m I* $\{\varphi\}$ **and** $\neg I \models_s CNot \varphi$
shows $I \models \varphi$
using *assms* *total-not-true-clss-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-clss-false*:
 $C \in D \implies D \models_{ps} CNot C \implies D \models_p \{\#\}$
unfolding *true-clss-clss-def true-clss-clss-def total-over-m-def*
by (*metis Un-commute atms-of-empty atms-of-ms-CNot-atms-of atms-of-ms-insert atms-of-ms-union*
consistent-CNot-not insert-absorb sup-bot.left-neutral true-clss-def)

lemma *true-annots-CNot-all-atms-defined*:
assumes $M \models_{as} CNot T$ **and** $a1: L \in \# T$
shows $atm_of L \in atm_of \text{'lits_of } M$
by (*metis assms atm-of-uminus image-eqI in-CNot-implies-uminus(1) true-annot-singleton*)

lemma *true-clss-clss-false-left-right*:
assumes $\{\{\#L\#\}\} \cup B \models_p \{\#\}$
shows $B \models_{ps} CNot \{\#L\#\}$
unfolding *true-clss-clss-def true-clss-clss-def*
proof (*intro allI impI*)
fix I
assume
tot: *total-over-m* $I (B \cup CNot \{\#L\#\})$ **and**
cons: *consistent-interp* I **and**
 $I: I \models_s B$
have *total-over-m* $I (\{\{\#L\#\}\} \cup B)$ **using** *tot* **by** *auto*
hence $\neg I \models_s insert \{\#L\#\} B$
using *assms cons* **unfolding** *true-clss-clss-def* **by** *simp*
thus $I \models_s CNot \{\#L\#\}$
using *tot I* **by** (*cases L*) *auto*
qed

lemma *true-annots-true-clss-def-iff-negation-in-model*:
 $M \models_{as} CNot C \longleftrightarrow (\forall L \in \# C. -L \in lits\text{-}of\ M)$
unfolding *CNot-def true-annots-true-clss true-clss-def* **by** *auto*

lemma *consistent-CNot-not-tautology*:
consistent-interp $M \implies M \models_s CNot D \implies \neg tautology\ D$
by (*metis atms-of-ms-CNot-atms-of consistent-CNot-not satisfiable-carac' satisfiable-def tautology-def total-over-m-def*)

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

lemma *total-over-m-CNot-total-over-m*[*simp*]:
total-over-m $I (CNot\ C) = total\text{-}over\text{-}set\ I (atms\text{-}of\ C)$
unfolding *total-over-m-def total-over-set-def* **by** *auto*

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

lemma *true-clss-clss-plus-CNot*:
assumes *CC-L*: $A \models_p CC + \{\#L\#\}$
and *CNot-CC*: $A \models_{ps} CNot\ CC$
shows $A \models_p \{\#L\#\}$
unfolding *true-clss-clss-def true-clss-clss-def CNot-def total-over-m-def*
proof (*intro allI impI*)
fix I
assume *tot*: *total-over-set* $I (atms\text{-}of\text{-}ms\ (A \cup \{\{\#L\#\}\}))$
and *cons*: *consistent-interp* I
and $I: I \models_s A$
let $?I = I \cup \{Pos\ P \mid P. P \in atms\text{-}of\ CC \wedge P \notin atm\text{-}of\ 'I\}$
have *cons'*: *consistent-interp* $?I$
using *cons* **unfolding** *consistent-interp-def*
by (*auto simp add: uminus-lit-swap atms-of-def rev-image-eqI*)
have $I': ?I \models_s A$
using I *true-clss-union-increase* **by** *blast*
have *tot-CNot*: *total-over-m* $?I (A \cup CNot\ CC)$
using *tot atms-of-s-def* **by** (*fastforce simp add: total-over-m-def total-over-set-def*)

hence *tot-I-A-CC-L*: *total-over-m* ?*I* ($A \cup \{CC + \{\#L\#\}\}$)
 using *tot unfolding total-over-m-def total-over-set-atm-of* **by** *auto*
 hence ?*I* $\models CC + \{\#L\#\}$ using *CC-L cons' I'* **unfolding** *true-clss-clss-def* **by** *blast*
moreover
 have ?*I* $\models_s CNot\ CC$ using *CNot-CC cons' I'* *tot-CNot* **unfolding** *true-clss-clss-def* **by** *auto*
 hence $\neg A \models_p CC$
 by (*metis* (*no-types*, *lifting*) *I'* *atms-of-ms-CNot-atms-of-ms atms-of-ms-union cons'*
consistent-CNot-not tot-CNot total-over-m-def true-clss-clss-def)
 hence $\neg ?I \models CC$ using $\langle ?I \models_s CNot\ CC \rangle$ *cons'* *consistent-CNot-not* **by** *blast*
ultimately have ?*I* $\models \{\#L\#\}$ **by** *blast*
thus *I* $\models \{\#L\#\}$
 by (*metis* (*no-types*, *lifting*) *atms-of-ms-union cons'* *consistent-CNot-not tot total-not-CNot*
total-over-m-def total-over-set-union true-clss-union-increase)
qed

lemma *true-annots-CNot-lit-of-notin-skip*:

assumes *LM*: $L \# M \models_{as} CNot\ A$ **and** *LA*: $lit\ of\ L \notin \# A \rightarrow lit\ of\ L \notin \# A$
 shows $M \models_{as} CNot\ A$
 using *LM unfolding true-annots-def Ball-def*

proof (*intro allI impI*)

fix *l*
 assume *H*: $\forall x. x \in CNot\ A \rightarrow L \# M \models_a x$ **and** *l*: $l \in CNot\ A$
 hence $L \# M \models_a l$ **by** *auto*
 thus $M \models_a l$ using *LA l* **by** (*cases L*) (*auto simp add: CNot-def*)
qed

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:

$A \cup \{B\} \models_{ps} \{\{\#\}\} \longleftrightarrow A \models_{ps} CNot\ B$
 using *total-not-CNot consistent-CNot-not* **unfolding** *total-over-m-def true-clss-clss-def*
by *fastforce*

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

assumes $a \# M' \models_a D$
 and *atm-of* (*lit-of* *a*) \notin *atms-of* *D*
 shows $M' \models_a D$
 using *assms true-clss-remove-hd-if-notin-vars* **unfolding** *true-annot-def* **by** *auto*

lemma *true-annot-remove-if-notin-vars*:

assumes $M @ M' \models_a D$
 and $\forall x \in atms\ of\ D. x \notin atm\ of\ 'lits\ of\ M$
 shows $M' \models_a D$
 using *assms* **apply** (*induct M, simp*)
 using *true-annot-remove-hd-if-notin-vars* **by** *force+*

lemma *true-annots-remove-if-notin-vars*:

assumes $M @ M' \models_{as} D$
 and $\forall x \in atms\ of\ ms\ D. x \notin atm\ of\ 'lits\ of\ M$
 shows $M' \models_{as} D$ **unfolding** *true-annots-def*
 using *assms true-annot-remove-if-notin-vars[of M M']*
unfolding *true-annots-def atms-of-ms-def* **by** *force*

lemma *all-variables-defined-not-imply-cnot*:

assumes $\forall s \in atms\ of\ ms\ \{B\}. s \in atm\ of\ 'lits\ of\ A$
 and $\neg A \models_a B$

shows $A \models_{as} CNot\ B$
unfolding *true-annot-def true-annots-def Ball-def CNot-def true-lit-def*
proof (*clarify, rule ccontr*)
fix L
assume $LB: L \in \# B$ **and** $\neg \text{ lits-of } A \models l - L$
hence $\text{ atm-of } L \in \text{ atm-of } ' \text{ lits-of } A$
using *assms(1)* **by** (*simp add: atm-of-lit-in-atms-of lits-of-def*)
hence $L \in \text{ lits-of } A \vee \neg L \in \text{ lits-of } A$
using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set* **by** *metis*
hence $L \in \text{ lits-of } A$ **using** $\langle \neg \text{ lits-of } A \models l - L \rangle$ **by** *auto*
thus *False*
using LB *assms(2)* **unfolding** *true-annot-def true-lit-def true-cls-def Bex-mset-def*
by *blast*
qed

lemma *CNot-union-mset[simp]:*
 $CNot\ (A \# \cup B) = CNot\ A \cup CNot\ B$
unfolding *CNot-def* **by** *auto*

13.5 Other

abbreviation $\text{no-dup } L \equiv \text{distinct } (\text{map } (\lambda l. \text{ atm-of } (\text{lit-of } l))\ L)$

lemma *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:*
assumes $\text{no-dup } M$
shows $\text{length } M = \text{card } (\text{atm-of } ' \text{ lits-of } M)$
using *assms* **unfolding** *lits-of-def* **by** (*induct M*) (*auto simp add: image-image*)

lemma *distinctconsistent-interp:*
 $\text{no-dup } M \implies \text{consistent-interp } (\text{lits-of } M)$
proof (*induct M*)
case *Nil*
show *?case* **by** *auto*
next
case (*Cons L M*)
hence *a1: consistent-interp (lits-of M)* **by** *auto*
have *a2: atm-of (lit-of L) \notin ($\lambda l. \text{ atm-of } (\text{lit-of } l)$) ' 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*
thus *?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 $\text{no-dup } M$
shows $\text{no-dup } (a @ b)$
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 $\forall l \in \# A. -l \in lits-of\ (L \# M)$
using *assms(1) in-CNot-implies-uminus(2)* **by** *blast*
moreover
have $atm-of\ (lit-of\ L) \notin atm-of\ 'lits-of\ M$
using *assms(3) unfolding lits-of-def* **by** *force*
hence $- lit-of\ L \notin lits-of\ M$ **unfolding** *lits-of-def*
by (*metis (no-types) atm-of-uminus imageI*)
ultimately have $\forall l \in \# A. -l \in lits-of\ M$
using *assms(2) unfolding Ball-mset-def* **by** (*metis insertE lits-of-cons uminus-of-uminus-id*)
thus *?thesis* **by** (*auto simp add: true-annots-def*)
qed

type-synonym *'v clauses = 'v clause multiset*

abbreviation *true-annots-mset* (**infix** \models_{asm} 50) **where**
 $I \models_{asm} C \equiv I \models_{as} (set-mset\ C)$

abbreviation *true-clss-clss-m:: 'a clauses \Rightarrow 'a clauses \Rightarrow bool* (**infix** \models_{psm} 50) **where**
 $I \models_{psm} C \equiv set-mset\ I \models_{ps} (set-mset\ C)$

Analog of $\llbracket ?N \models_{ps} ?B; ?A \subseteq ?B \rrbracket \Longrightarrow ?N \models_{ps} ?A$

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**
 $I \models_{pm} C \equiv set-mset\ I \models_p C$

abbreviation *distinct-mset-mset :: 'a multiset multiset \Rightarrow bool* **where**
 $distinct-mset-mset\ \Sigma \equiv distinct-mset-set\ (set-mset\ \Sigma)$

abbreviation *all-decomposition-implies-m* **where**
 $all-decomposition-implies-m\ A\ B \equiv all-decomposition-implies\ (set-mset\ A)\ B$

abbreviation *atms-of-msu* **where**
 $atms-of-msu\ U \equiv atms-of-ms\ (set-mset\ U)$

abbreviation *true-clss-m:: 'a interp \Rightarrow 'a clauses \Rightarrow bool* (**infix** \models_{sm} 50) **where**
 $I \models_{sm} C \equiv I \models_s set-mset\ C$

abbreviation *true-clss-ext-m* (**infix** \models_{sextm} 49) **where**
 $I \models_{sextm} C \equiv I \models_{sext} set-mset\ C$

end

theory *CDCL-NOT*

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

14 NOT's CDCL

sledgehammer-params[*verbose, prover=e spass z3 cvc4 verit remote-vampire*]

declare *set-mset-minus-replicate-mset*[*simp*]

14.1 Auxiliary Lemmas and Measure

lemma *no-dup-cannot-not-lit-and-uminus*:

$no_dup\ M \implies -\ lit_of\ xa = lit_of\ x \implies x \in set\ M \implies xa \notin set\ M$
by (*metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id*)

lemma *true-clss-single-iff-incl*:

$I \models_s single\ 'B \longleftrightarrow B \subseteq I$
unfolding *true-clss-def* **by** *auto*

lemma *atms-of-ms-single-atm-of[simp]*:

$atms_of_ms\ \{\{\#lit_of\ L\# \mid L.\ P\ L\} = atm_of\ ' \{\ lit_of\ L \mid L.\ P\ L\}$
unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-uminus-lit-atm-of-lit-of*:

$atms_of\ \{\# - lit_of\ x.\ x \in \# A\# \} = atm_of\ ' (lit_of\ ' (set_mset\ A))$
unfolding *atms-of-def* **by** (*auto simp add: Fun.image-comp*)

lemma *atms-of-ms-single-image-atm-of-lit-of*:

$atms_of_ms\ ((\lambda x.\ \{\#lit_of\ x\# \})\ 'A) = atm_of\ ' (lit_of\ 'A)$
unfolding *atms-of-ms-def* **by** *auto*

This measure can also be seen as the increasing lexicographic order: it is an order on bounded sequences, when each element is bounded. The proof involves a measure like the one defined here (the same?).

definition $\mu_C :: nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow nat$ **where**

$\mu_C\ s\ b\ M \equiv (\sum i=0..<length\ M.\ M!i * b^\wedge (s + i - length\ M))$

lemma $\mu_C\ nil[simp]$:

$\mu_C\ s\ b\ [] = 0$
unfolding $\mu_C\text{-def}$ **by** *auto*

lemma $\mu_C\ single[simp]$:

$\mu_C\ s\ b\ [L] = L * b^\wedge (s - Suc\ 0)$
unfolding $\mu_C\text{-def}$ **by** *auto*

lemma *set-sum-atLeastLessThan-add*:

$(\sum i=k..<k+(b::nat).\ f\ i) = (\sum i=0..<b.\ f\ (k + i))$
by (*induction b*) *auto*

lemma *set-sum-atLeastLessThan-Suc*:

$(\sum i=1..<Suc\ j.\ f\ i) = (\sum i=0..<j.\ f\ (Suc\ i))$
using *set-sum-atLeastLessThan-add[of - 1 j]* **by** *force*

lemma $\mu_C\ cons$:

$\mu_C\ s\ b\ (L \# M) = L * b^\wedge (s - 1 - length\ M) + \mu_C\ s\ b\ M$

proof –

have $\mu_C\ s\ b\ (L \# M) = (\sum i=0..<length\ (L\#M).\ (L\#M)!i * b^\wedge (s + i - length\ (L\#M)))$
unfolding $\mu_C\text{-def}$ **by** *blast*

also have $\dots = (\sum i=0..<1.\ (L\#M)!i * b^\wedge (s + i - length\ (L\#M)))$
 $+ (\sum i=1..<length\ (L\#M).\ (L\#M)!i * b^\wedge (s + i - length\ (L\#M)))$

by (*rule setsum-add-nat-ivl[symmetric]*) *simp-all*

finally have $\mu_C\ s\ b\ (L \# M) = L * b^\wedge (s - 1 - length\ M)$
 $+ (\sum i=1..<length\ (L\#M).\ (L\#M)!i * b^\wedge (s + i - length\ (L\#M)))$

by *auto*

moreover {

have $(\sum_{i=1..<\text{length } (L\#M)}. (L\#M)!i * b^\wedge (s+i - \text{length } (L\#M))) =$
 $(\sum_{i=0..<\text{length } (M)}. (L\#M)!(\text{Suc } i) * b^\wedge (s + (\text{Suc } i) - \text{length } (L\#M)))$
 unfolding *length-Cons set-sum-atLeastLessThan-Suc* by *blast*
 also have ... = $(\sum_{i=0..<\text{length } (M)}. M!i * b^\wedge (s + i - \text{length } M))$
 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 ... = $(\sum_{i=0..<\text{length } M}. (M\@M')!i * b^\wedge (s+i - \text{length } (M\@M')))$
 $+ (\sum_{i=\text{length } M..<\text{length } (M\@M')}. (M\@M')!i * b^\wedge (s+i - \text{length } (M\@M')))$
 by (*auto intro!: setsum-add-nat-ivl[symmetric]*)
 moreover
 have $\forall i \in \{0..<\text{length } M\}. (M\@M')!i * b^\wedge (s+i - \text{length } (M\@M')) = M!i * b^\wedge (s - \text{length } M' + i - \text{length } M)$
 using $\langle s \geq \text{length } (M\@M') \rangle$ by (*auto simp add: nth-append ac-simps*)
 then have $\mu_C (s - \text{length } M') b M = (\sum_{i=0..<\text{length } M}. (M\@M')!i * b^\wedge (s+i - \text{length } (M\@M')))$
 unfolding $\mu_C\text{-def}$ 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 *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 (*cases M*) (*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 (*cases k = 0*)
 apply (*cases n; simp*)
 by (*induct n*) (*auto simp: Nat.nat-distrib*)

In the degenerated cases, we only have the large inequality holds. In the other cases, the following strict inequality holds:

lemma $\mu_C\text{-bounded-non-degenerated}$:

```

fixes  $b :: \text{nat}$ 
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  $(\text{metis Nat.add-diff-assoc2 add.commute assms(4) mult.assoc power-add})$ 
      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  $(\text{auto simp add: ac-simps})$ 
      also have  $\dots = (\sum i=0..<\text{length } M. b^\wedge i) * b^\wedge (s - \text{length } M) * (b-1)$ 
        by  $(\text{simp add: setsum-left-distrib setsum-right-distrib ac-simps})$ 
      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  $(\text{simp add: ac-simps})$ 

    also
      have  $(\sum i=0..<\text{length } M. b^\wedge i) * (b-1) = b^\wedge (\text{length } M) - 1$ 
        using  $\text{sum-of-powers}[of \ b \ \text{length } M] \langle b > 1 \rangle$ 
        by  $(\text{auto simp add: ac-simps})$ 
      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  $(\text{metis assms(4) le-add-diff-inverse power-add})$ 
      finally show  $?thesis$  unfolding  $\mu_C\text{-def}$  by  $(\text{auto simp add: ac-simps})$ 
  qed
qed

```

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

lemma $\mu_C\text{-bounded}$:

```

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 ( $\text{cases } b, \text{cases } M$ ) 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  $\mu_C\text{-bounded-non-degenerated}[OF\ M\ \text{assms}(1,2)]$  by arith
    qed
qed

```

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

lemma $\mu_C\text{-base-0}$:

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  $(\sum_{i=0..<n}. M!i * (0::nat)^\wedge i) \leq M!0$ 
      apply (induction n rule: nat-induct)
      by simp (case-tac n, auto)
  }
  ultimately have  $?thesis$  unfolding  $\mu_C\text{-def}$  by auto
}
moreover
{
  assume  $\text{length } M < s$ 
  then have  $\mu_C\ s\ 0\ M = 0$  unfolding  $\mu_C\text{-def}$  by auto
  ultimately show  $?thesis$  using assms unfolding  $\mu_C\text{-def}$  by linarith
}
qed

```

14.2 Initial definitions

14.2.1 The state

We define here an abstraction over operation on the state we are manipulating.

```

locale  $dpll\text{-state} =$ 
  fixes
     $\text{trail} :: 'st \Rightarrow ('v, \text{unit}, \text{unit}) \text{ marked-lits}$  and
     $\text{clauses} :: 'st \Rightarrow 'v \text{ clauses}$  and
     $\text{prepend-trail} :: ('v, \text{unit}, \text{unit}) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st$  and
     $\text{tl-trail} :: 'st \Rightarrow 'st$  and
     $\text{add-cl}_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$  and
     $\text{remove-cl}_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ 
  assumes
     $\text{trail-prepend-trail}[simp]:$ 
       $\bigwedge st\ L. \text{undefined-lit } (\text{trail } st) (\text{lit-of } L) \implies \text{trail } (\text{prepend-trail } L\ st) = L \# \text{trail } st$ 
    and
     $\text{tl-trail}[simp]: \text{trail } (\text{tl-trail } S) = \text{tl } (\text{trail } S)$  and
     $\text{trail-add-cl}_{NOT}[simp]: \bigwedge st\ C. \text{no-dup } (\text{trail } st) \implies \text{trail } (\text{add-cl}_{NOT}\ C\ st) = \text{trail } st$  and
     $\text{trail-remove-cl}_{NOT}[simp]: \bigwedge st\ C. \text{trail } (\text{remove-cl}_{NOT}\ C\ st) = \text{trail } st$  and

```

clauses-prepend-trail[simp]:
 $\bigwedge st L. \text{undefined-lit } (\text{trail } st) (\text{lit-of } L) \implies \text{clauses } (\text{prepend-trail } L \ st) = \text{clauses } st$
and
clauses-tl-trail[simp]: $\bigwedge st. \text{clauses } (\text{tl-trail } st) = \text{clauses } st$ **and**
clauses-add-cls_{NOT}[simp]:
 $\bigwedge st C. \text{no-dup } (\text{trail } st) \implies \text{clauses } (\text{add-cls}_{NOT} \ C \ st) = \{\#C\# \} + \text{clauses } st$ **and**
clauses-remove-cls_{NOT}[simp]: $\bigwedge st C. \text{clauses } (\text{remove-cls}_{NOT} \ C \ st) = \text{remove-mset } C \ (\text{clauses } st)$
begin

function *reduce-trail-to_{NOT}* :: 'a list \Rightarrow 'st \Rightarrow 'st **where**
reduce-trail-to_{NOT} *F S* =
 (if length (trail *S*) = length *F* \vee trail *S* = [] then *S* else *reduce-trail-to_{NOT}* *F* (tl-trail *S*))
by fast+

termination by (relation measure ($\lambda(-, S). \text{length } (\text{trail } S)$)) *auto*
declare *reduce-trail-to_{NOT}.simps*[simp del]

lemma
shows
reduce-trail-to_{NOT}-nil[simp]: trail *S* = [] \implies *reduce-trail-to_{NOT}* *F S* = *S* **and**
reduce-trail-to_{NOT}-eq-length[simp]: length (trail *S*) = length *F* \implies *reduce-trail-to_{NOT}* *F S* = *S*
by (auto simp: *reduce-trail-to_{NOT}.simps*)

lemma *reduce-trail-to_{NOT}-length-ne*[simp]:
 length (trail *S*) \neq length *F* \implies trail *S* \neq [] \implies
reduce-trail-to_{NOT} *F S* = *reduce-trail-to_{NOT}* *F* (tl-trail *S*)
by (auto simp: *reduce-trail-to_{NOT}.simps*)

lemma *trail-reduce-trail-to_{NOT}-length-le*:
assumes length *F* > length (trail *S*)
shows trail (*reduce-trail-to_{NOT}* *F S*) = []
using *assms* **by** (induction *F S* rule: *reduce-trail-to_{NOT}.induct*)
 (simp add: *less-imp-diff-less* *reduce-trail-to_{NOT}.simps*)

lemma *trail-reduce-trail-to_{NOT}-nil*[simp]:
 trail (*reduce-trail-to_{NOT}* [] *S*) = []
by (induction [] *S* rule: *reduce-trail-to_{NOT}.induct*)
 (simp add: *less-imp-diff-less* *reduce-trail-to_{NOT}.simps*)

lemma *clauses-reduce-trail-to_{NOT}-nil*:
 clauses (*reduce-trail-to_{NOT}* [] *S*) = clauses *S*
by (induction [] *S* rule: *reduce-trail-to_{NOT}.induct*)
 (simp add: *less-imp-diff-less* *reduce-trail-to_{NOT}.simps*)

lemma *trail-reduce-trail-to_{NOT}-drop*:
 trail (*reduce-trail-to_{NOT}* *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-to_{NOT}.induct*)
apply (rename-tac *F S*)
apply (case-tac trail *S*)
apply auto[]
apply (rename-tac *list*)
apply (case-tac *Suc* (length *list*) > length *F*)

prefer 2 **apply** *simp*
apply (*subgoal-tac* *Suc* (*length list*) - *length F* = *Suc* (*length list* - *length F*))
apply *simp*
apply *simp*
done

lemma *reduce-trail-to_{NOT}-skip-beginning*:
assumes *trail S = F' @ F*
shows *trail (reduce-trail-to_{NOT} F S) = F*
using *assms* **by** (*auto simp: trail-reduce-trail-to_{NOT}-drop*)

lemma *reduce-trail-to_{NOT}-clauses[*simp*]*:
clauses (reduce-trail-to_{NOT} F S) = clauses S
by (*induction F S rule: reduce-trail-to_{NOT}.induct*)
(simp add: less-imp-diff-less reduce-trail-to_{NOT}.simps)

abbreviation *trail-weight* **where**
trail-weight S \equiv *map (($\lambda l. 1 + \text{length } l$) o *snd*) (get-all-marked-decomposition (trail S))*

definition *state-eq_{NOT}* :: '*st* \Rightarrow '*st* \Rightarrow bool (**infix** \sim 50) **where**
S \sim *T* \longleftrightarrow *trail S = trail T* \wedge *clauses S = clauses T*

lemma *state-eq_{NOT}-ref[*simp*]*:
S \sim *S*
unfolding *state-eq_{NOT}-def* **by** *auto*

lemma *state-eq_{NOT}-sym*:
S \sim *T* \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
state-eq_{NOT}-trail: *S* \sim *T* \Longrightarrow *trail S = trail T* **and**
state-eq_{NOT}-clauses: *S* \sim *T* \Longrightarrow *clauses S = clauses T*
unfolding *state-eq_{NOT}-def* **by** *auto*

lemmas *state-simp_{NOT}[*simp*]* = *state-eq_{NOT}-trail state-eq_{NOT}-clauses*

lemma *trail-eq-reduce-trail-to_{NOT}-eq*:
trail S = trail T \Longrightarrow *trail (reduce-trail-to_{NOT} F S) = trail (reduce-trail-to_{NOT} F T)*
apply (*induction F S arbitrary: T rule: reduce-trail-to_{NOT}.induct*)
by (*metis tl-trail reduce-trail-to_{NOT}-eq-length reduce-trail-to_{NOT}-length-ne reduce-trail-to_{NOT}-nil*)

lemma *reduce-trail-to_{NOT}-state-eq_{NOT}-compatible*:
assumes *ST: S* \sim *T*
shows *reduce-trail-to_{NOT} F S* \sim *reduce-trail-to_{NOT} F T*
proof –
have *clauses (reduce-trail-to_{NOT} F S) = clauses (reduce-trail-to_{NOT} F T)*
using *ST* **by** *auto*
moreover have *trail (reduce-trail-to_{NOT} F S) = trail (reduce-trail-to_{NOT} F T)*

using *trail-eq-reduce-trail-to_{NOT}-eq*[of *S T F*] *ST* **by** *auto*
ultimately show *?thesis* **by** (*auto simp del: state-simp_{NOT} simp: state-eq_{NOT}-def*)
qed

lemma *trail-reduce-trail-to_{NOT}-add-cls_{NOT}*[*simp*]:
no-dup (*trail S*) \implies
trail (*reduce-trail-to_{NOT} F* (*add-cls_{NOT} C S*)) = *trail* (*reduce-trail-to_{NOT} F S*)
by (*rule trail-eq-reduce-trail-to_{NOT}-eq*) *simp*

lemma *reduce-trail-to_{NOT}-trail-tl-trail-decomp*[*simp*]:
trail S = *F' @ Marked K () # F* \implies
trail (*reduce-trail-to_{NOT} F* (*tl-trail S*)) = *F*
apply (*rule reduce-trail-to_{NOT}-skip-beginning*[of - *tl* (*F' @ Marked K () # []*)])
by (*cases F'*) (*auto simp add:tl-append reduce-trail-to_{NOT}-skip-beginning*)

end

14.2.2 Definition of the operation

locale *propagate-ops* =
dpll-state trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT} **for**
trail :: '*st* \Rightarrow ('*v*, *unit*, *unit*) marked-lits **and**
clauses :: '*st* \Rightarrow '*v* clauses **and**
prepend-trail :: ('*v*, *unit*, *unit*) marked-lit \Rightarrow '*st* \Rightarrow '*st* **and**
tl-trail :: '*st* \Rightarrow '*st* **and**
add-cls_{NOT} remove-cls_{NOT}:: '*v* clause \Rightarrow '*st* \Rightarrow '*st* **and**
propagate-cond :: ('*v*, *unit*, *unit*) marked-lit \Rightarrow '*st* \Rightarrow *bool*
begin
inductive *propagate_{NOT}* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**
propagate_{NOT}[*intro*]: *C* + {*#L#*} \in *# clauses S* \implies *trail S* \models_{as} *CNot C*
 \implies *undefined-lit* (*trail S*) *L*
 \implies *propagate-cond* (*Propagated L* ()) *S*
 \implies *T* \sim *prepend-trail* (*Propagated L* ()) *S*
 \implies *propagate_{NOT} S T*
inductive-cases *propagateE*[*elim*]: *propagate_{NOT} S T*
end

locale *decide-ops* =
dpll-state trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT} **for**
trail :: '*st* \Rightarrow ('*v*, *unit*, *unit*) marked-lits **and**
clauses :: '*st* \Rightarrow '*v* clauses **and**
prepend-trail :: ('*v*, *unit*, *unit*) marked-lit \Rightarrow '*st* \Rightarrow '*st* **and**
tl-trail :: '*st* \Rightarrow '*st* **and**
add-cls_{NOT} remove-cls_{NOT}:: '*v* clause \Rightarrow '*st* \Rightarrow '*st*
begin
inductive *decide_{NOT}* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**
decide_{NOT}[*intro*]: *undefined-lit* (*trail S*) *L* \implies *atm-of L* \in *atms-of-msu* (*clauses S*)
 \implies *T* \sim *prepend-trail* (*Marked L* ()) *S*
 \implies *decide_{NOT} S T*
inductive-cases *decideE*[*elim*]: *decide_{NOT} S S'*
end

locale *backjumping-ops* =
dpll-state trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT}

```

for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive backjump where
  trail S = F' @ Marked K () # F
   $\Rightarrow$  T ~ prepend-trail (Propagated L ()) (reduce-trail-toNOT F S)
   $\Rightarrow$  C  $\in$  # clauses S
   $\Rightarrow$  trail S  $\models_{as}$  CNot C
   $\Rightarrow$  undefined-lit F L
   $\Rightarrow$  atm-of L  $\in$  atms-of-msu (clauses S)  $\cup$  atm-of ' (lits-of (trail S))
   $\Rightarrow$  clauses S  $\models_{pm}$  C' + {#L#}
   $\Rightarrow$  F  $\models_{as}$  CNot C'
   $\Rightarrow$  backjump-conds C C' L S T
   $\Rightarrow$  backjump S T
inductive-cases backjumpE: backjump S T
end

```

14.3 DPLL with backjumping

```

locale dpll-with-backjumping-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clNOT remove-clNOT +
  propagate-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds +
  decide-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT +
  backjumping-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT backjump-conds
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool +
assumes
  bj-can-jump:
   $\bigwedge S C F' K F L.$ 
  inv S  $\Rightarrow$ 
  no-dup (trail S)  $\Rightarrow$ 
  trail S = F' @ Marked K () # F  $\Rightarrow$ 
  C  $\in$  # clauses S  $\Rightarrow$ 
  trail S  $\models_{as}$  CNot C  $\Rightarrow$ 
  undefined-lit F L  $\Rightarrow$ 
  atm-of L  $\in$  atms-of-msu (clauses S)  $\cup$  atm-of ' (lits-of (F' @ Marked K () # F))  $\Rightarrow$ 
  clauses S  $\models_{pm}$  C' + {#L#}  $\Rightarrow$ 
  F  $\models_{as}$  CNot C'  $\Rightarrow$ 
   $\neg$ no-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 $atm\text{-}of\ L \in atm\text{-}of\ ' \textit{lits-of}\ (F' @ \textit{Marked}\ K\ () \# F)$ is important, otherwise you are not sure that you can backtrack.

14.3.1 Definition

We define $dpll$ with backjumping:

inductive $dpll\text{-}bj :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**
 $bj\text{-}decide_{NOT}:$ $decide_{NOT}\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S' \mid$
 $bj\text{-}propagate_{NOT}:$ $propagate_{NOT}\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S' \mid$
 $bj\text{-}backjump:$ $backjump\ S\ S' \Longrightarrow dpll\text{-}bj\ S\ S'$

lemmas $dpll\text{-}bj\text{-}induct = dpll\text{-}bj.induct[split\text{-}format(complete)]$

thm $dpll\text{-}bj\text{-}induct[OF\ dpll\text{-}with\text{-}backjumping\text{-}ops\text{-}axioms]$

lemma $dpll\text{-}bj\text{-}all\text{-}induct[consumes\ 2, case\text{-}names\ decide_{NOT}\ propagate_{NOT}\ backjump]:$

fixes $S\ T :: 'st$

assumes

$dpll\text{-}bj\ S\ T$ **and**

$inv\ S$

$\bigwedge L\ T. \textit{undefined-lit}\ (trail\ S)\ L \Longrightarrow atm\text{-}of\ L \in atm\text{-}of\text{-}msu\ (clauses\ S)$

$\Longrightarrow T \sim \textit{prepend-trail}\ (\textit{Marked}\ L\ ())\ S$

$\Longrightarrow P\ S\ T$ **and**

$\bigwedge C\ L\ T. C + \{\#L\#\} \in \# \textit{clauses}\ S \Longrightarrow trail\ S \models_{as} CNot\ C \Longrightarrow \textit{undefined-lit}\ (trail\ S)\ L$

$\Longrightarrow T \sim \textit{prepend-trail}\ (\textit{Propagated}\ L\ ())\ S$

$\Longrightarrow P\ S\ T$ **and**

$\bigwedge C\ F'\ K\ F\ L\ C'\ T. C \in \# \textit{clauses}\ S \Longrightarrow F' @ \textit{Marked}\ K\ () \# F \models_{as} CNot\ C$

$\Longrightarrow trail\ S = F' @ \textit{Marked}\ K\ () \# F$

$\Longrightarrow \textit{undefined-lit}\ F\ L$

$\Longrightarrow atm\text{-}of\ L \in atm\text{-}of\text{-}msu\ (clauses\ S) \cup atm\text{-}of\ ' \textit{lits-of}\ (F' @ \textit{Marked}\ K\ () \# F)$

$\Longrightarrow clauses\ S \models_{pm} C' + \{\#L\#\}$

$\Longrightarrow F \models_{as} CNot\ C'$

$\Longrightarrow T \sim \textit{prepend-trail}\ (\textit{Propagated}\ L\ ())\ (\textit{reduce-trail-to}_{NOT}\ F\ S)$

$\Longrightarrow P\ S\ T$

shows $P\ S\ T$

apply ($induct\ T\ rule: dpll\text{-}bj\text{-}induct[OF\ local.dpll\text{-}with\text{-}backjumping\text{-}ops\text{-}axioms]$)

apply ($rule\ assms(1)$)

using $assms(3)$ **apply** $blast$

apply ($elim\ propagateE$) **using** $assms(4)$ **apply** $blast$

apply ($elim\ backjumpE$) **using** $assms(5)$ $\langle inv\ S \rangle$ **by** $simp$

14.3.2 Basic properties

First, some better suited induction principle **lemma** $dpll\text{-}bj\text{-}clauses:$

assumes $dpll\text{-}bj\ S\ T$ **and** $inv\ S$

shows $clauses\ S = clauses\ T$

using $assms$ **by** ($induction\ rule: dpll\text{-}bj\text{-}all\text{-}induct$) $auto$

No duplicates in the trail **lemma** $dpll\text{-}bj\text{-}no\text{-}dup:$

assumes $dpll\text{-}bj\ S\ T$ **and** $inv\ S$

and $no\text{-}dup\ (trail\ S)$

shows $no\text{-}dup\ (trail\ T)$

using $assms$ **by** ($induction\ rule: dpll\text{-}bj\text{-}all\text{-}induct$)

($auto\ simp\ add: \textit{defined-lit-map}\ \textit{reduce-trail-to}_{NOT}\text{-}skip\text{-}beginning$)

Valuations **lemma** $dpll\text{-}bj\text{-}sat\text{-}iff:$

assumes *dpll-bj S T and inv S*
shows $I \models_{sm} \text{clauses } S \longleftrightarrow I \models_{sm} \text{clauses } T$
using *assms by (induction rule: 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-msu (clauses S) = atms-of-msu (clauses T)*
using *assms by (induction rule: dpll-bj-all-induct) auto*

lemma *dpll-bj-atms-in-trail:*

assumes
dpll-bj S T and
inv S and
atm-of ' (lits-of (trail S)) \subseteq atms-of-msu (clauses S)
shows *atm-of ' (lits-of (trail T)) \subseteq atms-of-msu (clauses S)*
using *assms by (induction rule: dpll-bj-all-induct)*
(auto simp: in-plus-implies-atm-of-on-atms-of-ms reduce-trail-to_{NOT}-skip-beginning)

lemma *dpll-bj-atms-in-trail-in-set:*

assumes *dpll-bj S T and*
inv S and
atms-of-msu (clauses S) \subseteq A and
atm-of ' (lits-of (trail S)) \subseteq A
shows *atm-of ' (lits-of (trail T)) \subseteq A*
using *assms by (induction rule: dpll-bj-all-induct)*
(auto simp: in-plus-implies-atm-of-on-atms-of-ms)

lemma *dpll-bj-all-decomposition-implies-inv:*

assumes
dpll-bj S T and
inv: inv S and
decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
shows *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*
using *assms(1,2)*

proof *(induction rule: dpll-bj-all-induct)*

case *decide_{NOT}*

then show *?case using decomp by auto*

next

case *(propagate_{NOT} C L T) note propa = this(1) and undef = this(3) and T = this(4)*

let *?M' = trail (prepend-trail (Propagated L ()) S)*

let *?N = clauses S*

obtain *a y l where ay: get-all-marked-decomposition ?M' = (a, y) # l*

by *(cases get-all-marked-decomposition ?M') fastforce+*

then have *M': ?M' = y @ a using get-all-marked-decomposition-decomp[of ?M'] by auto*

have *M: get-all-marked-decomposition (trail S) = (a, tl y) # l*

using *ay undef by (cases get-all-marked-decomposition (trail S)) auto*

have *y₀: y = (Propagated L ()) # (tl y)*

using *ay undef by (auto simp add: M)*

from *arg-cong[OF this, of set] have y[simp]: set y = insert (Propagated L ()) (set (tl y))*

by *simp*

have *tr-S: trail S = tl y @ a*

using *arg-cong[OF M', of tl] y₀ M get-all-marked-decomposition-decomp by force*

have *a-Un-N-M: ($\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup \text{set-mset } ?N \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } (tl y)$*

```

using decomp ay unfolding all-decomposition-implies-def by (simp add: M)+

moreover have ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ set  $a \cup \text{set-mset } ?N \models_p \{\#L\# \}$  (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.hyps(2) tr-S true-annots-true-clss-clss
    by (force simp add: image-Un sup-commute)
  have a2: set-mset (clauses S)  $\cup (\lambda a. \{\#lit\text{-of } a\# \})$  ‘ set  $a$ 
     $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ‘ set (tl y)
    using calculation by (auto simp add: sup-commute)
  show ( $\lambda m. \{\#lit\text{-of } m\# \}$ ) ‘ set  $a \cup \text{set-mset } (\text{clauses } S) \models_{ps} CNot\ C$ 
    proof –
      have set-mset (clauses S)  $\cup (\lambda m. \{\#lit\text{-of } m\# \})$  ‘ set  $a \models_{ps}$ 
        ( $\lambda m. \{\#lit\text{-of } m\# \})$  ‘ set  $a \cup (\lambda m. \{\#lit\text{-of } m\# \})$  ‘ set (tl y)
        using a2 true-clss-clss-def by blast
      then show ( $\lambda m. \{\#lit\text{-of } m\# \}$ ) ‘ set  $a \cup \text{set-mset } (\text{clauses } S) \models_{ps} CNot\ C$ 
        using a1 unfolding sup-commute by (meson true-clss-clss-left-right
          true-clss-clss-union-and true-clss-clss-union-l-r )
    qed
  qed

ultimately have ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ set  $a \cup \text{set-mset } ?N \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ‘ set  $?M'$ 
unfolding M' by (auto simp add: all-in-true-clss-clss image-Un)

then show ?case
  using decomp T M undef unfolding ay all-decomposition-implies-def by (auto simp add: ay)
next
case (backjump C F' K F L D T) note confl = this(2) and tr = this(3) and undef = this(4)
and L = this(5) and N-C = this(6) and vars-D = this(5) and T = this(8)
have decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition F)
using decomp unfolding tr all-decomposition-implies-def
by (metis (no-types, lifting) get-all-marked-decomposition.simps(1)
  get-all-marked-decomposition-never-empty hd-Cons-tl insert-iff list.sel(3) list.set(2)
  tl-get-all-marked-decomposition-skip-some)

moreover have ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ set (fst (hd (get-all-marked-decomposition F)))
   $\cup \text{set-mset } (\text{clauses } S)$ 
 $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ‘ set (snd (hd (get-all-marked-decomposition F)))
by (metis all-decomposition-implies-cons-single decomp get-all-marked-decomposition-never-empty
  hd-Cons-tl)
moreover
have vars-of-D: atms-of D  $\subseteq$  atm-of ‘ lits-of F
using (F  $\models_{as} CNot\ D$ ) unfolding atms-of-def
by (meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)

obtain a b li where F: get-all-marked-decomposition F = (a, b) # li
by (cases get-all-marked-decomposition F) auto
have F = b @ a
using get-all-marked-decomposition-decomp[of F a b] F by auto
have a-N-b: ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ‘ set  $a \cup \text{set-mset } (\text{clauses } S) \models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ‘ set  $b$ 

```

```

using decomp unfolding all-decomposition-implies-def by (auto simp add: F)

have F-D: $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } F \models_{ps} CNot D$ 
  using  $\langle F \models_{as} CNot D \rangle$  by (simp add: true-annots-true-clss-clss)
then have  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } b \models_{ps} CNot D$ 
  unfolding  $\langle F = b @ a \rangle$  by (simp add: image-Un sup.commute)
have a-N-CNot-D: $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup \text{set-mset } (clauses S) \models_{ps} CNot D \cup (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } b$ 
  apply (rule true-clss-clss-left-right)
  using a-N-b F-D unfolding  $\langle F = b @ a \rangle$  by (auto simp add: image-Un ac-simps)

have a-N-D-L: $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup \text{set-mset } (clauses S) \models_p D + \{\#L\# \}$ 
  by (simp add: N-C)
have  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } a \cup \text{set-mset } (clauses S) \models_p \{\#L\# \}$ 
  using a-N-D-L a-N-CNot-D by (blast intro: true-clss-clss-plus-CNot)
then show ?case
  using decomp T tr undef unfolding all-decomposition-implies-def by (auto simp add: F)
qed

```

14.3.3 Termination

Using a proper measure **lemma** *length-get-all-marked-decomposition-append-Marked*:
 $length (get\text{-all-marked-decomposition } (F' @ Marked K () \# F)) =$
 $length (get\text{-all-marked-decomposition } F')$
 $+ length (get\text{-all-marked-decomposition } (Marked K () \# F))$
 $- 1$
by (*induction F' rule: marked-lit-list-induct*) *auto*

lemma *take-length-get-all-marked-decomposition-marked-sandwich*:
 $take (length (get\text{-all-marked-decomposition } F))$
 $(map (f o snd) (rev (get\text{-all-marked-decomposition } (F' @ Marked K () \# F))))$
 $=$
 $map (f o snd) (rev (get\text{-all-marked-decomposition } F))$

```

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':  $get\text{-all-marked-decomposition } (F' @ Marked K () \# F) = (a, b) \# l$ 
    by (cases get-all-marked-decomposition (F' @ Marked K () \# F)) auto
  have  $length (get\text{-all-marked-decomposition } F) - length l = 0$ 
    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*:
 $length (get\text{-all-marked-decomposition } M) \leq 1 + length M$
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**

$NA: \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

$MA: \text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d}: \text{no-dup } (\text{trail } S)$ **and**

$\text{finite}: \text{finite } A$

shows $\mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$

$> \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$

using *assms(1,2)*

proof (*induction rule: dpll-bj-all-induct*)

case ($\text{propagate}_{NOT} \ C \ L$) **note** $CLN = \text{this}(1)$ **and** $MC = \text{this}(2)$ **and** $\text{undef-L} = \text{this}(3)$ **and** $T = \text{this}(4)$

have $\text{incl}: \text{atm-of } ' \text{ lits-of } (\text{Propagated } L \ ()) \# \text{trail } S \subseteq \text{atms-of-ms } A$

using $\text{propagate}_{NOT}.\text{hyps}$ $\text{propagate-ops.propagate}_{NOT}$ $\text{dpll-bj-atms-in-trail-in-set}$ $\text{bj-propagate}_{NOT}$

$NA \ MA \ CLN$ **by** (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

have $\text{no-dup}: \text{no-dup } (\text{Propagated } L \ ()) \# \text{trail } S$

using $\text{defined-lit-map } n\text{-d } \text{undef-L}$ **by** *auto*

obtain $a \ b \ l$ **where** $M: \text{get-all-marked-decomposition } (\text{trail } S) = (a, b) \# l$

by (*case-tac get-all-marked-decomposition (trail S) auto*)

have $b\text{-le-M}: \text{length } b \leq \text{length } (\text{trail } S)$

using $\text{get-all-marked-decomposition-decomp}[of \ \text{trail } S]$ **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 incl finite **unfolding** $\text{no-dup-length-eq-card-atm-of-lits-of}[OF \ \text{no-dup}]$

by (*simp add: card-mono*)

```

then have latm: unassigned-lit A b = Suc (unassigned-lit A (Propagated L d # b))
  using b-le-M by auto
then show ?case using T undef-L by (auto simp: latm M  $\mu_C$ -cons)
next
case (decideNOT L) note undef-L = this(1) and MC = this(2) and T = this(3)
have incl: atm-of ' lits-of (Marked L () # (trail S))  $\subseteq$  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 (case-tac 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[OF no-dup]
  by (simp add: card-mono)
then have latm: unassigned-lit A (trail S) = Suc (unassigned-lit A (Marked L lv # (trail S)))
  by force
show ?case using T undef-L by (simp add: latm  $\mu_C$ -cons)
next
case (backjump C F' K F L C' T) note undef-L = this(4) and MC = this(1) and tr-S = this(3)
and
  L = this(5) and T = this(8)
have incl: atm-of ' lits-of (Propagated L () # F)  $\subseteq$  atms-of-ms A
  using dpll-bj-atms-in-trail-in-set NA MA tr-S L by auto

have no-dup: no-dup (Propagated L () # F)
  using defined-lit-map n-d undef-L tr-S by auto
obtain a b l where M: get-all-marked-decomposition (trail S) = (a, b) # l
  by (cases get-all-marked-decomposition (trail S)) auto
have b-le-M: length b  $\leq$  length (trail S)
  using get-all-marked-decomposition-decomp[of trail S] by (simp add: M)
have fin-atms-A: finite (atms-of-ms A) using finite by simp

then have F-le-A: length (Propagated L () # F)  $\leq$  card (atms-of-ms A)
  using incl finite unfolding no-dup-length-eq-card-atm-of-lits-of[OF no-dup]
  by (simp add: card-mono)
have tr-S-le-A: length (trail S)  $\leq$  (card (atms-of-ms A))
  using n-d MA by (metis fin-atms-A card-mono no-dup-length-eq-card-atm-of-lits-of)
obtain a b l where F: get-all-marked-decomposition F = (a, b) # l
  by (cases get-all-marked-decomposition F) auto
then have F = b @ a
  using get-all-marked-decomposition-decomp[of Propagated L () # F a
    Propagated L () # b] by simp
then have latm: unassigned-lit A b = Suc (unassigned-lit A (Propagated L () # b))
  using F-le-A by simp
obtain rem where
  rem: map ( $\lambda a.$  Suc (length (snd a))) (rev (get-all-marked-decomposition (F' @ Marked K () # F)))
  = map ( $\lambda a.$  Suc (length (snd a))) (rev (get-all-marked-decomposition F)) @ rem
  using take-length-get-all-marked-decomposition-marked-sandwich[of F  $\lambda a.$  Suc (length a) F' K]
  unfolding o-def by (metis append-take-drop-id)
then have rem: map ( $\lambda a.$  Suc (length (snd a)))
  (get-all-marked-decomposition (F' @ Marked K () # F))

```

```

= rev rem @ map (λa. Suc (length (snd a))) ((get-all-marked-decomposition F))
by (simp add: rev-map[symmetric] rev-swap)
have length (rev rem @ map (λa. Suc (length (snd a))) (get-all-marked-decomposition F))
  ≤ Suc (card (atms-of-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-msu (clauses S) ⊆ atms-of-ms A* **and**
M-A: *atm-of ' lits-of (trail S) ⊆ atms-of-ms A* **and**
nd: *no-dup (trail S)* **and**
fin-A: *finite A*

shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$

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 (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 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 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* $\{(T, S). \text{dpll-bj } S \ T$

$\wedge \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$

$\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$

(is *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*: $(b, a) \in \{(T, S). \text{dpll-bj } S \ T$

$\wedge \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$

$\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$

have *fin-A*: *finite* (*atms-of-ms A*)

using *fin* **by** *auto*

have

dpll-bj: *dpll-bj a b* **and**

N-A: *atms-of-msu* (*clauses a*) $\subseteq \text{atms-of-ms } A$ **and**

M-A: *atm-of* ' *lits-of* (*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* (*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 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 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

14.3.4 Normal Forms

We prove that given a normal form of DPLL, with some invariants, the either N is satisfiable and the built valuation M is a model; or N is unsatisfiable.

Idea of the proof: We have to prove that *satisfiable* N , $\neg M \models_{as} N$ and there is no remaining step is incompatible.

1. The *decide* rules tells us that every variable in N has a value.
2. $\neg M \models_{as} N$ tells us that there is conflict.
3. There is at least one decision in the trail (otherwise, M is a model of N).
4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

The assumption are saying that we have a finite upper bound A for the literals, that we cannot do any step *no-step dpll-bj* S

theorem *dpll-backjump-final-state*:

fixes $A :: 'v \text{ literal multiset set}$ **and** $S \ T :: 'st$

assumes

atms-of-msu (*clauses* S) \subseteq *atms-of-ms* A **and**

atm-of ' *lits-of* (*trail* S) \subseteq *atms-of-ms* A **and**

no-dup (*trail* S) **and**

finite A **and**

inv: *inv* S **and**

n-s: *no-step dpll-bj* S **and**

decomp: *all-decomposition-implies-m* (*clauses* S) (*get-all-marked-decomposition* (*trail* S))

shows *unsatisfiable* (*set-mset* (*clauses* S))

\vee (*trail* $S \models_{asm}$ *clauses* $S \wedge$ *satisfiable* (*set-mset* (*clauses* S)))

proof –

let $?N = \text{set-mset} \ (\text{clauses } S)$

let $?M = \text{trail } S$

consider

(*sat*) *satisfiable* $?N$ **and** $?M \models_{as} ?N$

| (*sat'*) *satisfiable* $?N$ **and** $\neg ?M \models_{as} ?N$

| (*unsat*) *unsatisfiable* $?N$

by *auto*

then show *?thesis*

proof *cases*

case *sat'* **note** $\text{sat} = \text{this}(1)$ **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  $?I = I \cup \{P \mid P. P \in \text{lits-of } ?M \wedge \text{atm-of } P \notin \text{atm-of } I\}$ 
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  $?I$   $(?N \cup (\lambda a. \{\# \text{lit-of } a\# \}))$   $' \text{set } ?M$ 
  using tot atms-of-s-def unfolding total-over-m-def total-over-set-def
  by fastforce
have  $\{P \mid P. P \in \text{lits-of } ?M \wedge \text{atm-of } P \notin \text{atm-of } I\} \models_s ?O$ 
  using  $\langle I \models_s ?N \rangle$  atm-I-N by (auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def)
then have  $I' \models_s ?N \cup ?O$ 
  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  $?N \subseteq \text{atm-of } ' \text{lits-of } ?M$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $l :: 'v$  where
     $l \in \text{atms-of-ms } ?N$  and
     $l \notin \text{atm-of } ' \text{lits-of } ?M$ 
    by auto
  have undefined-lit  $?M$   $(\text{Pos } l)$ 
    using  $l \in \text{atms-of-ms } ?N$  by (metis Marked-Propagated-in-iff-in-lits-of
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
  from bj-decideNOT $[OF \text{ decide}_{NOT}[OF \text{ this}]]$  show False
    using  $l \in \text{atms-of-ms } ?N$  by (metis literal.sel(1) state-eqNOT-ref)
qed

have  $?M \models_{as} C \text{Not } C$ 
by (metis  $\langle C \in \text{set-mset } (\text{clauses } S) \rangle \langle \neg \text{trail } S \models_a C \rangle$  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  $\vartheta[\text{iff}]: \bigwedge I. \text{total-over-m } I$   $(?N \cup ?O \cup (\lambda a. \{\# \text{lit-of } a\# \}))$   $' \text{set } ?M$ 
     $\longleftrightarrow \text{total-over-m } I$   $(?N \cup (\lambda a. \{\# \text{lit-of } a\# \}))$   $' \text{set } ?M$ 
    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  $?N \cup ?O \models_{ps} (\lambda a. \{\# \text{lit-of } a\# \})$   $' \text{set } ?M$ 
    using all-decomposition-implies-propagated-lits-are-implied $[OF \text{ decomp}]$  by auto

```

```

then have ?I  $\models_s$  ( $\lambda a. \{\#lit\text{-}of\ a\#\}$ ) ‘ set ?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 ?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  $\langle trail\ S \models_{as}\ CNot\ C \rangle$  consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
    true-annots-def true-clss-mono-set-mset-l true-clss-def)
then show False using M by fast
qed
from List.split-list-first-propE[OF this] obtain K :: 'v literal 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  $\in$  set ?M
  unfolding M-K by auto
let ?C = image-mset lit-of  $\{\#L \in \#mset\ ?M. is\ marked\ L \wedge L \neq ?K\#\}$  :: 'v literal multiset
let ?C' = set-mset (image-mset ( $\lambda L. :: 'v\ literal. \{\#L\#\}$ ) (?C +  $\{\#lit\text{-}of\ ?K\#\}$ ))
have ?N  $\cup \{\{\#lit\text{-}of\ L\#\} \mid L. is\ marked\ L \wedge L \in set\ ?M\} \models_{ps}$  ( $\lambda a. \{\#lit\text{-}of\ a\#\}$ ) ‘ set ?M
  using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have C': ?C' =  $\{\{\#lit\text{-}of\ L\#\} \mid L. is\ marked\ L \wedge L \in set\ ?M\}$ 
  unfolding M-K apply standard
  apply force
  using IntI by auto
ultimately have N-C-M: ?N  $\cup$  ?C'  $\models_{ps}$  ( $\lambda a. \{\#lit\text{-}of\ a\#\}$ ) ‘ set ?M
  by auto
have N-M-False: ?N  $\cup$  ( $\lambda L. \{\#lit\text{-}of\ L\#\}$ ) ‘ (set ?M)  $\models_{ps}$   $\{\{\#\}\}$ 
  using M  $\langle ?M \models_{as}\ CNot\ C \rangle \langle C \in ?N \rangle$  unfolding true-clss-clss-def true-annots-def Ball-def
    true-annot-def by (metis consistent-CNot-not sup.orderE sup-commute true-clss-def
      true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have undefined-lit F K using  $\langle no\text{-}dup\ ?M \rangle$  unfolding M-K by (simp add: defined-lit-map)
moreover
  have ?N  $\cup$  ?C'  $\models_{ps}$   $\{\{\#\}\}$ 
  proof –
    have A: ?N  $\cup$  ?C'  $\cup$  ( $\lambda a. \{\#lit\text{-}of\ a\#\}$ ) ‘ set ?M =
      ?N  $\cup$  ( $\lambda a. \{\#lit\text{-}of\ a\#\}$ ) ‘ set ?M
    unfolding M-K by auto
    show ?thesis
    using true-clss-clss-left-right[OF N-C-M, of  $\{\{\#\}\}$ ] N-M-False unfolding A by auto
  qed
have ?N  $\models_p$  image-mset uminus ?C +  $\{\#-K\#\}$ 
unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
proof (intro allI impI)
  fix I
  assume
    tot: total-over-set I (atms-of-ms (?N  $\cup$   $\{image\text{-}mset\ uminus\ ?C + \{\#-K\#\}\}$ ) and
    cons: consistent-interp I and
    I  $\models_s$  ?N
  have ( $K \in I \wedge -K \notin I$ )  $\vee$  ( $-K \in I \wedge K \notin I$ )
    using cons tot unfolding consistent-interp-def by (cases K) auto
  have tot': total-over-set I
    (atm-of ‘ lit-of ‘ (set ?M  $\cap$   $\{L. is\ marked\ L \wedge L \neq Marked\ K\ ()\}$ ))

```

```

    using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
  { fix x :: ('v, unit, unit) marked-lit
    assume
      a3: lit-of x  $\notin$  I and
      a1: x  $\in$  set ?M and
      a4: is-marked x and
      a5: x  $\neq$  Marked K ()
    then have Pos (atm-of (lit-of x))  $\in$  I  $\vee$  Neg (atm-of (lit-of x))  $\in$  I
      using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast
    moreover have f6: Neg (atm-of (lit-of x)) = - Pos (atm-of (lit-of x))
      by simp
    ultimately have - lit-of x  $\in$  I
      using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
        literal.sel(1))
  } note H = this

  have  $\neg I \models_s ?C'$ 
    using  $\langle ?N \cup ?C' \models_{ps} \{\{\#\}\} \rangle$  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-mset-def
    using  $\langle (K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I) \rangle$ 
    by (auto dest!: H)
  qed
  moreover have F  $\models_{as}$  CNot (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 trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv backjump-conds
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
  +
  assumes dpll-bj-inv:  $\bigwedge S T. dpll-bj S T \Longrightarrow inv S \Longrightarrow inv T$ 
begin

lemma rtranclp-dpll-bj-inv:
  assumes dpll-bj* S T and inv S
  shows inv T

```

using *assms* **by** (*induction rule: rtranclp-induct*)
 (*auto simp add: dpll-bj-no-dup intro: dpll-bj-inv*)

lemma *rtranclp-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: rtranclp-induct*)
 (*auto simp add: dpll-bj-no-dup dest: rtranclp-dpll-bj-inv dpll-bj-inv*)

lemma *rtranclp-dpll-bj-atms-of-ms-clauses-inv*:
assumes
*dpll-bj** S T and inv S*
shows *atms-of-msu (clauses S) = atms-of-msu (clauses T)*
using *assms* **by** (*induction rule: rtranclp-induct*)
 (*auto dest: rtranclp-dpll-bj-inv dpll-bj-atms-of-ms-clauses-inv*)

lemma *rtranclp-dpll-bj-atms-in-trail*:
assumes
*dpll-bj** S T and*
inv S and
atm-of ' (lits-of (trail S)) \subseteq atms-of-msu (clauses S)
shows *atm-of ' (lits-of (trail T)) \subseteq atms-of-msu (clauses T)*
using *assms* **apply** (*induction rule: rtranclp-induct*)
using *dpll-bj-atms-in-trail dpll-bj-atms-of-ms-clauses-inv rtranclp-dpll-bj-inv* **by** *auto*

lemma *rtranclp-dpll-bj-sat-iff*:
assumes *dpll-bj** S T and inv S*
shows *I \models_{sm} clauses S \longleftrightarrow I \models_{sm} clauses T*
using *assms* **by** (*induction rule: rtranclp-induct*)
 (*auto dest!: dpll-bj-sat-iff simp: rtranclp-dpll-bj-inv*)

lemma *rtranclp-dpll-bj-atms-in-trail-in-set*:
assumes
*dpll-bj** S T and*
inv S
atms-of-msu (clauses S) \subseteq A and
atm-of ' (lits-of (trail S)) \subseteq A
shows *atm-of ' (lits-of (trail T)) \subseteq A*
using *assms*
by (*induction rule: rtranclp-induct*)
 (*auto dest: rtranclp-dpll-bj-inv*
simp add: dpll-bj-atms-in-trail-in-set rtranclp-dpll-bj-atms-of-ms-clauses-inv
rtranclp-dpll-bj-inv)

lemma *rtranclp-dpll-bj-all-decomposition-implies-inv*:
assumes
*dpll-bj** S T and*
inv S
all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
shows *all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))*
using *assms* **by** (*induction rule: rtranclp-induct*)
 (*auto intro: dpll-bj-all-decomposition-implies-inv simp: rtranclp-dpll-bj-inv*)

lemma *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*:

$\{(T, S). \text{dpll-bj}^{++} S T$
 $\wedge \text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$
 $\subseteq \{(T, S). \text{dpll-bj } S T \wedge \text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}^+$
 $(\text{is } ?A \subseteq ?B^+)$
proof *standard*
fix x
assume $x-A: x \in ?A$
obtain $S T::'st$ **where**
 $x[\text{simp}]: x = (T, S)$ **by** $(\text{cases } x)$ *auto*
have
 $\text{dpll-bj}^{++} S T$ **and**
 $\text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{no-dup } (\text{trail } S)$ **and**
 $\text{inv } S$
using $x-A$ **by** *auto*
then show $x \in ?B^+$ **unfolding** x
proof (*induction rule: tranclp-induct*)
case *base*
then show $?case$ **by** *auto*
next
case ($\text{step } T U$) **note** $\text{step} = \text{this}(1)$ **and** $ST = \text{this}(2)$ **and** $IH = \text{this}(3)[\text{OF } \text{this}(4-7)]$
and $N-A = \text{this}(4)$ **and** $M-A = \text{this}(5)$ **and** $nd = \text{this}(6)$ **and** $\text{inv} = \text{this}(7)$

have $[\text{simp}]: \text{atms-of-msu} (\text{clauses } S) = \text{atms-of-msu} (\text{clauses } T)$
using $\text{step rtranclp-dpll-bj-atms-of-ms-clauses-inv tranclp-into-rtranclp inv}$ **by** *fastforce*
have $\text{no-dup } (\text{trail } T)$
using $\text{local.step nd rtranclp-dpll-bj-no-dup tranclp-into-rtranclp inv}$ **by** *fastforce*
moreover have $\text{atm-of ' (lits-of } (\text{trail } T)) \subseteq \text{atms-of-ms } A$
by $(\text{metis inv M-A N-A local.step rtranclp-dpll-bj-atms-in-trail-in-set}$
 $\text{tranclp-into-rtranclp})$
moreover have $\text{inv } T$
using $\text{inv local.step rtranclp-dpll-bj-inv tranclp-into-rtranclp}$ **by** *fastforce*
ultimately have $(U, T) \in ?B$ **using** $ST N-A M-A \text{inv}$ **by** *auto*
then show $?case$ **using** IH **by** (*rule trancl-into-trancl2*)
qed
qed

lemma *wf-tranclp-dpll-bj*:

assumes $\text{fin}: \text{finite } A$

shows $\text{wf } \{(T, S). \text{dpll-bj}^{++} S T$

$\wedge \text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$

$\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$

using $\text{wf-trancl}[\text{OF } \text{wf-dpll-bj}[\text{OF } \text{fin}]] \text{ rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl}$

by (*rule wf-subset*)

lemma *dpll-bj-sat-ext-iff*:

$\text{dpll-bj } S T \implies \text{inv } S \implies I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$

by (*simp add: dpll-bj-clauses*)

lemma *rtranclp-dpll-bj-sat-ext-iff*:

$\text{dpll-bj}^{**} S T \implies \text{inv } S \implies I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$

by (*induction rule: rtranclp-induct*) (*simp-all add: rtranclp-dpll-bj-inv dpll-bj-sat-ext-iff*)

theorem *full-dpll-backjump-final-state:*

fixes $A :: 'v$ literal multiset set **and** $S\ T :: 'st$

assumes

full: *full dpll-bj* $S\ T$ **and**

atms-S: *atms-of-msu* (*clauses* S) \subseteq *atms-of-ms* A **and**

atms-trail: *atm-of* ' *lits-of* (*trail* S) \subseteq *atms-of-ms* A **and**

n-d: *no-dup* (*trail* S) **and**

finite A **and**

inv: *inv* S **and**

decomp: *all-decomposition-implies-m* (*clauses* S) (*get-all-marked-decomposition* (*trail* S))

shows *unsatisfiable* (*set-mset* (*clauses* S))

\vee (*trail* $T \models_{asm}$ *clauses* $S \wedge$ *satisfiable* (*set-mset* (*clauses* S)))

proof –

have *st*: *dpll-bj*** $S\ T$ **and** *no-step dpll-bj* T

using *full unfolding full-def* **by** *fast+*

moreover have *atms-of-msu* (*clauses* 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* (*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* T) (*get-all-marked-decomposition* (*trail* T))

using $\langle inv\ S \rangle$ *decomp rtranclp-dpll-bj-all-decomposition-implies-inv st* **by** *blast*

ultimately have *unsatisfiable* (*set-mset* (*clauses* T))

\vee (*trail* $T \models_{asm}$ *clauses* $T \wedge$ *satisfiable* (*set-mset* (*clauses* T)))

using $\langle finite\ A \rangle$ *dpll-backjump-final-state* **by** *force*

then show *?thesis*

by (*meson* $\langle inv\ S \rangle$ *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 $S = []$ **and**

clauses $S = N$ **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-msu* (*clauses* S) \subseteq *atms-of-ms* A **and**

M-A: *atm-of* ' *lits-of* (*trail* S) \subseteq *atms-of-ms* A **and**

n-d: *no-dup* (*trail* S) **and**

fin-A: *finite* A

shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$

$- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$

$< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$

$- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$

using *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-msu (clauses S) = atms-of-msu (clauses 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-msu (clauses T) ⊆ atms-of-ms A
    using N-A by auto
  moreover have M-A': atm-of ' lits-of (trail T) ⊆ 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

```

14.4 CDCL

14.4.1 Learn and Forget

```

locale learn-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
  for
    trail :: 'st ⇒ ('v, unit, unit) marked-lits and
    clauses :: 'st ⇒ 'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and tl-trail :: 'st ⇒ 'st and
    add-clsNOT remove-clsNOT :: 'v clause ⇒ 'st ⇒ 'st +
  fixes
    learn-cond :: 'v clause ⇒ 'st ⇒ bool

  begin
  inductive learn :: 'st ⇒ 'st ⇒ bool where
    clauses S ⊨pm C ⇒ atms-of C ⊆ atms-of-msu (clauses S) ∪ atm-of ' (lits-of (trail S))
     $\Rightarrow$  learn-cond C S
     $\Rightarrow$  T ∼ add-clsNOT C S
     $\Rightarrow$  learn S T
  inductive-cases learnE: learn S T

  lemma learn-μC-stable:
    assumes learn S T and no-dup (trail S)
    shows  $\mu_C A B$  (trail-weight S) =  $\mu_C A B$  (trail-weight T)
    using assms by (auto elim: learnE)
  end

```

```

locale forget-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
  for
    trail :: 'st ⇒ ('v, unit, unit) marked-lits and

```

```

  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT:: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive forgetNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
forgetNOT:clauses S – replicate-mset (count (clauses S) C) C  $\models_{pm}$  C
 $\Rightarrow$  forget-cond C S
 $\Rightarrow$  C  $\in \#$  clauses S
 $\Rightarrow$  T  $\sim$  remove-clNOT C S
 $\Rightarrow$  forgetNOT S T
inductive-cases forgetE: forgetNOT S T

lemma forget- $\mu_C$ -stable:
  assumes forgetNOT S T
  shows  $\mu_C$  A B (trail-weight S) =  $\mu_C$  A B (trail-weight T)
  using assms by (auto elim!: forgetE)
end

locale learn-and-forgetNOT =
  learn-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT learn-cond +
  forget-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT forget-cond
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT:: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  learn-cond forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive learn-and-forgetNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool
where
lf-learn: learn S T  $\Rightarrow$  learn-and-forgetNOT S T |
lf-forget: forgetNOT S T  $\Rightarrow$  learn-and-forgetNOT S T
end

```

14.4.2 Definition of CDCL

```

locale conflict-driven-clause-learning-ops =
  dpll-with-backjumping-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv backjump-conds +
  learn-and-forgetNOT trail clauses prepend-trail tl-trail add-clNOT remove-clNOT learn-cond
  forget-cond
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT:: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  learn-cond forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

```


inductive $cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**

$c-dpll-bj$: $dpll-bj\ S\ S' \Longrightarrow cdcl_{NOT}\ S\ S' \mid$

$c-learn$: $learn\ S\ S' \Longrightarrow cdcl_{NOT}\ S\ S' \mid$

$c-forget_{NOT}$: $forget_{NOT}\ S\ S' \Longrightarrow cdcl_{NOT}\ S\ S'$

lemma $cdcl_{NOT}$ -all-induct[consumes 1, case-names $dpll-bj\ learn\ forget_{NOT}$]:

fixes $S\ T :: 'st$

assumes $cdcl_{NOT}\ S\ T$ **and**

$dpll$: $\bigwedge T. dpll-bj\ S\ T \Longrightarrow P\ S\ T$ **and**

$learning$:

$\bigwedge C\ T. clauses\ S \models_{pm} C \Longrightarrow$

$atms-of\ C \subseteq atms-of-msu\ (clauses\ S) \cup atm-of\ ' (lits-of\ (trail\ S)) \Longrightarrow$

$T \sim add-cl_{NOT}\ C\ S \Longrightarrow$

$P\ S\ T$ **and**

$forgetting$: $\bigwedge C\ T. clauses\ S - replicate-mset\ (count\ (clauses\ S)\ C)\ C \models_{pm} C \Longrightarrow$

$C \in \# clauses\ S \Longrightarrow$

$T \sim remove-cl_{NOT}\ C\ S \Longrightarrow$

$P\ S\ T$

shows $P\ S\ T$

using $assms(1)$ **by** (induction rule: $cdcl_{NOT}.induct$)

(auto intro: $assms(2, 3, 4)$ elim!: $learnE\ forgetE$)+

lemma $cdcl_{NOT}$ -no-dup:

assumes

$cdcl_{NOT}\ S\ T$ **and**

$inv\ S$ **and**

$no-dup\ (trail\ S)$

shows $no-dup\ (trail\ T)$

using $assms$ **by** (induction rule: $cdcl_{NOT}$ -all-induct) (auto intro: $dpll-bj$ -no-dup)

Consistency of the trail lemma $cdcl_{NOT}$ -consistent:

assumes

$cdcl_{NOT}\ S\ T$ **and**

$inv\ S$ **and**

$no-dup\ (trail\ S)$

shows $consistent-interp\ (lits-of\ (trail\ T))$

using $cdcl_{NOT}$ -no-dup[OF $assms$] $distinctconsistent-interp$ **by** fast

The subtle problem here is that tautologies can be removed, meaning that some variable can disappear of the problem. It is also possible that some variable of the trail are not in the clauses anymore.

lemma $cdcl_{NOT}$ -atms-of-ms-clauses-decreasing:

assumes $cdcl_{NOT}\ S\ T$ **and** $inv\ S$ **and** $no-dup\ (trail\ S)$

shows $atms-of-msu\ (clauses\ T) \subseteq atms-of-msu\ (clauses\ S) \cup atm-of\ ' (lits-of\ (trail\ S))$

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 $atm-of\ ' (lits-of\ (trail\ S)) \subseteq atms-of-msu\ (clauses\ S)$

shows $atm-of\ ' (lits-of\ (trail\ T)) \subseteq atms-of-msu\ (clauses\ S)$

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

```

    cdclNOT S T and inv S and no-dup (trail S) and
    atms-of-msu (clauses S) ⊆ A and
    atm-of ‘ (lits-of (trail S)) ⊆ A
  shows atm-of ‘ (lits-of (trail T)) ⊆ A
  using assms
  by (induction rule: cdclNOT-all-induct)
    (simp-all add: dpll-bj-atms-in-trail-in-set dpll-bj-atms-of-ms-clauses-inv)

lemma cdclNOT-all-decomposition-implies:
  assumes cdclNOT S T and inv S and n-d[simp]: no-dup (trail S) and
    all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
  shows
    all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))
  using assms(1,2,4)
proof (induction rule: cdclNOT-all-induct)
  case dpll-bj
  then show ?case
    using dpll-bj-all-decomposition-implies-inv n-d by blast
next
  case learn
  then show ?case by (auto simp add: all-decomposition-implies-def)
next
  case (forgetNOT C T) note cls-C = this(1) and C = this(2) and T = this(3) and inv = this(4)
  and
    decomp = this(5)
  show ?case
    unfolding all-decomposition-implies-def Ball-def
  proof (intro allI, clarify)
    fix a b
    assume (a, b) ∈ set (get-all-marked-decomposition (trail T))
    then have (λa. {#lit-of a#}) ‘ set a ∪ set-mset (clauses S) ⊨ps (λa. {#lit-of a#}) ‘ set b
      using decomp T by (auto simp add: all-decomposition-implies-def)
    moreover
      have C ∈ set-mset (clauses S)
      by (simp add: C)
    then have set-mset (clauses T) ⊨ps set-mset (clauses S)
      by (metis (no-types) T clauses-remove-clsNOT cls-C insert-Diff order-refl
        set-mset-minus-replicate-mset(1) state-eqNOT-clauses true-clss-clss-def
        true-clss-clss-insert)
    ultimately show (λa. {#lit-of a#}) ‘ set a ∪ set-mset (clauses T)
      ⊨ps (λa. {#lit-of a#}) ‘ set b
      using true-clss-clss-generalise-true-clss-clss by blast
  qed
qed

```

Extension of models lemma cdcl_{NOT}-bj-sat-ext-iff:

```

  assumes cdclNOT S T and inv S and n-d: no-dup (trail S)
  shows I ⊨sextm clauses S ↔ I ⊨sextm clauses T
  using assms
proof (induction rule: cdclNOT-all-induct)
  case dpll-bj
  then show ?case by (simp add: dpll-bj-clauses)
next
  case (learn C T) note T = this(3)
  { fix J

```

```

assume
   $I \models_{\text{sextm}} \text{clauses } S$  and
   $I \subseteq J$  and
   $\text{tot: total-over-}m \ J \ (\text{set-mset } (\{\#C\} + (\text{clauses } S)))$  and
   $\text{cons: consistent-interp } J$ 
then have  $J \models_{\text{sm}} \text{clauses } S$  unfolding true-clss-ext-def by auto

moreover
  with  $\langle \text{clauses } S \models_{\text{pm}} C \rangle$  have  $J \models C$ 
  using tot cons unfolding true-clss-cls-def by auto
ultimately have  $J \models_{\text{sm}} \{\#C\} + \text{clauses } S$  by auto
}
then have  $H: I \models_{\text{sextm}} (\text{clauses } S) \implies I \models_{\text{sext}} \text{insert } C \ (\text{set-mset } (\text{clauses } S))$ 
unfolding true-clss-ext-def by auto
show ?case
apply standard
  using T n-d apply (auto simp add: H)[]
using T n-d apply simp
by (metis Diff-insert-absorb insert-subset subsetI subset-antisym
  true-clss-ext-decrease-right-remove-r)
next
case (forgetNOT C T) note  $\text{cls-}C = \text{this}(1)$  and  $T = \text{this}(3)$ 
{ fix  $J$ 
  assume
     $I \models_{\text{sext}} \text{set-mset } (\text{clauses } S) - \{C\}$  and
     $I \subseteq J$  and
     $\text{tot: total-over-}m \ J \ (\text{set-mset } (\text{clauses } S))$  and
     $\text{cons: consistent-interp } J$ 
  then have  $J \models_{\text{s}} \text{set-mset } (\text{clauses } S) - \{C\}$ 
  unfolding true-clss-ext-def by (meson Diff-subset total-over-m-subset)

  moreover
    with  $\text{cls-}C$  have  $J \models C$ 
    using tot cons unfolding true-clss-cls-def
    by (metis Un-commute forgetNOT.hyps(2) insert-Diff insert-is-Un mem-set-mset-iff order-refl
      set-mset-minus-replicate-mset(1))
    ultimately have  $J \models_{\text{sm}} (\text{clauses } S)$  by (metis insert-Diff-single true-clss-insert)
  }
then have  $H: I \models_{\text{sext}} \text{set-mset } (\text{clauses } S) - \{C\} \implies I \models_{\text{sextm}} (\text{clauses } S)$ 
unfolding true-clss-ext-def by blast
show ?case using T by (auto simp: true-clss-ext-decrease-right-remove-r H)
qed

end — end of conflict-driven-clause-learning-ops

```

14.5 CDCL with invariant

```

locale conflict-driven-clause-learning =
  conflict-driven-clause-learning-ops +
  assumes  $\text{cdcl}_{\text{NOT-inv}}: \bigwedge S \ T. \ \text{cdcl}_{\text{NOT}} \ S \ T \implies \text{inv } S \implies \text{inv } T$ 
begin
sublocale dpll-with-backjumping
  apply unfold-locales
  using cdclNOT.simps cdclNOT-inv by auto

lemma rtranchp-cdclNOT-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: $atms-of-msu (clauses S) \subseteq A$ **and**
atms-trail-S: $atm-of (lits-of (trail S)) \subseteq A$
shows $atm-of (lits-of (trail T)) \subseteq A \wedge atms-of-msu (clauses T) \subseteq A$
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 $atms-of-msu (clauses U) \subseteq A$
using *cdcl_{NOT}-atms-of-ms-clauses-decreasing*[OF $cdcl_{NOT}$] IH $n-d$ $\langle inv T \rangle$ **by** *auto*
moreover
have $atm-of (lits-of (trail U)) \subseteq A$
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 S) (*get-all-marked-decomposition* (trail S))
shows
all-decomposition-implies-m (clauses T) (*get-all-marked-decomposition* (trail T))
using *assms* **by** (induction)
(auto intro: *rtranclp-cdcl_{NOT}-inv cdcl_{NOT}-all-decomposition-implies rtranclp-cdcl_{NOT}-no-dup*)

lemma *rtranclp-cdcl_{NOT}-bj-sat-ext-iff*:
assumes $cdcl_{NOT}^{**} S T$ **and** $inv S$ **and** *no-dup* (trail S)
shows $I \models_{extm} clauses S \longleftrightarrow I \models_{extm} clauses T$
using *assms* **apply** (induction rule: *rtranclp-induct*)
using *cdcl_{NOT}-bj-sat-ext-iff* **by** (auto intro: *rtranclp-cdcl_{NOT}-inv rtranclp-cdcl_{NOT}-no-dup*)

definition *cdcl_{NOT}-NOT-all-inv* **where**
 $cdcl_{NOT}-NOT-all-inv A S \longleftrightarrow (finite A \wedge inv S \wedge atms-of-msu (clauses S) \subseteq atms-of-ms A$
 $\wedge atm-of (lits-of (trail S)) \subseteq atms-of-ms A \wedge no-dup (trail S))$

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: rtrancp-cdcl_{NOT-inv} rtrancp-cdcl_{NOT-no-dup} rtrancp-cdcl_{NOT-trail-clauses-bound}*)

abbreviation *learn-or-forget* **where**

learn-or-forget $S T \equiv (\lambda S T. \text{learn } S T \vee \text{forget}_{NOT} S T) S T$

lemma *rtrancp-learn-or-forget-cdcl_{NOT}*:

*learn-or-forget*** $S T \implies cdcl_{NOT}^{**} S T$

using *rtrancp-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 $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } U)$
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$
(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 *rtrancp-cdcl_{NOT-no-dup}*[*of S T*] *cdcl_{NOT-NOT-all-inv-def inv}*

rtrancp-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 *rtrancp-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 $\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i))$

using *assms*

proof (*induction* $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$)

$- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (f 0))$

arbitrary: *f*

rule: *nat-less-induct-case*)

case (*Suc* n) **note** $IH = \text{this}(1)$ **and** $\mu = \text{this}(2)$ **and** $cdcl_{NOT} = \text{this}(3)$ **and** $inv = \text{this}(4)$

consider

```

  (dpll-end)  $\exists j. \forall i \geq j. \text{learn-or-forget } (f \ i) \ (f \ (\text{Suc } i))$ 
| (dpll-more)  $\neg(\exists j. \forall i \geq j. \text{learn-or-forget } (f \ i) \ (f \ (\text{Suc } i)))$ 
by blast
then show ?case
proof cases
  case dpll-end
  then show ?thesis by auto
next
case dpll-more
then have j:  $\exists i. \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ 
  by blast
obtain i where
   $\neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$  and
   $\forall k < i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
proof -
  obtain i0 where  $\neg \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f \ i_0) \ (f \ (\text{Suc } i_0))$ 
    using j by auto
  then have {i.  $i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ }  $\neq \{\}$ 
    by auto
  let ?I = {i.  $i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ }
  let ?i = Min ?I
  have finite ?I
    by auto
  have  $\neg \text{learn } (f \ ?i) \ (f \ (\text{Suc } ?i)) \wedge \neg \text{forget}_{NOT} (f \ ?i) \ (f \ (\text{Suc } ?i))$ 
    using Min-in[OF ⟨finite ?I⟩ ⟨?I  $\neq \{\}$ ⟩] by auto
  moreover have  $\forall k < ?i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
    using Min.coboundedI[of {i.  $i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ }, simplified]
    by (meson  $\neg \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f \ i_0) \ (f \ (\text{Suc } i_0))$ ) less-imp-le
    dual-order.trans not-le
  ultimately show ?thesis using that by blast
qed
def g  $\equiv \lambda n. f \ (n + \text{Suc } i)$ 
have dpll-bj (f i) (g 0)
  using  $\neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$  cdclNOT cdclNOT.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
       $\langle \forall k < i. \text{learn } (f \ k) \ (f \ (\text{Suc } k)) \vee \text{forget}_{NOT} (f \ k) \ (f \ (\text{Suc } k)) \rangle$ )
}
then have learn-or-forget** (f 0) (f i) by blast
then have  $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (g \ 0))$ 
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (f \ 0))$ 
  using learn-or-forget-dpll- $\mu_C$ [of f 0 f i g 0 A] inv ⟨dpll-bj (f i) (g 0)⟩
  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] ⟨learn-or-forget** (f 0) (f i)⟩

```

```

    cdclNOT[of i] unfolding g-def by auto
moreover have  $\bigwedge i. \text{cdcl}_{NOT} (g \ i) (g \ (\text{Suc } i))$ 
    using cdclNOT g-def by auto
moreover have cdclNOT-NOT-all-inv A (g 0)
    using inv cdclNOT-i rtrancpl-cdclNOT-trail-clauses-bound g-def cdclNOT-NOT-all-inv by auto
ultimately obtain j where j:  $\bigwedge i. i \geq j \implies \text{learn-or-forget} (g \ i) (g \ (\text{Suc } i))$ 
    using IH unfolding  $\mu[\text{symmetric}]$  by presburger
show ?thesis
proof
  {
    fix k
    assume  $k \geq j + \text{Suc } i$ 
    then have learn-or-forget (f k) (f (Suc k))
      using j[of k-Suc i] unfolding g-def by auto
  }
  then show  $\forall k \geq j + \text{Suc } i. \text{learn-or-forget} (f \ k) (f \ (\text{Suc } k))$ 
    by auto
qed
qed
next
case 0 note H = this(1) and cdclNOT = this(2) and inv = this(3)
show ?case
proof (rule ccontr)
  assume  $\neg ?case$ 
  then have j:  $\exists i. \neg \text{learn} (f \ i) (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) (f \ (\text{Suc } i))$ 
    by blast
  obtain i where
     $\neg \text{learn} (f \ i) (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) (f \ (\text{Suc } i))$  and
     $\forall k < i. \text{learn-or-forget} (f \ k) (f \ (\text{Suc } k))$ 
  proof -
    obtain i0 where  $\neg \text{learn} (f \ i_0) (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f \ i_0) (f \ (\text{Suc } i_0))$ 
      using j by auto
    then have  $\{i. i \leq i_0 \wedge \neg \text{learn} (f \ i) (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) (f \ (\text{Suc } i))\} \neq \{\}$ 
      by auto
    let ?I =  $\{i. i \leq i_0 \wedge \neg \text{learn} (f \ i) (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) (f \ (\text{Suc } i))\}$ 
    let ?i = Min ?I
    have finite ?I
      by auto
    have  $\neg \text{learn} (f \ ?i) (f \ (\text{Suc } ?i)) \wedge \neg \text{forget}_{NOT} (f \ ?i) (f \ (\text{Suc } ?i))$ 
      using Min-in[OF (finite ?I) (?I ≠ {})] by auto
    moreover have  $\forall k < ?i. \text{learn-or-forget} (f \ k) (f \ (\text{Suc } k))$ 
      using Min.coboundedI[of {i. i ≤ i0 ∧ ¬ learn (f i) (f (Suc i)) ∧ ¬ forgetNOT (f i) (f (Suc i))}, simplified]
      by (meson (¬ learn (f i0) (f (Suc i0)) ∧ ¬ forgetNOT (f i0) (f (Suc i0))) less-imp-le
        dual-order.trans not-le)
    ultimately show ?thesis using that by blast
  qed
have dpll-bj (f i) (f (Suc i))
  using (¬ learn (f i) (f (Suc i)) ∧ ¬ forgetNOT (f i) (f (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
    ⟨ $\forall k < i. \text{learn } (f k) (f (Suc k)) \vee \text{forget}_{NOT} (f k) (f (Suc k))$ ⟩)
}
then have learn-or-forget** (f 0) (f i) by blast

then show False
  using learn-or-forget-dpll- $\mu_C$ [of f 0 f i f (Suc i) A] inv 0
  ⟨dpll-bj (f i) (f (Suc i))⟩ unfolding cdclNOT-NOT-all-inv-def by linarith
qed
qed

lemma wf-cdclNOT-no-learn-and-forget-infinite-chain:
  assumes
    no-infinite-lf:  $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i)))$ 
  shows wf {(T, S). cdclNOT S T  $\wedge$  cdclNOT-NOT-all-inv A S} (is wf {(T, S). cdclNOT S T
     $\wedge$  ?inv S})
  unfolding wf-iff-no-infinite-down-chain
proof (rule ccontr)
  assume  $\neg \neg (\exists f. \forall i. (f (Suc i), f i) \in \{(T, S). \text{cdcl}_{NOT} S T \wedge ?inv S\})$ 
  then obtain f where
     $\forall i. \text{cdcl}_{NOT} (f i) (f (Suc i)) \wedge ?inv (f i)$ 
  by fast
  then have  $\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i))$ 
  using infinite-cdclNOT-exists-learn-and-forget-infinite-chain[of f] by meson
  then show False using no-infinite-lf by blast
qed

lemma inv-and-tranclp-cdclNOT-tranclp-cdclNOT-and-inv:
   $\text{cdcl}_{NOT}^{++} S T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A S \longleftrightarrow (\lambda S T. \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT}\text{-NOT-all-inv } A S)^{++} S T$ 
  (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  $\wedge$  ?I by blast
qed

lemma wf-tranclp-cdclNOT-no-learn-and-forget-infinite-chain:
  assumes
    no-infinite-lf:  $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i)))$ 
  shows wf {(T, S). cdclNOT++ S T  $\wedge$  cdclNOT-NOT-all-inv A S}
  using wf-tranclp[OF wf-cdclNOT-no-learn-and-forget-infinite-chain[OF no-infinite-lf]]
  apply (rule wf-subset)
  by (auto simp: trancl-set-tranclp inv-and-tranclp-cdclNOT-tranclp-cdclNOT-and-inv)

lemma cdclNOT-final-state:

```



```

assumes
  n-s: no-step cdclNOT S and
  inv: cdclNOT-NOT-all-inv A S and
  decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
shows unsatisfiable (set-mset (clauses S))
   $\vee$  (trail S  $\models_{asm}$  clauses S  $\wedge$  satisfiable (set-mset (clauses S)))
proof –
  have n-s': no-step dpll-bj S
    using n-s by (auto simp: cdclNOT.simps)
  show ?thesis
    apply (rule dpll-backjump-final-state[of S A])
    using inv decomp n-s' unfolding cdclNOT-NOT-all-inv-def by auto
qed

lemma full-cdclNOT-final-state:
assumes
  full: full cdclNOT S T and
  inv: cdclNOT-NOT-all-inv A S and
  n-d: no-dup (trail S) and
  decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
shows unsatisfiable (set-mset (clauses T))
   $\vee$  (trail T  $\models_{asm}$  clauses T  $\wedge$  satisfiable (set-mset (clauses T)))
proof –
  have st: cdclNOT** S T and n-s: no-step cdclNOT T
    using full unfolding full-def by blast+
  have n-s': cdclNOT-NOT-all-inv A T
    using cdclNOT-NOT-all-inv inv st by blast
  moreover have all-decomposition-implies-m (clauses T) (get-all-marked-decomposition (trail T))
    using cdclNOT-NOT-all-inv-def decomp inv rtranclp-cdclNOT-all-decomposition-implies st by auto
  ultimately show ?thesis
    using cdclNOT-final-state n-s by blast
qed

end — end of conflict-driven-clause-learning

```

14.6 Termination

14.6.1 Restricting learn and forget

```

locale conflict-driven-clause-learning-learning-before-backjump-only-distinct-learn =
  conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
  propagate-conds inv backjump-conds
 $\lambda C S.$  distinct-mset C  $\wedge \neg$ tautology C  $\wedge$  learn-restrictions C S  $\wedge$ 
  ( $\exists F K d F' C' L.$  trail S = F' @ Marked K () # F  $\wedge C$  = C' + {#L#}  $\wedge F \models_{as} CNot C'$ 
     $\wedge C' + {#L\#} \notin \# clauses S$ )
 $\lambda C S.$   $\neg(\exists F' F K d L.$  trail S = F' @ Marked K () # F  $\wedge F \models_{as} CNot (C - {#L\#})$ )
 $\wedge$  forget-restrictions C S
for
  trail :: 'st  $\Rightarrow$  ('v::linorder, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT remove-clsNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and

```

```

learn-restrictions forget-restrictions :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

lemma cdclNOT-learn-all-induct[consumes 1, case-names dpll-bj learn forgetNOT]:
  fixes S T :: 'st
  assumes cdclNOT S T and
    dpll:  $\bigwedge T. \text{dpll-bj } S \ T \Longrightarrow P \ S \ T$  and
    learning:
       $\bigwedge C \ F \ K \ F' \ C' \ L \ T. \text{clauses } S \models_{pm} C$ 
       $\Longrightarrow \text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$ 
       $\Longrightarrow \text{distinct-mset } C \Longrightarrow \neg \text{tautology } C \Longrightarrow \text{learn-restrictions } C \ S$ 
       $\Longrightarrow \text{trail } S = F' @ \text{Marked } K \ () \ \# \ F \Longrightarrow C = C' + \{\#L\# \} \Longrightarrow F \models_{as} CNot \ C'$ 
       $\Longrightarrow C' + \{\#L\# \} \notin \text{clauses } S \Longrightarrow T \sim \text{add-cl}_S \ C \ S$ 
       $\Longrightarrow P \ S \ T$  and
    forgetting:  $\bigwedge C \ T. \text{clauses } S - \text{replicate-mset } (\text{count } (\text{clauses } S) \ C) \ C \models_{pm} C$ 
       $\Longrightarrow C \in \# \text{clauses } S$ 
       $\Longrightarrow \neg(\exists F' \ F \ K \ L. \text{trail } S = F' @ \text{Marked } K \ () \ \# \ F \wedge F \models_{as} CNot \ (C - \{\#L\# \}))$ 
       $\Longrightarrow T \sim \text{remove-cl}_S \ C \ S$ 
       $\Longrightarrow \text{forget-restrictions } C \ S \Longrightarrow P \ S \ T$ 
  shows P S T
  using assms(1)
  apply (induction rule: cdclNOT.induct)
  apply (auto dest: assms(2) simp add: learn-ops-axioms)[]
  apply (auto elim!: learn-ops.learn.cases[OF learn-ops-axioms] dest: assms(3))[]
  apply (auto elim!: forget-ops.forgetNOT.cases[OF forget-ops-axioms] dest!: assms(4))
  done

lemma rtranclp-cdclNOT-inv:
  cdclNOT** S T  $\Longrightarrow$  inv S  $\Longrightarrow$  inv T
  apply (induction rule: rtranclp-induct)
  apply simp
  using cdclNOT-inv unfolding conflict-driven-clause-learning-def
  conflict-driven-clause-learning-axioms-def by blast

lemma learn-always-simple-clauses:
  assumes
    learn: learn S T and
    n-d: no-dup (trail S)
  shows set-mset (clauses T - clauses S)
     $\subseteq \text{build-all-simple-clss } (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S)))$ 
proof
  fix C assume C: C  $\in$  set-mset (clauses T - clauses S)
  have distinct-mset C  $\neg$ tautology C using learn C n-d by (elim learnE; auto)+
  then have C  $\in$  build-all-simple-clss (atms-of C)
    using distinct-mset-not-tautology-implies-in-build-all-simple-clss by blast
  moreover have atms-of C  $\subseteq$  atms-of-msu (clauses S)  $\cup$  atm-of ' lits-of (trail S)
    using learn C n-d by (elim learnE) (auto simp: atms-of-ms-def atms-of-def image-Un
      true-annots-CNot-all-atms-defined)
  moreover have finite (atms-of-msu (clauses S)  $\cup$  atm-of ' lits-of (trail S))
    by auto
  ultimately show C  $\in$  build-all-simple-clss (atms-of-msu (clauses S)  $\cup$  atm-of ' lits-of (trail S))
    using build-all-simple-clss-mono by (metis (no-types) insert-subset mk-disjoint-insert)
qed

definition conflicting-bj-clss S  $\equiv$ 

```

$\{C + \{\#L\# \} \mid C \text{ L. } C + \{\#L\# \} \in \# \text{ clauses } S \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$
 $\wedge (\exists F' K F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} \text{CNot } C)\}$

lemma *conflicting-bj-clss-remove-cl_{NOT}[simp]:*
conflicting-bj-clss (remove-cl_{NOT} C S) = conflicting-bj-clss S - {C}
unfolding *conflicting-bj-clss-def* **by** *fastforce*

lemma *conflicting-bj-clss-add-cl_{NOT}-state-eq:*
 $T \sim \text{add-cl}_{\text{NOT}} C' S \implies \text{no-dup } (\text{trail } S) \implies \text{conflicting-bj-clss } T$
 $= \text{conflicting-bj-clss } S$
 $\cup (\text{if } \exists C \text{ L. } C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} \text{CNot } C)$
 $\text{then } \{C'\} \text{ else } \{\})$
unfolding *conflicting-bj-clss-def* **by** *auto metis+*

lemma *conflicting-bj-clss-add-cl_{NOT}:*
 $\text{no-dup } (\text{trail } S) \implies$
 $\text{conflicting-bj-clss } (\text{add-cl}_{\text{NOT}} C' S)$
 $= \text{conflicting-bj-clss } S$
 $\cup (\text{if } \exists C \text{ L. } C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} \text{CNot } C)$
 $\text{then } \{C'\} \text{ else } \{\})$
using *conflicting-bj-clss-add-cl_{NOT}-state-eq* **by** *auto*

lemma *conflicting-bj-clss-incl-clauses:*
 $\text{conflicting-bj-clss } S \subseteq \text{set-mset } (\text{clauses } S)$
unfolding *conflicting-bj-clss-def* **by** *auto*

lemma *finite-conflicting-bj-clss[simp]:*
 $\text{finite } (\text{conflicting-bj-clss } S)$
using *conflicting-bj-clss-incl-clauses[of S]* *rev-finite-subset* **by** *blast*

lemma *learn-conflicting-increasing:*
 $\text{no-dup } (\text{trail } S) \implies \text{learn } S T \implies \text{conflicting-bj-clss } S \subseteq \text{conflicting-bj-clss } T$
apply (*elim learnE*)
by (*subst conflicting-bj-clss-add-cl_{NOT}-state-eq[of T]*) *auto*

abbreviation *conflicting-bj-clss-yet b S* \equiv
 $3 \wedge b - \text{card } (\text{conflicting-bj-clss } S)$

abbreviation $\mu_L :: \text{nat} \Rightarrow 'st \Rightarrow \text{nat} \times \text{nat}$ **where**
 $\mu_L b S \equiv (\text{conflicting-bj-clss-yet } b S, \text{card } (\text{set-mset } (\text{clauses } S)))$

lemma *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched:*
assumes *forget_{NOT} S T*
shows $\text{conflicting-bj-clss } S = \text{conflicting-bj-clss } T$
using *assms* **apply** *induction*
unfolding *conflicting-bj-clss-def*
by (*metis (no-types, lifting) Diff-insert-absorb Set.set-insert clauses-remove-cl_{NOT}*
diff-union-cancelR insert-iff mem-set-mset-iff order-refl set-mset-minus-replicate-mset(1)
state-eq_{NOT}-clauses state-eq_{NOT}-trail trail-remove-cl_{NOT})

lemma *forget- μ_L -decrease:*
assumes *forget_{NOT}: forget_{NOT} S T*
shows $(\mu_L b T, \mu_L b S) \in \text{less-than} <*\text{lex}*> \text{less-than}$

proof –

have $\text{card } (\text{set-mset } (\text{clauses } T)) < \text{card } (\text{set-mset } (\text{clauses } S))$
using forget_{NOT} **apply** *induction*
by ($\text{metis card-Diff1-less clauses-remove-cl}_{NOT}$ $\text{finite-set-mset mem-set-mset-iff order-refl}$
 $\text{set-mset-minus-replicate-mset}(1)$ state-eq_{NOT} - clauses)
then show *?thesis*
unfolding $\text{do-not-forget-before-backtrack-rule-clause-learned-clause-untouched}[OF \text{forget}_{NOT}]$
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 : $\text{learn } S \ T$ **and** $n\text{-d}$: $\text{no-dup } (\text{trail } S)$ **and**
 A : $\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S) \subseteq A$ **and**
 fin-A : $\text{finite } A$
shows $(\mu_L (\text{card } A) \ T, \mu_L (\text{card } A) \ S) \in \text{less-than } <*\text{lex}*> \text{less-than}$

proof –

have $[\text{simp}]$: $(\text{atms-of-msu } (\text{clauses } T) \cup \text{atm-of ' lits-of } (\text{trail } T))$
 $= (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S))$
using learnST $n\text{-d}$ **by** (elim learnE) *auto*

then have $\text{card } (\text{atms-of-msu } (\text{clauses } T) \cup \text{atm-of ' lits-of } (\text{trail } T))$
 $= \text{card } (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S))$
by (auto intro! : card-mono)

then have 3 : $(3::\text{nat}) \wedge \text{card } (\text{atms-of-msu } (\text{clauses } T) \cup \text{atm-of ' lits-of } (\text{trail } T))$
 $= 3 \wedge \text{card } (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S))$
by (auto intro : power-mono)

moreover have $\text{conflicting-bj-clss } S \subseteq \text{conflicting-bj-clss } T$
using learnST $n\text{-d}$ **by** (simp add : $\text{learn-conflicting-increasing}$)

moreover have $\text{conflicting-bj-clss } S \neq \text{conflicting-bj-clss } T$
using learnST

proof (elim learnE , goal-cases)

case $(1 \ C)$ **note** $\text{clss-S} = \text{this}(1)$ **and** $\text{atms-C} = \text{this}(2)$ **and** $\text{inv} = \text{this}(3)$ **and** $T = \text{this}(4)$

then obtain $F \ K \ F' \ C' \ L$ **where**

tr-S : $\text{trail } S = F' @ \text{Marked } K \ () \ \# \ F$ **and**

C : $C = C' + \{\#L\# \}$ **and**

F : $F \models_{\text{as}} C \text{Not } C'$ **and**

$C\text{-S}$: $C' + \{\#L\# \} \notin \text{clauses } S$

by *blast*

moreover have $\text{distinct-mset } C \neg \text{tautology } C$ **using** inv **by** *blast+*

ultimately have $C' + \{\#L\# \} \in \text{conflicting-bj-clss } T$

using $T \ n\text{-d}$ **unfolding** $\text{conflicting-bj-clss-def}$ **by** *fastforce*

moreover have $C' + \{\#L\# \} \notin \text{conflicting-bj-clss } S$

using $C\text{-S}$ **unfolding** $\text{conflicting-bj-clss-def}$ **by** *auto*

ultimately show *?case* **by** *blast*

qed

moreover have fin-T : $\text{finite } (\text{conflicting-bj-clss } T)$

using learnST **by** *induction* (auto simp add : $\text{conflicting-bj-clss-add-cl}_{NOT}$)

ultimately have $\text{card } (\text{conflicting-bj-clss } T) \geq \text{card } (\text{conflicting-bj-clss } S)$
using *card-mono* **by** *blast*

moreover

have fin' : $\text{finite } (\text{atms-of-msu } (\text{clauses } T) \cup \text{atm-of } \text{' lits-of } (\text{trail } T))$
by *auto*
have 1 : $\text{atms-of-ms } (\text{conflicting-bj-clss } T) \subseteq \text{atms-of-msu } (\text{clauses } T)$
unfolding *conflicting-bj-clss-def* *atms-of-ms-def* **by** *auto*
have 2 : $\bigwedge x. x \in \text{conflicting-bj-clss } T \implies \neg \text{tautology } x \wedge \text{distinct-mset } x$
unfolding *conflicting-bj-clss-def* **by** *auto*
have T : $\text{conflicting-bj-clss } T$
 $\subseteq \text{build-all-simple-clss } (\text{atms-of-msu } (\text{clauses } T) \cup \text{atm-of } \text{' lits-of } (\text{trail } T))$
by *standard* (*meson* $1\ 2\ \text{fin}'\ \langle \text{finite } (\text{conflicting-bj-clss } T) \rangle\ \text{build-all-simple-clss-mono}$
distinct-mset-set-def *simplified-in-build-all* *subsetCE* *sup.coboundedI1*)

moreover

then have $\#$: $3 \wedge \text{card } (\text{atms-of-msu } (\text{clauses } T) \cup \text{atm-of } \text{' lits-of } (\text{trail } T))$
 $\geq \text{card } (\text{conflicting-bj-clss } T)$
by (*meson* *Nat.le-trans* *build-all-simple-clss-card* *build-all-simple-clss-finite* *card-mono* fin')
have $\text{atms-of-msu } (\text{clauses } T) \cup \text{atm-of } \text{' lits-of } (\text{trail } T) \subseteq A$
using *learnE*[*OF* *learnST*] *A* **by** *simp*
then have $3 \wedge (\text{card } A) \geq \text{card } (\text{conflicting-bj-clss } T)$
using $\#$ $\text{fin}-A$ **by** (*meson* *build-all-simple-clss-card* *build-all-simple-clss-finite*
build-all-simple-clss-mono *calculation*(2) *card-mono* *dual-order.trans*)

ultimately show *?thesis*

using *psubset-card-mono*[*OF* $\text{fin}-T$]
unfolding *less-than-iff* *lex-prod-def* **by** *clarify*
(*meson* $\langle \text{conflicting-bj-clss } S \neq \text{conflicting-bj-clss } T \rangle$
 $\langle \text{conflicting-bj-clss } S \subseteq \text{conflicting-bj-clss } T \rangle$
diff-less-mono2 *le-less-trans* *not-le* *psubsetI*)

qed

We have to assume the following:

- *inv* S : the invariant holds in the initial state.
- A is a (finite *finite* A) superset of the literals in the trail $\text{atm-of } \text{' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ and in the clauses $\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$. This can be the set of all the literals in the starting set of clauses.
- *no-dup* $(\text{trail } S)$: no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} **where**

$\mu_{CDCL} A\ T \equiv ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T),$
 $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A))\ T, \text{card } (\text{set-mset } (\text{clauses } T)))$

lemma *cdcl_{NOT}-decreasing-measure*:

assumes

cdcl_{NOT} $S\ T$ **and**
inv: *inv* S **and**
atm-clss: $\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**
atm-lits: $\text{atm-of } \text{' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**
n-d: *no-dup* $(\text{trail } S)$ **and**
fin-A: *finite* A

shows $(\mu_{CDCL} A\ T, \mu_{CDCL} A\ S)$
 $\in \text{less-than } \langle *lex* \rangle (\text{less-than } \langle *lex* \rangle \text{less-than})$

```

using assms(1)
proof induction
  case (c-dpll-bj T)
  from dpll-bj-trail-mes-decreasing-prop[OF this(1) inv atm-clss atm-lits n-d fin-A]
  show ?case unfolding  $\mu_{CDCL-def}$ 
    by (meson in-lex-prod less-than-iff)
next
  case (c-learn T) note learn = this(1)
  then have S: trail S = trail T
    using inv atm-clss atm-lits n-d fin-A
    by (elim learnE) auto
  show ?case
    using learn- $\mu_L$ -decrease[OF learn - ] atm-clss atm-lits fin-A n-d unfolding  $\mu_{CDCL-def}$  by auto
next
  case (c-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-msu (clauses S)  $\subseteq$  atms-of-ms A  $\wedge$  atm-of ' lits-of (trail S)  $\subseteq$  atms-of-ms A
 $\wedge$  no-dup (trail S)
 $\wedge$  inv S)
 $\wedge$  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 $\mu_C' :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_C' A T \equiv \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$

definition $\mu_{CDCL}' :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_{CDCL}' A T \equiv$
 $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A T) * (1 + 3^{\text{card} (\text{atms-of-ms } A)}) * 2$
 $+ \text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) T * 2$
 $+ \text{card} (\text{set-mset} (\text{clauses } T))$

lemma *cdcl_{NOT}-decreasing-measure'*:

```

assumes
  cdclNOT S T and
  inv: inv S and
  atms-clss: atms-of-msu (clauses S)  $\subseteq$  atms-of-ms A and
  atms-trail: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-ms A and
  n-d: no-dup (trail S) and
  fin-A: 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 μ_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 $(1 + 3 \wedge \text{card } (\text{atms-of-ms } A))$]
have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T) *$
 $(1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) + (1 + 3 \wedge \text{card } (\text{atms-of-ms } A))$
 $\leq ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A S)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A))$
unfolding Nat.add-mult-distrib
by presburger
moreover
have cl-T-S: clauses $T = \text{clauses } S$
using dpll-bj.hyps inv dpll-bj-clauses **by** auto
have conflicting-bj-clss-yet $(\text{card } (\text{atms-of-ms } A)) S < 1 + 3 \wedge \text{card } (\text{atms-of-ms } A)$
by simp
ultimately have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) + \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T$
 $< ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A S) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A))$
by linarith
then have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A))$
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T$
 $< ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A S)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A))$
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) S$
by linarith
then have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T * 2$
 $< ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A S)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
 $+ \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) S * 2$
by linarith
then show ?case **unfolding** μ_{CDCL}' -def cl-T-S **by** presburger
next
case (learn $C F' K F C' L T$) **note** clss-S-C = this(1) **and** atms-C = this(2) **and** dist = this(3)
and tauto = this(4) **and** learn-restr = this(5) **and** tr-S = this(6) **and** $C' = \text{this}(7)$ **and**
 $F-C = \text{this}(8)$ **and** $C\text{-new} = \text{this}(9)$ **and** $T = \text{this}(10)$
have insert C (conflicting-bj-clss S) \subseteq build-all-simple-clss (atms-of-ms A)
proof –
have $C \in \text{build-all-simple-clss } (\text{atms-of-ms } A)$
by (metis (no-types, hide-lams) Un-subset-iff atms-of-ms-finite build-all-simple-clss-mono
contra-subsetD dist distinct-mset-not-tautology-implies-in-build-all-simple-clss
dual-order.trans fin-A atms-C atms-clss atms-trail tauto)
moreover have conflicting-bj-clss $S \subseteq \text{build-all-simple-clss } (\text{atms-of-ms } A)$
unfolding conflicting-bj-clss-def
proof
fix $x :: 'v$ literal multiset
assume $x \in \{C + \{\#L\# \} \mid C L. C + \{\#L\# \} \in \# \text{ clauses } S$
 $\wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$
 $\wedge (\exists F' K F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} C \text{Not } C)\}$
then have $\exists m l. x = m + \{\#l\# \} \wedge m + \{\#l\# \} \in \# \text{ clauses } S$
 $\wedge \text{distinct-mset } (m + \{\#l\# \}) \wedge \neg \text{tautology } (m + \{\#l\# \})$

```

     $\wedge (\exists ms\ l\ msa. \text{ trail } S = ms @ \text{ Marked } l () \# msa \wedge msa \models_{as} C \text{Not } m)$ 
  by blast
then show  $x \in \text{build-all-simple-clss } (\text{atms-of-ms } A)$ 
  by (meson atms-clss atms-of-atms-of-ms-mono atms-of-ms-finite build-all-simple-clss-mono
    distinct-mset-not-tautology-implies-in-build-all-simple-clss fin-A finite-subset
    mem-set-mset-iff set-rev-mp)
qed
ultimately show ?thesis
  by auto
qed
then have  $\text{card } (\text{insert } C (\text{conflicting-bj-clss } S)) \leq 3 \wedge (\text{card } (\text{atms-of-ms } A))$ 
  by (meson Nat.le-trans atms-of-ms-finite build-all-simple-clss-card build-all-simple-clss-finite
    card-mono fin-A)
moreover have [simp]:  $\text{card } (\text{insert } C (\text{conflicting-bj-clss } S)) = \text{Suc } (\text{card } ((\text{conflicting-bj-clss } S)))$ 
  = Suc (card ((conflicting-bj-clss S)))
  by (metis (no-types) C' C-new card-insert-if conflicting-bj-clss-incl-clauses contra-subsetD
    finite-conflicting-bj-clss mem-set-mset-iff)
moreover have [simp]:  $\text{conflicting-bj-clss } (\text{add-cl}_{NOT} C S) = \text{conflicting-bj-clss } S \cup \{C\}$ 
  using dist tauto F-C n-d by (subst conflicting-bj-clss-add-clNOT)
  (force simp add: ac-simps C' tr-S)+
ultimately have [simp]:  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) S = \text{Suc } (\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S))$ 
  by simp
have 1:  $\text{clauses } T = \text{clauses } (\text{add-cl}_{NOT} C S)$  using T by auto
have 2:  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T = \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S)$ 
  using T unfolding conflicting-bj-clss-def by auto
have 3:  $\mu_{C'} A T = \mu_{C'} A (\text{add-cl}_{NOT} C S)$ 
  using T unfolding  $\mu_{C'}$ -def by auto
have  $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_{C'} A (\text{add-cl}_{NOT} C S))$ 
  *  $(1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$ 
  =  $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_{C'} A S)$ 
  *  $(1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$ 
  using n-d unfolding  $\mu_{C'}$ -def by auto
moreover
  have  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S)$ 
    * 2
    +  $\text{card } (\text{set-mset } (\text{clauses } (\text{add-cl}_{NOT} C S)))$ 
    <  $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) S * 2$ 
    +  $\text{card } (\text{set-mset } (\text{clauses } S))$ 
    by (simp add: C' C-new n-d)
ultimately show ?case unfolding  $\mu_{CDCL}$ '-def 1 2 3 by presburger
next
case (forgetNOT C T) note T = this(4)
have [simp]:  $\mu_{C'} A (\text{remove-cl}_{NOT} C S) = \mu_{C'} A S$ 
  unfolding  $\mu_{C'}$ -def by auto
have forgetNOT S T
  apply (rule forgetNOT.intros) using forgetNOT by auto
then have conflicting-bj-clss T = conflicting-bj-clss S
  using do-not-forget-before-backtrack-rule-clause-learned-clause-untouched by blast
moreover have  $\text{card } (\text{set-mset } (\text{clauses } T)) < \text{card } (\text{set-mset } (\text{clauses } S))$ 
  by (metis T card-Diff1-less clauses-remove-clNOT finite-set-mset forgetNOT.hyps(2)
    mem-set-mset-iff order-refl set-mset-minus-replicate-mset(1) state-eqNOT-clauses)
ultimately show ?case unfolding  $\mu_{CDCL}$ '-def
  by (metis (no-types) T  $\mu_{C'} A (\text{remove-cl}_{NOT} C S) = \mu_{C'} A S$  add-le-cancel-left)

```


μ_C '-def not-le state-eq_{NOT}-trail)

qed

lemma *cdcl_{NOT}-clauses-bound*:

assumes

cdcl_{NOT} S T and

inv S and

atms-of-msu (clauses S) \subseteq A and

atm-of '(lits-of (trail S)) \subseteq A and

n-d: no-dup (trail S) and

fin-A[simp]: finite A

shows *set-mset (clauses T) \subseteq set-mset (clauses S) \cup build-all-simple-clss A*

using *assms*

proof (*induction rule: cdcl_{NOT}-learn-all-induct*)

case *dpll-bj*

then show *?case using dpll-bj-clauses by simp*

next

case *forget_{NOT}*

then show *?case using clauses-remove-cl_{NOT} unfolding state-eq_{NOT}-def by auto*

next

case (*learn C F K d F' C' L*) **note** *atms-C = this(2) and dist = this(3) and tauto = this(4) and*

T = this(10) and atms-clss-S = this(12) and atms-trail-S = this(13)

have *atms-of C \subseteq A*

using *atms-C atms-clss-S atms-trail-S by auto*

then have *build-all-simple-clss (atms-of C) \subseteq build-all-simple-clss A*

by (*simp add: build-all-simple-clss-mono*)

then have *C \in build-all-simple-clss A*

using *finite dist tauto*

by (*auto dest: distinct-mset-not-tautology-implies-in-build-all-simple-clss*)

then show *?case using T n-d by auto*

qed

lemma *rtrancpl-cdcl_{NOT}-clauses-bound*:

assumes

*cdcl_{NOT}** S T and*

inv S and

atms-of-msu (clauses S) \subseteq A and

atm-of '(lits-of (trail S)) \subseteq A and

n-d: no-dup (trail S) and

finite: finite A

shows *set-mset (clauses T) \subseteq set-mset (clauses S) \cup build-all-simple-clss A*

using *assms(1–5)*

proof *induction*

case *base*

then show *?case by simp*

next

case (*step T U*) **note** *st = this(1) and cdcl_{NOT} = this(2) and IH = this(3)[OF this(4–7)] and*

inv = this(4) and atms-clss-S = this(5) and atms-trail-S = this(6) and finite-clss-S = this(7)

have *inv T*

using *rtrancpl-cdcl_{NOT}-inv st inv by blast*

moreover have *atms-of-msu (clauses T) \subseteq A and atm-of ' lits-of (trail T) \subseteq A*

using *rtrancpl-cdcl_{NOT}-trail-clauses-bound[OF st] inv atms-clss-S atms-trail-S n-d by blast+*

moreover have *no-dup (trail T)*

using *rtrancpl-cdcl_{NOT}-no-dup[OF st <inv S> n-d] by simp*

ultimately have *set-mset (clauses U) \subseteq set-mset (clauses T) \cup build-all-simple-clss A*

```

    using cdclNOT finite n-d by (auto simp: cdclNOT-clauses-bound)
  then show ?case using IH by auto
qed

```

lemma *rtrancp-cdcl_{NOT}-card-clauses-bound*:

```

assumes
  cdclNOT** S T and
  inv S and
  atms-of-msu (clauses S) ⊆ A and
  atm-of '(lits-of (trail S)) ⊆ A and
  n-d: no-dup (trail S) and
  finite: finite A
shows card (set-mset (clauses T)) ≤ card (set-mset (clauses S)) + 3 ^ (card A)
using rtrancp-cdclNOT-clauses-bound[OF assms] finite by (meson Nat.le-trans
  build-all-simple-clss-card build-all-simple-clss-finite card-Un-le card-mono finite-UnI
  finite-set-mset nat-add-left-cancel-le)

```

lemma *rtrancp-cdcl_{NOT}-card-clauses-bound'*:

```

assumes
  cdclNOT** S T and
  inv S and
  atms-of-msu (clauses S) ⊆ A and
  atm-of '(lits-of (trail S)) ⊆ A and
  n-d: no-dup (trail S) and
  finite: finite A
shows card {C | C. C ∈# clauses T ∧ (tautology C ∨ ¬distinct-mset C)}
  ≤ card {C | C. C ∈# clauses S ∧ (tautology C ∨ ¬distinct-mset C)} + 3 ^ (card A)
  (is card ?T ≤ card ?S + -)
using rtrancp-cdclNOT-clauses-bound[OF assms] finite
proof -
  have ?T ⊆ ?S ∪ build-all-simple-clss A
    using rtrancp-cdclNOT-clauses-bound[OF assms] by force
  then have card ?T ≤ card (?S ∪ build-all-simple-clss A)
    using finite by (simp add: assms(5) build-all-simple-clss-finite card-mono)
  then show ?thesis
    by (meson le-trans build-all-simple-clss-card card-Un-le local.finite nat-add-left-cancel-le)
qed

```

lemma *rtrancp-cdcl_{NOT}-card-simple-clauses-bound*:

```

assumes
  cdclNOT** S T and
  inv S and
  atms-of-msu (clauses S) ⊆ A and
  atm-of '(lits-of (trail S)) ⊆ A and
  n-d: no-dup (trail S) and
  finite: finite A
shows card (set-mset (clauses T))
  ≤ card {C. C ∈# clauses S ∧ (tautology C ∨ ¬distinct-mset C)} + 3 ^ (card A)
  (is card ?T ≤ card ?S + -)
using rtrancp-cdclNOT-clauses-bound[OF assms] finite
proof -
  have ∧x. x ∈# clauses T ⇒ ¬tautology x ⇒ distinct-mset x ⇒ x ∈ build-all-simple-clss A
    using rtrancp-cdclNOT-clauses-bound[OF assms] by (metis (no-types, hide-lams) Un-iff assms(3)
      atms-of-atms-of-ms-mono build-all-simple-clss-mono contra-subsetD)

```

distinct-mset-not-tautology-implies-in-build-all-simple-clss local.finite mem-set-mset-iff subset-trans)
then have *set-mset (clauses T) ⊆ ?S ∪ build-all-simple-clss A*
using *rtrancpl-cdcl_{NOT}-clauses-bound[OF assms]* **by** *auto*
then have *card(set-mset (clauses T)) ≤ card (?S ∪ build-all-simple-clss A)*
using *finite by (simp add: assms(5) build-all-simple-clss-finite card-mono)*
then show *?thesis*
by *(meson le-trans build-all-simple-clss-card card-Un-le local.finite nat-add-left-cancel-le)*
qed

definition $\mu_{CDCL}'\text{-bound} :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**
 $\mu_{CDCL}'\text{-bound } A \ S =$
 $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
 $+ 2 * 3 \wedge (\text{card } (\text{atms-of-ms } A))$
 $+ \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } (\text{atms-of-ms } A))$

lemma $\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}[simp]:$
 $\mu_{CDCL}'\text{-bound } A \ (\text{reduce-trail-to}_{NOT} \ M \ S) = \mu_{CDCL}'\text{-bound } A \ S$
unfolding $\mu_{CDCL}'\text{-bound-def}$ **by** *auto*

lemma $rtrancpl-cdcl_{NOT}-\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}:$
assumes
 $cdcl_{NOT}^{**} \ S \ T$ **and**
 $inv \ S$ **and**
 $\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**
 $n\text{-d: no-dup } (\text{trail } S)$ **and**
 $\text{finite: finite } (\text{atms-of-ms } A)$ **and**
 $U: U \sim \text{reduce-trail-to}_{NOT} \ M \ T$
shows $\mu_{CDCL}' \ A \ U \leq \mu_{CDCL}'\text{-bound } A \ S$

proof –
have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' \ A \ U)$
 $\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
by *auto*
then have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' \ A \ U)$
 $* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
 $\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
using *mult-le-mono1* **by** *blast*
moreover
have $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) \ T * 2 \leq 2 * 3 \wedge \text{card } (\text{atms-of-ms } A)$
by *linarith*
moreover have $\text{card } (\text{set-mset } (\text{clauses } U))$
 $\leq \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge \text{card } (\text{atms-of-ms } A)$
using *rtrancpl-cdcl_{NOT}-card-simple-clauses-bound[OF assms(1-6)] U* **by** *auto*
ultimately show *?thesis*
unfolding $\mu_{CDCL}'\text{-def}$ $\mu_{CDCL}'\text{-bound-def}$ **by** *linarith*
qed

lemma $rtrancpl-cdcl_{NOT}-\mu_{CDCL}'\text{-bound}:$
assumes
 $cdcl_{NOT}^{**} \ S \ T$ **and**
 $inv \ S$ **and**
 $\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**
 $n\text{-d: no-dup } (\text{trail } S)$ **and**

finite: *finite* (*atms-of-ms* *A*)
shows $\mu_{CDCL}' A T \leq \mu_{CDCL}'\text{-bound } A S$
proof –
have $\mu_{CDCL}' A (\text{reduce-trail-to}_{NOT} (\text{trail } T) T) = \mu_{CDCL}' A T$
unfolding $\mu_{CDCL}'\text{-def } \mu_C'\text{-def conflicting-bj-clss-def}$ **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-msu (clauses S) \subseteq atms-of-ms A and
atm-of '(lits-of (trail S)) \subseteq atms-of-ms A and
n-d: no-dup (trail S) and
finite[simp]: finite (atms-of-ms A)
shows $\mu_{CDCL}'\text{-bound } A T \leq \mu_{CDCL}'\text{-bound } A S$
proof –
have $\{C. C \in \# \text{ clauses } T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$
 $\subseteq \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$ (**is** *?T \subseteq ?S*)
proof (*rule Set.subsetI*)
fix *C* **assume** *C* \in *?T*
then have *C-T: C \in # clauses T and t-d: tautology C \vee \neg distinct-mset C*
by *auto*
then have *C \notin build-all-simple-clss (atms-of-ms A)*
by (*auto dest: build-all-simple-clssE*)
then show *C \in ?S*
using *C-T rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] t-d* **by** *force*
qed
then have $\text{card } \{C. C \in \# \text{ clauses } T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} \leq$
 $\text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$
by (*simp add: card-mono*)
then show *?thesis*
unfolding $\mu_{CDCL}'\text{-bound-def}$ **by** *auto*
qed

end — end of *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt*

14.7 CDCL with restarts

14.7.1 Definition

locale *restart-ops* =
fixes
cdcl_{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 trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}

```

propagate-conds inv backjump-conds learn-cond forget-cond
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  learn-cond forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

lemma cdclNOT-iff-cdclNOT-raw-restart-no-restarts:
  cdclNOT S T  $\longleftrightarrow$  restart-ops.cdclNOT-raw-restart cdclNOT ( $\lambda$ - . False) S T
  (is ?C S T  $\longleftrightarrow$  ?R S T)
proof
  fix S T
  assume ?C S T
  then show ?R S T by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
next
  fix S T
  assume ?R S T
  then show ?C S T
    apply (cases rule: restart-ops.cdclNOT-raw-restart.cases)
    using (?R S T) by fast+
qed

lemma cdclNOT-cdclNOT-raw-restart:
  cdclNOT S T  $\implies$  restart-ops.cdclNOT-raw-restart cdclNOT restart S T
  by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
end

```

14.7.2 Increasing restarts

To add restarts we need some assumptions on the predicate (called *cdcl_{NOT}* here):

- a function f that is strictly monotonic. The first step is actually only used as a restart to clean the state (e.g. to ensure that the trail is empty). Then we assume that $(1::'a) \leq f$ n for $(1::'a) \leq n$: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...
- a measure μ : it should decrease under the assumptions *bound-inv*, whenever a *cdcl_{NOT}* or a *restart* is done. A parameter is given to μ : for conflict- driven clause learning, it is an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.
- we also assume that the measure decrease after any *cdcl_{NOT}* step.
- an invariant on the states *cdcl_{NOT}-inv* that also holds after restarts.
- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ -*bound* taking the same parameter as μ and the initial state of the considered *cdcl_{NOT}* chain.

```

locale cdclNOT-increasing-restarts-ops =
  restart-ops cdclNOT restart for
    restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
    cdclNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool +
fixes
  f :: nat  $\Rightarrow$  nat and
  bound-inv :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  bool and
   $\mu$  :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat and
  cdclNOT-inv :: 'st  $\Rightarrow$  bool and
   $\mu$ -bound :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat
assumes
  f: unbounded f and
  f-ge-1:  $\bigwedge n. n \geq 1 \Rightarrow f\ n \neq 0$  and
  bound-inv:  $\bigwedge A\ S\ T. \text{cdcl}_{NOT}\text{-inv}\ S \Rightarrow \text{bound-inv}\ A\ S \Rightarrow \text{cdcl}_{NOT}\ S\ T \Rightarrow \text{bound-inv}\ A\ T$  and
  cdclNOT-measure:  $\bigwedge A\ S\ T. \text{cdcl}_{NOT}\text{-inv}\ S \Rightarrow \text{bound-inv}\ A\ S \Rightarrow \text{cdcl}_{NOT}\ S\ T \Rightarrow \mu\ A\ T < \mu$ 
A S and
  measure-bound2:  $\bigwedge A\ T\ U. \text{cdcl}_{NOT}\text{-inv}\ T \Rightarrow \text{bound-inv}\ A\ T \Rightarrow \text{cdcl}_{NOT}^{**}\ T\ U$ 
 $\Rightarrow \mu\ A\ U \leq \mu\text{-bound}\ A\ T$  and
  measure-bound4:  $\bigwedge A\ T\ U. \text{cdcl}_{NOT}\text{-inv}\ T \Rightarrow \text{bound-inv}\ A\ T \Rightarrow \text{cdcl}_{NOT}^{**}\ T\ U$ 
 $\Rightarrow \mu\text{-bound}\ A\ U \leq \mu\text{-bound}\ A\ T$  and
  cdclNOT-restart-inv:  $\bigwedge A\ U\ V. \text{cdcl}_{NOT}\text{-inv}\ U \Rightarrow \text{restart}\ U\ V \Rightarrow \text{bound-inv}\ A\ U \Rightarrow \text{bound-inv}$ 
A V
and
  exists-bound:  $\bigwedge R\ S. \text{cdcl}_{NOT}\text{-inv}\ R \Rightarrow \text{restart}\ R\ S \Rightarrow \exists A. \text{bound-inv}\ A\ S$  and
  cdclNOT-inv:  $\bigwedge S\ T. \text{cdcl}_{NOT}\text{-inv}\ S \Rightarrow \text{cdcl}_{NOT}\ S\ T \Rightarrow \text{cdcl}_{NOT}\text{-inv}\ T$  and
  cdclNOT-inv-restart:  $\bigwedge S\ T. \text{cdcl}_{NOT}\text{-inv}\ S \Rightarrow \text{restart}\ S\ T \Rightarrow \text{cdcl}_{NOT}\text{-inv}\ T$ 
begin

lemma cdclNOT-cdclNOT-inv:
assumes
  (cdclNOT  $\widetilde{\sim} n$ ) S T and
  cdclNOT-inv S
shows cdclNOT-inv T
using assms by (induction n arbitrary: T) (auto intro: bound-inv cdclNOT-inv)

lemma cdclNOT-bound-inv:
assumes
  (cdclNOT  $\widetilde{\sim} n$ ) S T and
  cdclNOT-inv S
  bound-inv A S
shows bound-inv A T
using assms by (induction n arbitrary: T) (auto intro: bound-inv cdclNOT-cdclNOT-inv)

lemma rtrancplp-cdclNOT-cdclNOT-inv:
assumes
  cdclNOT** S T and
  cdclNOT-inv S
shows cdclNOT-inv T
using assms by induction (auto intro: cdclNOT-inv)

lemma rtrancplp-cdclNOT-bound-inv:
assumes
  cdclNOT** S T and
  bound-inv A S and
  cdclNOT-inv S

```

shows $\text{bound-inv } A \ T$
using *assms* **by** *induction* (*auto intro:bound-inv rtranclp-cdcl_{NOT}-cdcl_{NOT}-inv*)

lemma *cdcl_{NOT}-comp-n-le*:

assumes
 $(\text{cdcl}_{\text{NOT}} \sim (\text{Suc } n)) \ S \ T$ **and**
 $\text{bound-inv } A \ S$
 $\text{cdcl}_{\text{NOT-inv}} \ S$

shows $\mu \ A \ T < \mu \ A \ S - n$

using *assms*

proof (*induction n arbitrary: T*)

case 0

then show ?case **using** *cdcl_{NOT}-measure* **by** *auto*

next

case (*Suc n*) **note** $IH = \text{this}(1)[OF - \text{this}(3) \ \text{this}(4)]$ **and** $S-T = \text{this}(2)$ **and** $b\text{-inv} = \text{this}(3)$ **and** $c\text{-inv} = \text{this}(4)$

obtain $U :: 'st$ **where** $S-U: (\text{cdcl}_{\text{NOT}} \sim (\text{Suc } n)) \ S \ U$ **and** $U-T: \text{cdcl}_{\text{NOT}} \ U \ T$ **using** $S-T$ **by** *auto*
then have $\mu \ A \ U < \mu \ A \ S - n$ **using** $IH[of \ U]$ **by** *simp*

moreover

have $\text{bound-inv } A \ U$

using $S-U \ b\text{-inv} \ \text{cdcl}_{\text{NOT-bound-inv}} \ c\text{-inv}$ **by** *blast*

then have $\mu \ A \ T < \mu \ A \ U$ **using** $\text{cdcl}_{\text{NOT-measure}}[OF - - \ U-T] \ S-U \ c\text{-inv} \ \text{cdcl}_{\text{NOT}} \ \text{cdcl}_{\text{NOT-inv}}$

by *auto*

ultimately show ?case **by** *linarith*

qed

lemma *wf-cdcl_{NOT}*:

$\text{wf } \{(T, S). \ \text{cdcl}_{\text{NOT}} \ S \ T \wedge \text{cdcl}_{\text{NOT-inv}} \ S \wedge \text{bound-inv } A \ S\}$ (**is** $\text{wf } ?A$)

apply (*rule wfP-if-measure2[of - - $\mu \ A$]*)

using *cdcl_{NOT}-comp-n-le*[*of 0 - - A*] **by** *auto*

lemma *rtranclp-cdcl_{NOT}-measure*:

assumes

$\text{cdcl}_{\text{NOT}}^{**} \ S \ T$ **and**

$\text{bound-inv } A \ S$ **and**

$\text{cdcl}_{\text{NOT-inv}} \ S$

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** $IH = \text{this}(3)[OF \ \text{this}(4) \ \text{this}(5)]$ **and** $st = \text{this}(1)$ **and** $\text{cdcl}_{\text{NOT}} = \text{this}(2)$ **and** $b\text{-inv} = \text{this}(4)$ **and** $c\text{-inv} = \text{this}(5)$

have $\text{bound-inv } A \ T$

by (*meson cdcl_{NOT}-bound-inv rtranclp-imp-relpoup st step.prem*s)

moreover have $\text{cdcl}_{\text{NOT-inv}} \ T$

using $c\text{-inv} \ rtranclp\text{-cdcl}_{\text{NOT}} \ \text{cdcl}_{\text{NOT-inv}} \ st$ **by** *blast*

ultimately have $\mu \ A \ U < \mu \ A \ T$ **using** $\text{cdcl}_{\text{NOT-measure}}[OF - - \ \text{cdcl}_{\text{NOT}}]$ **by** *auto*

then show ?case **using** IH **by** *linarith*

qed

lemma *cdcl_{NOT}-comp-bounded*:

assumes

$\text{bound-inv } A \ S$ **and** $\text{cdcl}_{\text{NOT-inv}} \ S$ **and** $m \geq 1 + \mu \ A \ S$

shows $\neg(\text{cdcl}_{NOT} \sim m) S T$
using *assms cdcl_{NOT}-comp-n-le[of m-1 S T A]* **by** *fastforce*

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

inductive *cdcl_{NOT}-restart* **where**

restart-step: $(\text{cdcl}_{NOT} \sim m) S T \implies m \geq f n \implies \text{restart } T U$

$\implies \text{cdcl}_{NOT}\text{-restart } (S, n) (U, \text{Suc } n) \mid$

restart-full: $\text{full1 } \text{cdcl}_{NOT} S T \implies \text{cdcl}_{NOT}\text{-restart } (S, n) (T, \text{Suc } n)$

lemmas *cdcl_{NOT}-with-restart-induct* = *cdcl_{NOT}-restart.induct[split-format(complete),*
OF cdcl_{NOT}-increasing-restarts-ops-axioms]

lemma *cdcl_{NOT}-restart-cdcl_{NOT}-raw-restart*:

cdcl_{NOT}-restart $S T \implies \text{cdcl}_{NOT}\text{-raw-restart}^{**} (fst S) (fst T)$

proof (*induction rule: cdcl_{NOT}-restart.induct*)

case (*restart-step* $m S T n U$)

then have *cdcl_{NOT}*** $S T$ **by** (*meson relpowp-imp-rtrancp*)

then have *cdcl_{NOT}-raw-restart*** $S T$ **using** *cdcl_{NOT}-raw-restart.intros(1)*

rtrancp-mono[of cdcl_{NOT} cdcl_{NOT}-raw-restart] **by** *blast*

moreover have *cdcl_{NOT}-raw-restart* $T U$

using (*restart* $T U$) *cdcl_{NOT}-raw-restart.intros(2)* **by** *blast*

ultimately show *?case* **by** *auto*

next

case (*restart-full* $S T$)

then have *cdcl_{NOT}*** $S T$ **unfolding** *full1-def* **by** *auto*

then show *?case* **using** *cdcl_{NOT}-raw-restart.intros(1)*

rtrancp-mono[of cdcl_{NOT} cdcl_{NOT}-raw-restart] **by** *auto*

qed

lemma *cdcl_{NOT}-with-restart-bound-inv*:

assumes

cdcl_{NOT}-restart $S T$ **and**

bound-inv $A (fst S)$ **and**

cdcl_{NOT}-inv $(fst S)$

shows *bound-inv* $A (fst T)$

using *assms* **apply** (*induction rule: cdcl_{NOT}-restart.induct*)

prefer 2 **apply** (*metis rtrancp-unfold fstI full1-def rtrancp-cdcl_{NOT}-bound-inv*)

by (*metis cdcl_{NOT}-bound-inv cdcl_{NOT}-cdcl_{NOT}-inv cdcl_{NOT}-restart-inv fst-conv*)

lemma *cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*:

assumes

cdcl_{NOT}-restart $S T$ **and**

cdcl_{NOT}-inv $(fst S)$

shows *cdcl_{NOT}-inv* $(fst T)$

using *assms* **apply** *induction*

apply (*metis cdcl_{NOT}-cdcl_{NOT}-inv cdcl_{NOT}-inv-restart fst-conv*)

apply (*metis fstI full-def full-unfold 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 *rtrancpl-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* *rtrancpl-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

for

trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**

clauses :: 'st \Rightarrow 'v clauses **and**

prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**

tl-trail :: 'st \Rightarrow 'st **and**

add-cl_{NOT} remove-cl_{NOT} :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**

f :: nat \Rightarrow nat **and**

restart :: 'st \Rightarrow 'st \Rightarrow bool **and**

bound-inv :: 'bound \Rightarrow 'st \Rightarrow bool **and**

μ :: 'bound \Rightarrow 'st \Rightarrow nat **and**

cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool **and**

cdcl_{NOT}-inv :: 'st \Rightarrow bool **and**

μ -bound :: 'bound \Rightarrow 'st \Rightarrow nat +

assumes

measure-bound: $\bigwedge A T V n. cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A T$

$\implies cdcl_{NOT}\text{-restart } (T, n) (V, Suc\ n) \implies \mu\ A\ V \leq \mu\text{-bound } A\ T$ **and**

cdcl_{NOT}-raw-restart- μ -bound:

$cdcl_{NOT}\text{-restart } (T, a) (V, b) \implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A\ T$

$\implies \mu\text{-bound } A\ V \leq \mu\text{-bound } A\ T$

begin

lemma *rtrancpl-cdcl_{NOT}-raw-restart- μ -bound:*

*cdcl_{NOT}-restart** (T, a) (V, b) $\implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A\ T$*

$\implies \mu\text{-bound } A\ V \leq \mu\text{-bound } A\ T$

apply (*induction rule: rtrancpl-induct2*)

apply *simp*

by (*metis cdcl_{NOT}-raw-restart- μ -bound dual-order.trans fst-conv*

rtrancpl-cdcl_{NOT}-with-restart-bound-inv rtrancpl-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv)

lemma *cdcl_{NOT}-raw-restart-measure-bound:*

cdcl_{NOT}-restart (T, a) (V, b) $\implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A\ T$

$\implies \mu\ A\ V \leq \mu\text{-bound } A\ T$

apply (*cases rule: cdcl_{NOT}-restart.cases*)

apply *simp*

using *measure-bound relpowp-imp-rtrancpl* **apply** *fastforce*

by (*metis full-def full-unfold measure-bound2 prod.inject*)

```

lemma rtrancp-cdclNOT-raw-restart-measure-bound:
  cdclNOT-restart** (T, a) (V, b)  $\implies$  cdclNOT-inv T  $\implies$  bound-inv A T
   $\implies \mu$  A V  $\leq \mu$ -bound A T
apply (induction rule: rtrancp-induct2)
apply (simp add: measure-bound2)
by (metis dual-order.trans fst-conv measure-bound2 r-into-rtrancp rtrancp.rtrancp-refl
  rtrancp-cdclNOT-with-restart-bound-inv rtrancp-cdclNOT-with-restart-cdclNOT-inv
  rtrancp-cdclNOT-raw-restart- $\mu$ -bound)

lemma wf-cdclNOT-restart:
  wf {(T, S). cdclNOT-restart S T  $\wedge$  cdclNOT-inv (fst S)} (is wf ?A)
proof (rule ccontr)
assume  $\neg$  ?thesis
then obtain g where
  g:  $\bigwedge i.$  cdclNOT-restart (g i) (g (Suc i)) and
  cdclNOT-inv-g:  $\bigwedge i.$  cdclNOT-inv (fst (g i))
  unfolding wf-iff-no-infinite-down-chain by fast

have snd-g:  $\bigwedge i.$  snd (g i) = i + snd (g 0)
apply (induct-tac i)
apply simp
by (metis Suc-eq-plus1-left add.commute add.left-commute
  cdclNOT-with-restart-increasing-number g)
then have snd-g-0:  $\bigwedge i.$  i > 0  $\implies$  snd (g i) = i + snd (g 0)
by blast
have unbounded-f-g: unbounded ( $\lambda i.$  f (snd (g i)))
using f unfolding bounded-def by (metis add.commute f less-or-eq-imp-le snd-g
  not-bounded-nat-exists-larger not-le ordered-cancel-comm-monoid-diff-class.le-iff-add)

{ fix i
  have H:  $\bigwedge T$  Ta m. (cdclNOT  $\rightsquigarrow$  m) T Ta  $\implies$  no-step cdclNOT T  $\implies$  m = 0
  apply (case-tac m) apply simp by (meson relpowp-E2)
  have  $\exists$  T m. (cdclNOT  $\rightsquigarrow$  m) (fst (g i)) T  $\wedge$  m  $\geq$  f (snd (g i))
  using g[of i] apply (cases rule: cdclNOT-restart.cases)
  apply auto[]
  using g[of Suc i] f-ge-1 apply (cases rule: cdclNOT-restart.cases)
  apply (auto simp add: full1-def full-def dest: H dest: rtrancpD)
  using H Suc-leI leD by blast
} note H = this
obtain A where bound-inv A (fst (g 1))
using g[of 0] cdclNOT-inv-g[of 0] apply (cases rule: cdclNOT-restart.cases)
apply (metis One-nat-def cdclNOT-inv exists-bound fst-conv relpowp-imp-rtrancp
  rtrancp-induct)
using H[of 1] unfolding full1-def by (metis One-nat-def Suc-eq-plus1 diff-is-0-eq' diff-zero
  f-ge-1 fst-conv le-add2 relpowp-E2 snd-conv)
let ?j =  $\mu$ -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 cdclNOT-with-restart: j  $\geq$  i  $\implies$  cdclNOT-restart** (g i) (g j)
  apply (induction j)
  apply simp

```

```

    by (metis g le-Suc-eq rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
  } note cdclNOT-restart = this
have cdclNOT-inv (fst (g (Suc 0)))
  by (simp add: cdclNOT-inv-g)
have cdclNOT-restart** (fst (g 1), snd (g 1)) (fst (g j), snd (g j))
  using ⟨j > 1⟩ by (simp add: cdclNOT-restart)
have μ A (fst (g j)) ≤ μ-bound A (fst (g 1))
  apply (rule rtranclp-cdclNOT-raw-restart-measure-bound)
  using ⟨cdclNOT-restart** (fst (g 1), snd (g 1)) (fst (g j), snd (g j))⟩ apply blast
  apply (simp add: cdclNOT-inv-g)
  using ⟨bound-inv A (fst (g 1))⟩ apply simp
done
then have μ A (fst (g j)) ≤ ?j
  by auto
have inv: bound-inv A (fst (g j))
  using ⟨bound-inv A (fst (g 1))⟩ ⟨cdclNOT-inv (fst (g (Suc 0)))⟩
  ⟨cdclNOT-restart** (fst (g 1), snd (g 1)) (fst (g j), snd (g j))⟩
  rtranclp-cdclNOT-with-restart-bound-inv by auto
obtain T m where
  cdclNOT-m: (cdclNOT  $\rightsquigarrow$  m) (fst (g j)) T and
  f-m: f (snd (g j)) ≤ m
  using H[of j] by blast
have ?j < m
  using f-m j Nat.le-trans by linarith

then show False
  using ⟨μ A (fst (g j)) ≤ μ-bound A (fst (g 1))⟩
  cdclNOT-comp-bounded[OF inv cdclNOT-inv-g, of ] cdclNOT-inv-g cdclNOT-m
  ⟨?j < m⟩ by auto
qed

lemma cdclNOT-restart-steps-bigger-than-bound:
  assumes
    cdclNOT-restart S T and
    bound-inv A (fst S) and
    cdclNOT-inv (fst S) and
    f (snd S) > μ-bound A (fst S)
  shows full1 cdclNOT (fst S) (fst T)
  using assms
proof (induction rule: cdclNOT-restart.induct)
  case restart-full
  then show ?case by auto
next
  case (restart-step m S T n U) note st = this(1) and f = this(2) and bound-inv = this(4) and
    cdclNOT-inv = this(5) and μ = this(6)
  then obtain m' where m: m = Suc m' by (cases m) auto
  have μ A S - m' = 0
    using f bound-inv cdclNOT-inv μ m rtranclp-cdclNOT-raw-restart-measure-bound by fastforce
  then have False using cdclNOT-comp-n-le[of m' S T A] restart-step unfolding m by simp
  then show ?case by fast
qed

lemma rtranclp-cdclNOT-with-inv-inv-rtranclp-cdclNOT:
  assumes
    inv: cdclNOT-inv S and

```

```

  binv: bound-inv A S
shows (λS T. cdclNOT S T ∧ cdclNOT-inv S ∧ bound-inv A S)** S T ↔ cdclNOT** S T
  (is ?A** S T ↔ ?B** S T)
apply (rule iffI)
  using rtrancpl-mono[of ?A ?B] apply blast
apply (induction rule: rtrancpl-induct)
  using inv binv apply simp
by (metis (mono-tags, lifting) binv inv rtrancpl.simps rtrancpl-cdclNOT-bound-inv
  rtrancpl-cdclNOT-cdclNOT-inv)

lemma no-step-cdclNOT-restart-no-step-cdclNOT:
  assumes
    n-s: no-step cdclNOT-restart S and
    inv: cdclNOT-inv (fst S) and
    binv: bound-inv A (fst S)
  shows no-step cdclNOT (fst S)
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain T where T: cdclNOT (fst S) T
  by blast
  then obtain U where U: full (λS T. cdclNOT S T ∧ cdclNOT-inv S ∧ bound-inv A S) T U
  using wf-exists-normal-form-full[OF wf-cdclNOT, of A T] by auto
  moreover have inv-T: cdclNOT-inv T
  using ⟨cdclNOT (fst S) T⟩ cdclNOT-inv inv by blast
  moreover have b-inv-T: bound-inv A T
  using ⟨cdclNOT (fst S) T⟩ binv bound-inv inv by blast
  ultimately have full cdclNOT T U
  using rtrancpl-cdclNOT-with-inv-inv-rtrancpl-cdclNOT rtrancpl-cdclNOT-bound-inv
  rtrancpl-cdclNOT-cdclNOT-inv unfolding full-def by blast
  then have fullI cdclNOT (fst S) U
  using T full-fullI by metis
  then show False by (metis n-s prod.collapse restart-full)
qed

end

```

14.8 Merging backjump and learning

```

locale cdclNOT-merge-bj-learn-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clNOT remove-clNOT +
  decide-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT +
  forget-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT forget-cond +
  propagate-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds
  for
    trail :: 'st ⇒ ('v, unit, unit) marked-lits and
    clauses :: 'st ⇒ 'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
    tl-trail :: 'st ⇒ 'st and
    add-clNOT remove-clNOT :: 'v clause ⇒ 'st ⇒ 'st and
    propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
    forget-cond :: 'v clause ⇒ 'st ⇒ bool +
  fixes backjump-l-cond :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ bool
begin
  inductive backjump-l where
  backjump-l: trail S = F' @ Marked K () # F
    ⇒ no-dup (trail S)

```

$\Rightarrow T \sim \text{prepend-trail } (\text{Propagated } L \ ()) \ (\text{reduce-trail-to}_{NOT} \ F \ (\text{add-cl}_{NOT} \ (C' + \{\#L\# \}) \ S))$
 $\Rightarrow C \in \# \text{ clauses } S$
 $\Rightarrow \text{trail } S \models_{as} CNot \ C$
 $\Rightarrow \text{undefined-lit } F \ L$
 $\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' \ (\text{lits-of } (\text{trail } S))$
 $\Rightarrow \text{clauses } S \models_{pm} C' + \{\#L\#\}$
 $\Rightarrow F \models_{as} CNot \ C'$
 $\Rightarrow \text{backjump-l-cond } C \ C' \ L \ T$
 $\Rightarrow \text{backjump-l } S \ T$

inductive-cases *backjump-lE*: *backjump-l* *S* *T*

inductive *cdcl*_{NOT}-merged-bj-learn :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

*cdcl*_{NOT}-merged-bj-learn-decide_{NOT}: *decide*_{NOT} *S* *S'* \Rightarrow *cdcl*_{NOT}-merged-bj-learn *S* *S'* |
*cdcl*_{NOT}-merged-bj-learn-propagate_{NOT}: *propagate*_{NOT} *S* *S'* \Rightarrow *cdcl*_{NOT}-merged-bj-learn *S* *S'* |
*cdcl*_{NOT}-merged-bj-learn-backjump-l: *backjump-l* *S* *S'* \Rightarrow *cdcl*_{NOT}-merged-bj-learn *S* *S'* |
*cdcl*_{NOT}-merged-bj-learn-forget_{NOT}: *forget*_{NOT} *S* *S'* \Rightarrow *cdcl*_{NOT}-merged-bj-learn *S* *S'*

lemma *cdcl*_{NOT}-merged-bj-learn-no-dup-inv:

*cdcl*_{NOT}-merged-bj-learn *S* *T* \Rightarrow *no-dup* (*trail* *S*) \Rightarrow *no-dup* (*trail* *T*)

apply (*induction rule*: *cdcl*_{NOT}-merged-bj-learn.induct)

using *defined-lit-map* **apply** *fastforce*

using *defined-lit-map* **apply** *fastforce*

apply (*force simp*: *defined-lit-map elim*!: *backjump-lE*)[]

using *forget*_{NOT}.simps **apply** *auto*[1]

done

end

locale *cdcl*_{NOT}-merge-bj-learn-proxy =

*cdcl*_{NOT}-merge-bj-learn-ops *trail* *clauses* *prepend-trail* *tl-trail* *add-cl*_{NOT} *remove-cl*_{NOT}

propagate-conds *forget-conds* $\lambda C \ C' \ L' \ S. \ \text{backjump-l-cond } C \ C' \ L' \ S$

$\wedge \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$

for

trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**

clauses :: 'st \Rightarrow 'v clauses **and**

prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**

tl-trail :: 'st \Rightarrow 'st **and**

*add-cl*_{NOT} *remove-cl*_{NOT}:: 'v clause \Rightarrow 'st \Rightarrow 'st **and**

propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool **and**

forget-conds :: 'v clause \Rightarrow 'st \Rightarrow bool **and**

backjump-l-cond :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow bool +

fixes

inv :: 'st \Rightarrow bool

assumes

bj-can-jump:

$\bigwedge S \ C \ F' \ K \ F \ L.$

inv *S*

$\Rightarrow \text{trail } S = F' @ \text{Marked } K \ () \ \# \ F$

$\Rightarrow C \in \# \text{ clauses } S$

$\Rightarrow \text{trail } S \models_{as} CNot \ C$

$\Rightarrow \text{undefined-lit } F \ L$

$\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' \ (\text{lits-of } (F' @ \text{Marked } K \ () \ \# \ F))$

$\Rightarrow \text{clauses } S \models_{pm} C' + \{\#L\#\}$

$\Rightarrow F \models_{as} CNot \ C'$

$\Rightarrow \neg \text{no-step } \text{backjump-l } S$ **and**

cdcl-merged-inv: $\bigwedge S \ T. \ \text{cdcl}_{NOT}\text{-merged-bj-learn } S \ T \Rightarrow \text{inv } S \Rightarrow \text{inv } T$

```

begin
abbreviation backjump-conds where
backjump-conds  $\equiv \lambda\text{- } C\ L\ \text{- } \cdot\ \text{distinct-mset } (C + \{\#L\#\}) \wedge \neg\text{tautology } (C + \{\#L\#\})$ 

sublocale dpll-with-backjumping-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
propagate-conds inv backjump-conds
proof (unfold-locales, goal-cases)
  case 1
  { fix S S'
    assume bj: backjump-l S S' and no-dup (trail S)
    then obtain F' K F L C' C where
      S': S' ~ prepend-trail (Propagated L ()) (reduce-trail-toNOT F
        (tl-trail(add-clNOT (C' + {\#L\#}) S)))
      and
      tr-S: trail S = F' @ Marked K () # F and
      C: C ∈ # clauses S and
      tr-S-C: trail S ⊨as CNot C and
      undef-L: undefined-lit F L and
      atm-L: atm-of L ∈ atms-of-msu (clauses S) ∪ atm-of ' lits-of (trail S) and
      cls-S-C': clauses S ⊨pm C' + {\#L\#} and
      F-C': F ⊨as CNot C' and
      dist: distinct-mset (C' + {\#L\#}) and
      not-tauto:  $\neg\text{tautology } (C' + \{\#L\#\})$ 
      by (elim backjump-lE) simp

    have  $\exists S'. \text{backjumping-ops.backjump trail clauses prepend-trail tl-trail backjump-conds } S\ S'$ 
    apply rule
    apply (rule backjumping-ops.backjump.intros)
      apply unfold-locales
      using tr-S apply simp
      apply (rule state-eqNOT-ref)
      using C apply simp
      using tr-S-C apply simp
      using undef-L apply simp
      using atm-L apply simp
      using cls-S-C' apply simp
      using F-C' apply simp
      using dist not-tauto apply simp
    done
  } note H = this(1)
  then show ?case using 1 bj-can-jump by meson
qed

end

locale cdclNOT-merge-bj-learn-proxy2 =
cdclNOT-merge-bj-learn-proxy trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
propagate-conds forget-conds backjump-l-cond inv
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT:: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and

```

```

  inv :: 'st  $\Rightarrow$  bool and
  forget-conds :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool and
  backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

sublocale conflict-driven-clause-learning-ops trail clauses prepend-trail tl-trail add-clNOT
  remove-clNOT propagate-conds inv backjump-conds  $\lambda C$  -. distinct-mset  $C \wedge \neg$ tautology  $C$ 
  forget-conds
by unfold-locales
end

locale cdclNOT-merge-bj-learn =
  cdclNOT-merge-bj-learn-proxy2 trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv forget-conds backjump-l-cond
for
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  clauses :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool and
  forget-conds :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool and
  backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  bool +
assumes
  dpll-bj-inv:  $\bigwedge S T. \text{dpll-bj } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$  and
  learn-inv:  $\bigwedge S T. \text{learn } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$ 
begin

interpretation cdclNOT:
  conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv backjump-conds  $\lambda C$  -. distinct-mset  $C \wedge \neg$ tautology  $C$  forget-conds
apply unfold-locales
apply (simp only: cdclNOT.simps)
using cdclNOT-merged-bj-learn-forgetNOT cdcl-merged-inv learn-inv
by (auto simp add: cdclNOT.simps dpll-bj-inv)

lemma backjump-l-learn-backjump:
assumes bt: backjump-l  $S T$  and inv: inv  $S$  and n-d: no-dup (trail  $S$ )
shows  $\exists C' L. \text{learn } S (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) S)$ 
 $\wedge \text{backjump } (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) S) T$ 
 $\wedge \text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$ 
proof –
obtain  $C F' K F L l C'$  where
  tr-S: trail  $S = F' @ \text{Marked } K () \# F$  and
  T:  $T \sim \text{prepend-trail } (\text{Propagated } L l) (\text{reduce-trail-to}_{\text{NOT}} F (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) S))$  and
  C-clS:  $C \in \# \text{clauses } S$  and
  tr-S-CNot-C: trail  $S \models_{\text{as}} \text{CNot } C$  and
  undef: undefined-lit  $F L$  and
  atm-L: atm-of  $L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$  and
  clss-C: clauses  $S \models_{\text{pm}} C' + \{\#L\# \}$  and
  F  $\models_{\text{as}} \text{CNot } C'$  and
  distinct: distinct-mset  $(C' + \{\#L\# \})$  and
  not-tauto:  $\neg$ tautology  $(C' + \{\#L\# \})$ 
using bt inv by (elim backjump-lE) simp

```

have $\text{atms-}C'$: $\text{atms-of } C' \subseteq \text{atm-of } (\text{lits-of } F)$
proof –
obtain $ll :: 'v \Rightarrow ('v \text{ literal} \Rightarrow 'v) \Rightarrow 'v \text{ literal set} \Rightarrow 'v \text{ literal}$ **where**
 $\forall v f L. v \notin f 'L \vee v = f (ll \ v \ f \ L) \wedge ll \ v \ f \ L \in L$
by *moura*
then show *?thesis unfolding tr-S*
by (*metis (no-types) (F \models_{as} CNot C') atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*
atms-of-def in-CNot-implies-uminus(2) mem-set-mset-iff subsetI)
qed
then have $\text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } (\text{lits-of } (\text{trail } S))$
using *atm-L tr-S by auto*
moreover have *learn: learn S (add-cl_{NOT} (C' + {#L#}) S)*
apply (*rule learn.intros*)
apply (*rule clss-C*)
using *atms-C' atm-L apply (fastforce simp add: tr-S in-plus-implies-atm-of-on-atms-of-ms)*
apply *standard*
apply (*rule distinct*)
apply (*rule not-tauto*)
apply *simp*
done
moreover have *bj: backjump (add-cl_{NOT} (C' + {#L#}) S) T*
apply (*rule backjump.intros*)
using *(F \models_{as} CNot C') C-cl_S tr-S-CNot-C undef T distinct not-tauto n-d*
by (*auto simp: tr-S state-eq_{NOT}-def simp del: state-simp_{NOT}*)
ultimately show *?thesis by auto*
qed

lemma *cdcl_{NOT}-merged-bj-learn-is-tranclp-cdcl_{NOT}:*
 $\text{cdcl}_{NOT}\text{-merged-bj-learn } S \ T \Longrightarrow \text{inv } S \Longrightarrow \text{no-dup } (\text{trail } S) \Longrightarrow \text{cdcl}_{NOT}^{++} \ S \ T$
proof (*induction rule: cdcl_{NOT}-merged-bj-learn.induct*)
case (*cdcl_{NOT}-merged-bj-learn-decide_{NOT} T*)
then have $\text{cdcl}_{NOT} \ S \ T$
using *bj-decide_{NOT} cdcl_{NOT}.simps by fastforce*
then show *?case by auto*
next
case (*cdcl_{NOT}-merged-bj-learn-propagate_{NOT} T*)
then have $\text{cdcl}_{NOT} \ S \ T$
using *bj-propagate_{NOT} cdcl_{NOT}.simps by fastforce*
then show *?case by auto*
next
case (*cdcl_{NOT}-merged-bj-learn-forget_{NOT} T*)
then have $\text{cdcl}_{NOT} \ S \ T$
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 $C' :: 'v \text{ literal multiset}$ **and** $L :: 'v \text{ literal}$ **where**
 $f3: \text{learn } S \ (\text{add-cl}_{NOT} \ (C' + \{\#L\# \}) \ S) \wedge$
 $\text{backjump } (\text{add-cl}_{NOT} \ (C' + \{\#L\# \}) \ S) \ T \wedge$
 $\text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } (\text{lits-of } (\text{trail } S))$
using *n-d backjump-l-learn-backjump[OF bt inv] by blast*
then have $f4: \text{cdcl}_{NOT} \ S \ (\text{add-cl}_{NOT} \ (C' + \{\#L\# \}) \ S)$
using *n-d c-learn by blast*
have $\text{cdcl}_{NOT} \ (\text{add-cl}_{NOT} \ (C' + \{\#L\# \}) \ S) \ T$

using f_3 n -d bj -backjump c -dpll- bj by blast
 then show ?case
 using f_4 by (meson $trancpl.r$ -into- $tranc$ $trancpl.tranc$ -into- $tranc$)
 qed

lemma $rtrancpl$ - $cdcl_{NOT}$ -merged- bj -learn-is- $rtrancpl$ - $cdcl_{NOT}$ -and- inv :

$cdcl_{NOT}$ -merged- bj -learn** S $T \implies inv$ $S \implies no$ -dup ($trail$ S) $\implies cdcl_{NOT}$ ** S $T \wedge inv$ T

proof (induction rule: $rtrancpl$ -induct)

case base

then show ?case by auto

next

case (step T U) **note** $st = this(1)$ **and** $cdcl_{NOT} = this(2)$ **and** $IH = this(3)[OF\ this(4-)]$ **and**
 $inv = this(4)$ **and** n -d = $this(5)$

have $cdcl_{NOT}$ ** T U

using $cdcl_{NOT}$ -merged- bj -learn-is- $trancpl$ - $cdcl_{NOT}$ [$OF\ cdcl_{NOT}$] IH

$cdcl_{NOT}$. $rtrancpl$ - $cdcl_{NOT}$ -no-dup inv n -d by auto

then have $cdcl_{NOT}$ ** S U **using** IH **by** fastforce

moreover have inv U **using** n -d IH $\langle cdcl_{NOT}$ ** T $U \rangle$ $cdcl_{NOT}$. $rtrancpl$ - $cdcl_{NOT}$ - inv **by** blast

ultimately show ?case **using** st **by** fast

qed

lemma $rtrancpl$ - $cdcl_{NOT}$ -merged- bj -learn-is- $rtrancpl$ - $cdcl_{NOT}$:

$cdcl_{NOT}$ -merged- bj -learn** S $T \implies inv$ $S \implies no$ -dup ($trail$ S) $\implies cdcl_{NOT}$ ** S T

using $rtrancpl$ - $cdcl_{NOT}$ -merged- bj -learn-is- $rtrancpl$ - $cdcl_{NOT}$ -and- inv **by** blast

lemma $rtrancpl$ - $cdcl_{NOT}$ -merged- bj -learn- inv :

$cdcl_{NOT}$ -merged- bj -learn** S $T \implies inv$ $S \implies no$ -dup ($trail$ S) $\implies inv$ T

using $rtrancpl$ - $cdcl_{NOT}$ -merged- bj -learn-is- $rtrancpl$ - $cdcl_{NOT}$ -and- inv **by** blast

definition $\mu_C' :: 'v$ literal multiset set $\Rightarrow 'st \Rightarrow nat$ **where**

$\mu_C' A$ $T \equiv \mu_C (1 + card (atms$ -of- ms $A)) (2 + card (atms$ -of- ms $A)) (trail$ -weight $T)$

definition μ_{CDCL}' -merged :: $'v$ literal multiset set $\Rightarrow 'st \Rightarrow nat$ **where**

μ_{CDCL}' -merged A $T \equiv$

$((2 + card (atms$ -of- ms $A)) \wedge (1 + card (atms$ -of- ms $A)) - \mu_C' A$ $T) * 2 + card (set$ -mset ($clauses$ $T))$

lemma $cdcl_{NOT}$ -decreasing-measure':

assumes

$cdcl_{NOT}$ -merged- bj -learn S T **and**

inv : inv S **and**

atm -cls: $atms$ -of- msu ($clauses$ S) $\subseteq atms$ -of- ms A **and**

atm -trail: atm -of ' lits-of ($trail$ S) $\subseteq atms$ -of- ms A **and**

n -d: no-dup ($trail$ S) **and**

fin - A : finite A

shows μ_{CDCL}' -merged A $T < \mu_{CDCL}'$ -merged A S

using $assms(1)$

proof induction

case ($cdcl_{NOT}$ -merged- bj -learn-decide $_{NOT}$ T)

have $clauses$ $S = clauses$ T

using $cdcl_{NOT}$ -merged- bj -learn-decide $_{NOT}$. $hypos$ **by** auto

moreover have

$(2 + card (atms$ -of- ms $A)) \wedge (1 + card (atms$ -of- ms $A))$

$- \mu_C (1 + card (atms$ -of- ms $A)) (2 + card (atms$ -of- ms $A)) (trail$ -weight $T)$

$< (2 + card (atms$ -of- ms $A)) \wedge (1 + card (atms$ -of- ms $A))$

$- \mu_C (1 + card (atms$ -of- ms $A)) (2 + card (atms$ -of- ms $A)) (trail$ -weight $S)$

apply (*rule dpll-bj-trail-mes-decreasing-prop*)
using *cdcl_{NOT}-merged-bj-learn-decide_{NOT} fin-A atm-clss atm-trail n-d inv*
by (*simp-all add: bj-decide_{NOT} cdcl_{NOT}-merged-bj-learn-decide_{NOT}.hyps*)
ultimately show ?*case*
unfolding μ_{CDCL}' -merged-def μ_C' -def **by** *simp*
next
case (*cdcl_{NOT}-merged-bj-learn-propagate_{NOT} T*)
have *clauses S = clauses T*
using *cdcl_{NOT}-merged-bj-learn-propagate_{NOT}.hyps*
by (*simp add: bj-propagate_{NOT} inv dpll-bj-clauses*)
moreover have
 $(2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $< (2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S)$
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 μ_{CDCL}' -merged-def μ_C' -def **by** *simp*
next
case (*cdcl_{NOT}-merged-bj-learn-forget_{NOT} T*)
have *card (set-mset (clauses T)) < card (set-mset (clauses S))*
using $\langle \text{forget}_{NOT} S T \rangle$ **by** (*metis card-Diff1-less*
cdcl_{NOT}-merged-bj-learn-forget_{NOT}.hyps clauses-remove-cls_{NOT} finite-set-mset forgetE
mem-set-mset-iff order-refl set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses)
moreover
have *trail S = trail T*
using $\langle \text{forget}_{NOT} S T \rangle$ **by** (*auto elim: forgetE*)
then have
 $(2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $= (2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S)$
by *auto*
ultimately show ?*case*
unfolding μ_{CDCL}' -merged-def μ_C' -def **by** *simp*
next
case (*cdcl_{NOT}-merged-bj-learn-backjump-l T*) **note** *bj-l = this(1)*
obtain *C' L* **where**
learn: learn S (add-cls_{NOT} (C' + {#L#}) S) and
bj: backjump (add-cls_{NOT} (C' + {#L#}) S) T and
atms-C: atms-of (C' + {#L#}) \subseteq atms-of-msu (clauses S) \cup atm-of ' (lits-of (trail S))
using *bj-l inv backjump-l-learn-backjump n-d atm-clss atm-trail* **by** *blast*
have *card-T-S: card (set-mset (clauses T)) \leq 1 + card (set-mset (clauses S))*
using *bj-l inv* **by** (*force elim!: backjump-lE simp: card-insert-if*)
have
 $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T))$
 $< ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A))$
 $(\text{trail-weight } (\text{add-cls}_{NOT} (C' + \{ \#L\# \}) S)))$
apply (*rule dpll-bj-trail-mes-decreasing-prop*)
using *bj bj-backjump* **apply** *blast*
using *cdcl_{NOT}.c-learn cdcl_{NOT}.cdcl_{NOT}-inv inv learn* **apply** *blast*

```

    using atms-C atm-clss atm-trail n-d clauses-add-clssNOT apply simp apply fast
    using atm-trail n-d apply simp
    apply (simp add: n-d)
    using fin-A apply simp
    done
then have  $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T))$ 
 $< ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S))$ 
    using n-d by auto
then show ?case
    using card-T-S unfolding  $\mu_{CDCL}'$ -merged-def  $\mu_C'$ -def by linarith
qed

```

lemma *wf-cdcl_{NOT}-merged-bj-learn:*

assumes

fin-A: finite A

shows *wf {(T, S).*

$(\text{inv } S \wedge \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S))$

$\wedge \text{cdcl}_{\text{NOT-merged-bj-learn } S T}$

apply (*rule wfP-if-measure[of - - μ_{CDCL}' -merged A]*)

using *cdcl_{NOT}-decreasing-measure' fin-A by simp*

lemma *tranclp-cdcl_{NOT}-cdcl_{NOT}-tranclp:*

assumes

cdcl_{NOT}-merged-bj-learn⁺⁺ S T and

inv: inv S and

atm-clss: atms-of-msu (clauses S) \subseteq atms-of-ms A and

atm-trail: atm-of ' lits-of (trail S) \subseteq atms-of-ms A and

n-d: no-dup (trail S) and

fin-A[simp]: finite A

shows $(T, S) \in \{(T, S).$

$(\text{inv } S \wedge \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S))$

$\wedge \text{cdcl}_{\text{NOT-merged-bj-learn } S T}^+ (\text{is } - \in ?P^+)$

using *assms(1)*

proof (*induction rule: tranclp-induct*)

case *base*

then show *?case using n-d atm-clss atm-trail inv by auto*

next

case (*step T U*) **note** *st = this(1) and cdcl_{NOT} = this(2) and IH = this(3)*

have *cdcl_{NOT}** S T*

apply (*rule rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}*)

using *st cdcl_{NOT} inv n-d atm-clss atm-trail inv by auto*

have *inv T*

apply (*rule rtranclp-cdcl_{NOT}-merged-bj-learn-inv*)

using *inv st cdcl_{NOT} n-d atm-clss atm-trail inv by auto*

moreover have *atms-of-msu (clauses T) \subseteq atms-of-ms A*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-trail-clauses-bound[OF $\langle \text{cdcl}_{\text{NOT}}^{**} S T \rangle$ inv n-d atm-clss atm-trail]*
by fast

moreover have *atm-of ' (lits-of (trail T)) \subseteq atms-of-ms A*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-trail-clauses-bound[OF $\langle \text{cdcl}_{\text{NOT}}^{**} S T \rangle$ inv n-d atm-clss atm-trail]*
by fast

moreover have *no-dup (trail T)*

```

    using cdclNOT.rtrancpl-cdclNOT-no-dup[OF ⟨cdclNOT** S T⟩ inv n-d] by fast
ultimately have (U, T) ∈ ?P
    using cdclNOT by auto
then show ?case using IH by (simp add: trancpl-into-trancpl2)
qed

lemma wf-trancpl-cdclNOT-merged-bj-learn:
  assumes finite A
  shows wf {(T, S).
    (inv S ∧ atms-of-msu (clauses S) ⊆ atms-of-ms A ∧ atm-of ‘ lits-of (trail S) ⊆ atms-of-ms A
    ∧ no-dup (trail S))
    ∧ cdclNOT-merged-bj-learn++ S T}
  apply (rule wf-subset)
  apply (rule wf-trancpl[OF wf-cdclNOT-merged-bj-learn])
  using assms apply simp
using trancpl-cdclNOT-cdclNOT-trancpl[OF - - - - ⟨finite A⟩] by auto

lemma backjump-no-step-backjump-l:
  backjump S T ⇒ inv S ⇒ ¬no-step backjump-l S
  apply (elim backjumpE)
  apply (rule bj-can-jump)
  apply auto[7]
  by blast

lemma cdclNOT-merged-bj-learn-final-state:
  fixes A :: ‘v literal multiset set and S T :: ‘st
  assumes
    n-s: no-step cdclNOT-merged-bj-learn S and
    atms-S: atms-of-msu (clauses S) ⊆ atms-of-ms A and
    atms-trail: atm-of ‘ lits-of (trail S) ⊆ atms-of-ms A and
    n-d: no-dup (trail S) and
    finite A and
    inv: inv S and
    decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
  shows unsatisfiable (set-mset (clauses S))
    ∨ (trail S ⊨asm clauses S ∧ satisfiable (set-mset (clauses S)))
proof -
  let ?N = set-mset (clauses S)
  let ?M = trail S
  consider
    (sat) satisfiable ?N and ?M ⊨as ?N
  | (sat') satisfiable ?N and ¬ ?M ⊨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 ∈ ?N and ¬ ?M ⊨a C using M unfolding true-annots-def by auto
  obtain I :: ‘v literal set where
    I ⊨s ?N and
    cons: consistent-interp I and
    tot: total-over-m I ?N and
    atm-I-N: atm-of ‘ I ⊆ atms-of-ms ?N
  using sat unfolding satisfiable-def-min by auto
  let ?I = I ∪ {P | P. P ∈ lits-of ?M ∧ atm-of P ∉ atm-of ‘ I}

```

```

let ?O = { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M ∧ atm-of (lit-of L) ∉ atms-of-ms ?N }
have cons-I': consistent-interp ?I
  using cons using ⟨no-dup ?M⟩ unfolding consistent-interp-def
  by (auto simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def
    dest!: no-dup-cannot-not-lit-and-uminus)
have tot-I': total-over-m ?I (?N ∪ (λa. {#lit-of a#})) ' set ?M
  using tot atms-of-s-def unfolding total-over-m-def total-over-set-def
  by fastforce
have {P | P. P ∈ lits-of ?M ∧ atm-of P ∉ atm-of ' I} ⊨s ?O
  using ⟨I ⊨s ?N⟩ atm-I-N by (auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def)
then have I'-N: ?I ⊨s ?N ∪ ?O
  using ⟨I ⊨s ?N⟩ true-clss-union-increase by force
have tot': total-over-m ?I (?N ∪ ?O)
  using atm-I-N tot unfolding total-over-m-def total-over-set-def
  by (force simp: image-iff lits-of-def dest!: is-marked-ex-Marked)

have atms-N-M: atms-of-ms ?N ⊆ atm-of ' lits-of ?M
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain l :: 'v where
    l-N: l ∈ atms-of-ms ?N and
    l-M: l ∉ atm-of ' lits-of ?M
  by auto
  have undefined-lit ?M (Pos l)
    using l-M by (metis Marked-Propagated-in-iff-in-lits-of
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
  have decideNOT S (prepend-trail (Marked (Pos l) ()) S)
    by (metis ⟨undefined-lit ?M (Pos l)⟩ decideNOT.intros l-N literal.sel(1)
      state-eqNOT-ref)
  then show False
    using cdclNOT-merged-bj-learn-decideNOT n-s by blast
qed

have ?M ⊨as CNot C
  by (metis atms-N-M ⟨C ∈ ?N⟩ ⟨¬ ?M ⊨a C⟩ all-variables-defined-not-imply-cnot
    atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of atms-of-ms-CNot-atms-of-ms subsetCE)
have ∃ l ∈ set ?M. is-marked l
proof (rule ccontr)
  let ?O = { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M ∧ atm-of (lit-of L) ∉ atms-of-ms ?N }
  have ∅[iff]: ∧ I. total-over-m I (?N ∪ ?O ∪ (λa. {#lit-of a#})) ' set ?M
    ↔ total-over-m I (?N ∪ (λa. {#lit-of a#})) ' set ?M
    unfolding total-over-set-def total-over-m-def atms-of-ms-def by auto
  assume ¬ ?thesis
  then have [simp]: { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M }
    = { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M ∧ atm-of (lit-of L) ∉ atms-of-ms ?N }
    by auto
  then have ?N ∪ ?O ⊨ps (λa. {#lit-of a#}) ' set ?M
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] by auto

  then have ?I ⊨s (λa. {#lit-of a#}) ' set ?M
    using cons-I' I'-N tot-I' ⟨I ⊨s ?N ∪ ?O⟩ unfolding ∅ true-clss-clss-def by blast
  then have lits-of ?M ⊆ ?I
    unfolding true-clss-def lits-of-def by auto
  then have ?M ⊨as ?N
    using I'-N ⟨C ∈ ?N⟩ ⟨¬ ?M ⊨a C⟩ cons-I' atms-N-M

```

```

    by (meson (trail S ⊨as CNot C) consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
      true-annot-def true-clss-mono-set-mset-l true-clss-def)
  then show False using M by fast
qed
from List.split-list-first-propE[OF this] obtain K :: 'v literal and d :: unit and
  F F' :: ('v, unit, unit) marked-lit list where
  M-K: ?M = F' @ Marked K () # F and
  nm: ∀ f ∈ set F'. ¬is-marked f
  unfolding is-marked-def by (metis (full-types) old.unit.exhaust)
let ?K = Marked K () :: ('v, unit, unit) marked-lit
have ?K ∈ set ?M
  unfolding M-K by auto
let ?C = image-mset lit-of {#L ∈ #mset ?M. is-marked L ∧ L ≠ ?K#} :: 'v literal multiset
let ?C' = set-mset (image-mset (λL::'v literal. {#L#}) (?C + {#lit-of ?K#}))
have ?N ∪ {#{#lit-of L#} | L. is-marked L ∧ L ∈ set ?M} ⊨ps (λa. {#lit-of a#}) ' set ?M
  using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have C': ?C' = {#{#lit-of L#} | L. is-marked L ∧ L ∈ set ?M}
  unfolding M-K apply standard
  apply force
  using IntI by auto
ultimately have N-C-M: ?N ∪ ?C' ⊨ps (λa. {#lit-of a#}) ' set ?M
  by auto
have N-M-False: ?N ∪ (λL. {#lit-of L#}) ' (set ?M) ⊨ps {#{#}}
  using M ( ?M ⊨as CNot C ) (C ∈ ?N) unfolding true-clss-clss-def true-annot-def Ball-def
  true-annot-def by (metis consistent-CNot-not sup.orderE sup-commute true-clss-def
    true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have undefined-lit F K using (no-dup ?M) unfolding M-K by (simp add: defined-lit-map)
moreover
  have ?N ∪ ?C' ⊨ps {#{#}}
  proof -
    have A: ?N ∪ ?C' ∪ (λa. {#lit-of a#}) ' set ?M =
      ?N ∪ (λa. {#lit-of a#}) ' set ?M
    unfolding M-K by auto
    show ?thesis
      using true-clss-clss-left-right[OF N-C-M, of {#{#}}] N-M-False unfolding A by auto
  qed
have ?N ⊨p image-mset uminus ?C + {#-K#}
  unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
  proof (intro allI impI)
    fix I
    assume
      tot: total-over-set I (atms-of-ms (?N ∪ {image-mset uminus ?C + {#-K#}})) and
      cons: consistent-interp I and
      I ⊨s ?N
    have (K ∈ I ∧ -K ∉ I) ∨ (-K ∈ I ∧ K ∉ I)
      using cons tot unfolding consistent-interp-def by (cases K) auto
    have tot': total-over-set I
      (atm-of ' lit-of ' (set ?M ∩ {L. is-marked L ∧ L ≠ Marked K ()}))
    using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
    { fix x :: ('v, unit, unit) marked-lit
      assume
        a3: lit-of x ∉ I and
        a1: x ∈ set ?M and
        a4: is-marked x and

```

```

    a5:  $x \neq \text{Marked } K$  ()
  then have  $\text{Pos } (\text{atm-of } (\text{lit-of } x)) \in I \vee \text{Neg } (\text{atm-of } (\text{lit-of } x)) \in I$ 
    using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast
  moreover have  $f6: \text{Neg } (\text{atm-of } (\text{lit-of } x)) = - \text{Pos } (\text{atm-of } (\text{lit-of } x))$ 
    by simp
  ultimately have  $- \text{lit-of } x \in I$ 
    using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      literal.sel(1))
} note  $H = \text{this}$ 

have  $\neg I \models_s ?C'$ 
  using  $\langle ?N \cup ?C' \models_{ps} \{\{\#\}\} \rangle \text{ tot cons } \langle I \models_s ?N \rangle$ 
  unfolding true-clss-clss-def total-over-m-def
  by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of)
then show  $I \models \text{image-mset } \text{uminus } ?C + \{\# - K\# \}$ 
  unfolding true-clss-def true-clss-def Bex-mset-def
  using  $\langle (K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I) \rangle$ 
  by (auto dest!:  $H$ )
qed
moreover have  $F \models_{as} \text{CNot } (\text{image-mset } \text{uminus } ?C)$ 
  using nm unfolding true-annots-def CNot-def  $M-K$  by (auto simp add: lits-of-def)
ultimately have False
  using bj-can-jump[of  $S F' K F C -K$ 
    image-mset uminus (image-mset lit-of  $\{\# L : \# \text{ mset } ?M. \text{is-marked } L \wedge L \neq \text{Marked } K ()\# \}$ )]
     $\langle C \in ?N \rangle n-s \langle ?M \models_{as} \text{CNot } C \rangle$  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  $A :: 'v \text{ literal multiset set}$  and  $S T :: 'st$ 
assumes
    full: full cdclNOT-merged-bj-learn  $S T$  and
    atms-S: atms-of-msu (clauses  $S$ )  $\subseteq$  atms-of-ms  $A$  and
    atms-trail: atm-of ' lits-of (trail  $S$ )  $\subseteq$  atms-of-ms  $A$  and
    n-d: no-dup (trail  $S$ ) and
    finite  $A$  and
    inv: inv  $S$  and
    decomp: all-decomposition-implies-m (clauses  $S$ ) (get-all-marked-decomposition (trail  $S$ ))
shows unsatisfiable (set-mset (clauses  $T$ ))
   $\vee$  (trail  $T \models_{asm}$  clauses  $T \wedge$  satisfiable (set-mset (clauses  $T$ )))

proof –
  have st: cdclNOT-merged-bj-learn**  $S T$  and n-s: no-step cdclNOT-merged-bj-learn  $T$ 
    using full unfolding full-def by blast+
  then have st: cdclNOT**  $S T$ 
    using inv rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT-and-inv n-d by auto
  have atms-of-msu (clauses  $T$ )  $\subseteq$  atms-of-ms  $A$  and atm-of ' lits-of (trail  $T$ )  $\subseteq$  atms-of-ms  $A$ 
    using cdclNOT.rtranclp-cdclNOT-trail-clauses-bound[OF st inv n-d atms-S atms-trail] by blast+
  moreover have no-dup (trail  $T$ )
    using cdclNOT.rtranclp-cdclNOT-no-dup inv n-d st by blast
  moreover have inv  $T$ 
    using cdclNOT.rtranclp-cdclNOT-inv inv st by blast
  moreover have all-decomposition-implies-m (clauses  $T$ ) (get-all-marked-decomposition (trail  $T$ ))
    using cdclNOT.rtranclp-cdclNOT-all-decomposition-implies inv st decomp n-d by blast

```

ultimately show ?thesis
 using cdcl_{NOT}-merged-bj-learn-final-state[of T A] ⟨finite A⟩ n-s by fast
 qed
 end

14.8.1 Instantiations

locale cdcl_{NOT}-with-backtrack-and-restarts =
 conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt trail clauses
 prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} propagate-conds inv backjump-conds
 learn-restrictions forget-restrictions
for
 trail :: 'st ⇒ ('v::linorder, unit, unit) marked-lits **and**
 clauses :: 'st ⇒ 'v::linorder clauses **and**
 prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st **and**
 tl-trail :: 'st ⇒ 'st **and**
 add-cl_{NOT} remove-cl_{NOT}:: 'v clause ⇒ 'st ⇒ 'st **and**
 propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool **and**
 inv :: 'st ⇒ bool **and**
 backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool **and**
 learn-restrictions forget-restrictions :: 'v::linorder clause ⇒ 'st ⇒ bool
 +
fixes f :: nat ⇒ 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-msu (clauses S) ⊆ atms-of-ms A **and**
 atms-trail-S-A: atm-of ' lits-of (trail S) ⊆ atms-of-ms A **and**
 finite A **and**
 cdcl_{NOT}: cdcl_{NOT} S T
shows
 atms-of-msu (clauses T) ⊆ atms-of-ms A **and**
 atm-of ' lits-of (trail T) ⊆ atms-of-ms A **and**
 finite A
proof –
 have cdcl_{NOT} S T
 using ⟨inv S⟩ cdcl_{NOT} by linarith
 then have atms-of-msu (clauses T) ⊆ atms-of-msu (clauses S) ∪ atm-of ' lits-of (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-msu (clauses T) ⊆ atms-of-ms A
 using atms-clss-S-A atms-trail-S-A by blast
next
 show atm-of ' lits-of (trail T) ⊆ atms-of-ms A
 by (meson ⟨inv S⟩ atms-clss-S-A atms-trail-S-A cdcl_{NOT} cdcl_{NOT}-atms-in-trail-in-set n-d)
next
 show finite A
 using ⟨finite A⟩ by simp

qed

sublocale $cdcl_{NOT}$ -increasing-restarts-ops $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}) S \text{ } cdcl_{NOT} f$
 $\lambda A S. \text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge$
 $\text{finite } A$
 $\mu_{CDCL}' \lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
apply *unfold-locales*
 apply (*simp add: unbounded*)
 using *f-ge-1* **apply** *force*
 using *bound-inv-inv* **apply** *meson*
 apply (*rule cdcl_{NOT}-decreasing-measure'; simp*)
 apply (*rule rtranclp-cdcl_{NOT}- μ_{CDCL}' -bound; simp*)
 apply (*rule rtranclp- μ_{CDCL}' -bound-decreasing; simp*)
 apply *auto*
 apply *auto*
 using *cdcl_{NOT}-inv cdcl_{NOT}-no-dup* **apply** *blast*
using *inv-restart* **apply** *auto*
done

abbreviation $cdcl_{NOT}$ -l **where**

$cdcl_{NOT}$ -l \equiv
conflict-driven-clause-learning-ops.cdcl_{NOT} trail clauses prepend-trail tl-trail add-cl_{NOT}
remove-cl_{NOT} propagate-conds ($\lambda - - S T. \text{backjump } S T$)
 $(\lambda C S. \text{distinct-mset } C \wedge \neg \text{tautology } C \wedge \text{learn-restrictions } C S$
 $\wedge (\exists F K F' C' L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge C = C' + \{\#L\#$
 $\wedge F \models_{as} C \text{Not } C' \wedge C' + \{\#L\# \notin \text{clauses } S))$
 $(\lambda C S. \neg (\exists F' F K L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} C \text{Not } (C - \{\#L\#}))$
 $\wedge \text{forget-restrictions } C S)$

lemma $cdcl_{NOT}$ -with-restart- μ_{CDCL}' -le- μ_{CDCL}' -bound:

assumes
 $cdcl_{NOT}: cdcl_{NOT}\text{-restart } (T, a) (V, b) \text{ and}$
 $cdcl_{NOT}\text{-inv:}$
 $\text{inv } T$
 $\text{no-dup } (\text{trail } T) \text{ and}$
 bound-inv:
 $\text{atms-of-msu } (\text{clauses } T) \subseteq \text{atms-of-ms } A$
 $\text{atm-of ' lits-of } (\text{trail } T) \subseteq \text{atms-of-ms } A$
 $\text{finite } A$
shows $\mu_{CDCL}' A V \leq \mu_{CDCL}'\text{-bound } A T$
using $cdcl_{NOT}\text{-inv bound-inv}$
proof (*induction rule: cdcl_{NOT}-with-restart-induct[OF cdcl_{NOT}]*)
case ($1 m S T n U$) **note** $U = \text{this}(3)$
show *?case*
 apply (*rule rtranclp-cdcl_{NOT}- μ_{CDCL}' -bound-reduce-trail-to_{NOT}[of S T]*)
 using $\langle (cdcl_{NOT} \sim m) S T \rangle$ **apply** (*fastforce dest!: relpowp-imp-rtranclp*)
 using 1 **by** *auto*
next
case ($2 S T n$) **note** $\text{full} = \text{this}(2)$
show *?case*
 apply (*rule rtranclp-cdcl_{NOT}- μ_{CDCL}' -bound*)
 using $\text{full } 2$ **unfolding** *full1-def* **by** *force+*
qed

lemma *cdcl_{NOT}-with-restart- μ_{CDCL}' -bound-le- μ_{CDCL}' -bound:*
assumes
cdcl_{NOT}: *cdcl_{NOT}-restart* (*T*, *a*) (*V*, *b*) **and**
cdcl_{NOT}-inv:
inv *T*
no-dup (*trail* *T*) **and**
bound-inv:
atms-of-msu (*clauses* *T*) \subseteq *atms-of-ms* *A*
atm-of ' *lits-of* (*trail* *T*) \subseteq *atms-of-ms* *A*
finite *A*
shows $\mu_{CDCL}'\text{-bound } A \ V \leq \mu_{CDCL}'\text{-bound } A \ T$
using *cdcl_{NOT}-inv* *bound-inv*
proof (*induction rule:* *cdcl_{NOT}-with-restart-induct*[*OF* *cdcl_{NOT}*])
case (*1 m S T n U*) **note** *U = this*(3)
have $\mu_{CDCL}'\text{-bound } A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$
apply (*rule* *rtrancp- μ_{CDCL}' -bound-decreasing*)
using $\langle (cdcl_{NOT} \rightsquigarrow m) \ S \ T \rangle$ **apply** (*fastforce* *dest: relpowp-imp-rtrancp*)
using *1* **by** *auto*
then show *?case* **using** *U* **unfolding** $\mu_{CDCL}'\text{-bound-def}$ **by** *auto*
next
case (*2 S T n*) **note** *full = this*(2)
show *?case*
apply (*rule* *rtrancp- μ_{CDCL}' -bound-decreasing*)
using *full 2* **unfolding** *full1-def* **by** *force+*
qed

sublocale *cdcl_{NOT}-increasing-restarts - - - - f*
 $\lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} (\llbracket :: 'a \text{ list} \rrbracket) \ S$
 $\lambda A \ S. \ \text{atms-of-msu} \ (\text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}' \text{ cdcl}_{NOT}$
 $\lambda S. \ \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
apply *unfold-locales*
using *cdcl_{NOT}-with-restart- μ_{CDCL}' -le- μ_{CDCL}' -bound* **apply** *simp*
using *cdcl_{NOT}-with-restart- μ_{CDCL}' -bound-le- μ_{CDCL}' -bound* **apply** *simp*
done

lemma *cdcl_{NOT}-restart-all-decomposition-implies:*
assumes *cdcl_{NOT}-restart* *S T* **and**
inv (*fst* *S*) **and**
no-dup (*trail* (*fst* *S*))
all-decomposition-implies-m (*clauses* (*fst* *S*)) (*get-all-marked-decomposition* (*trail* (*fst* *S*)))
shows
all-decomposition-implies-m (*clauses* (*fst* *T*)) (*get-all-marked-decomposition* (*trail* (*fst* *T*)))
using *assms* **apply** (*induction*)
using *rtrancp-cdcl_{NOT}-all-decomposition-implies* **by** (*auto* *dest!:* *trancp-into-rtrancp*
simp: full1-def)

lemma *rtrancp-cdcl_{NOT}-restart-all-decomposition-implies:*
assumes *cdcl_{NOT}-restart*** *S T* **and**
inv: *inv* (*fst* *S*) **and**
n-d: *no-dup* (*trail* (*fst* *S*)) **and**
decomp:
all-decomposition-implies-m (*clauses* (*fst* *S*)) (*get-all-marked-decomposition* (*trail* (*fst* *S*)))

```

shows
  all-decomposition-implies-m (clauses (fst T)) (get-all-marked-decomposition (trail (fst T)))
using assms(1)
proof (induction rule: 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-cdclNOT-with-restart-cdclNOT-inv[OF st] inv n-d by blast
  moreover have no-dup (trail (fst T))
    using rtrancpl-cdclNOT-with-restart-cdclNOT-inv[OF st] inv n-d by blast
  ultimately show ?case
    using cdclNOT-restart-all-decomposition-implies r IH n-d by fast
qed

lemma cdclNOT-restart-sat-ext-iff:
assumes
  st: cdclNOT-restart S T and
  n-d: no-dup (trail (fst S)) and
  inv: inv (fst S)
shows  $I \models_{\text{sextm}} \text{clauses}(\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}(\text{fst } T)$ 
using assms
proof (induction)
  case (restart-step m S T n U)
  then show ?case
    using rtrancpl-cdclNOT-bj-sat-ext-iff n-d by (fastforce dest!: relpowp-imp-rtrancpl)
next
  case restart-full
  then show ?case using rtrancpl-cdclNOT-bj-sat-ext-iff unfolding full1-def
    by (fastforce dest!: trancpl-into-rtrancpl)
qed

lemma rtrancpl-cdclNOT-restart-sat-ext-iff:
assumes
  st: cdclNOT-restart** S T and
  n-d: no-dup (trail (fst S)) and
  inv: inv (fst S)
shows  $I \models_{\text{sextm}} \text{clauses}(\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}(\text{fst } T)$ 
using st
proof (induction)
  case base
  then show ?case by simp
next
  case (step T U) note st = this(1) and r = this(2) and IH = this(3)
  have inv (fst T)
    using rtrancpl-cdclNOT-with-restart-cdclNOT-inv[OF st] inv n-d by blast+
  moreover have no-dup (trail (fst T))
    using rtrancpl-cdclNOT-with-restart-cdclNOT-inv rtrancpl-cdclNOT-no-dup st inv n-d by blast
  ultimately show ?case
    using cdclNOT-restart-sat-ext-iff[OF r] IH by blast
qed

theorem full-cdclNOT-restart-backjump-final-state:
fixes A :: 'v literal multiset set' and S T :: 'st'

```

assumes
full: *full cdcl_{NOT}-restart* (*S*, *n*) (*T*, *m*) **and**
atms-S: *atms-of-msu* (*clauses S*) \subseteq *atms-of-ms A* **and**
atms-trail: *atm-of* ‘*lits-of* (*trail S*) \subseteq *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A[simp]: *finite A* **and**
inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses S*) (*get-all-marked-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses S*))
 \vee (*lits-of* (*trail T*) \models_{sextm} *clauses S* \wedge *satisfiable* (*set-mset* (*clauses S*)))
proof –
have *st*: *cdcl_{NOT}-restart*** (*S*, *n*) (*T*, *m*) **and**
n-s: *no-step cdcl_{NOT}-restart* (*T*, *m*)
using *full unfolding full-def* **by** *fast+*
have *binv-T*: *atms-of-msu* (*clauses T*) \subseteq *atms-of-ms A* *atm-of* ‘*lits-of* (*trail T*) \subseteq *atms-of-ms A*
using *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 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 T*))
 \vee (*trail T* \models_{asm} *clauses T* \wedge *satisfiable* (*set-mset* (*clauses 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*: $\bigwedge I. I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$
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* (*trail T*))
using *inv-T(1) distinctconsistent-interp* **by** *blast*
consider
(*unsat*) *unsatisfiable* (*set-mset* (*clauses T*))
| (*sat*) *trail T* \models_{asm} *clauses T* **and** *satisfiable* (*set-mset* (*clauses T*))
using *T* **by** *blast*
then show *?thesis*
proof *cases*
case *unsat*
then have *unsatisfiable* (*set-mset* (*clauses S*))
using *eq-sat-S-T consistent-true-clss-ext-satisfiable true-clss-imp-true-cls-ext*
unfolding *satisfiable-def* **by** *blast*
then show *?thesis* **by** *fast*
next
case *sat*
then have *lits-of* (*trail T*) \models_{sextm} *clauses S*
using *rtrancpl-cdcl_{NOT}-restart-sat-ext-iff*[*OF st*] *inv n-d atms-S*
atms-trail **by** (*auto simp: true-clss-imp-true-cls-ext true-annots-true-cls*)
moreover then have *satisfiable* (*set-mset* (*clauses S*))
using *cons-T consistent-true-clss-ext-satisfiable* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed
qed
end — end of *cdcl_{NOT}-with-backtrack-and-restarts* locale

locale *most-general-cdcl_{NOT}* =

```

dpll-state trail clauses prepend-trail tl-trail add-clNOT remove-clNOT +
propagate-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds +
backjumping-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT λ- - - -. True
for
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clNOT remove-clNOT:: 'v clause ⇒ 'st ⇒ 'st and
  propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
  inv :: 'st ⇒ bool
begin
lemma backjump-bj-can-jump:
  assumes
    tr-S: trail S = F' @ Marked K () # F and
    C: C ∈ # clauses S and
    tr-S-C: trail S ⊨as CNot C and
    undef: undefined-lit F L and
    atm-L: atm-of L ∈ atms-of-msu (clauses S) ∪ atm-of ' (lits-of (F' @ Marked K () # F)) and
    cls-S-C': clauses S ⊨pm C' + {#L#} and
    F-C': F ⊨as CNot C'
  shows ¬no-step backjump S
  using backjump.intros[OF tr-S - C tr-S-C undef - cls-S-C' F-C',
    of prepend-trail (Propagated L -) (reduce-trail-toNOT F S)] atm-L unfolding tr-S
  by (auto simp: state-eqNOT-def simp del: state-simpNOT)

sublocale dpll-with-backjumping-ops - - - - - inv λ- - - -. True
  using backjump-bj-can-jump by unfold-locales auto
end

```

The restart does only reset the trail, contrary to Weidenbach's version. But there is a forget rule.

```

locale cdclNOT-merge-bj-learn-with-backtrack-restarts =
  cdclNOT-merge-bj-learn trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv forget-conds
  λC C' L' S. distinct-mset (C' + {#L'#}) ∧ backjump-l-cond C C' L' S
for
  trail :: 'st ⇒ ('v::linorder, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v::linorder clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clNOT remove-clNOT:: 'v clause ⇒ 'st ⇒ 'st and
  propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
  inv :: 'st ⇒ bool and
  forget-conds :: 'v clause ⇒ 'st ⇒ bool and
  backjump-l-cond :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ bool
+
fixes f :: nat ⇒ nat
assumes
  unbounded: unbounded f and f-ge-1: ∧n. n ≥ 1 ⇒ f n ≥ 1 and
  inv-restart: ∧S T. inv S ⇒ T ∼ reduce-trail-toNOT [] S ⇒ inv T
begin

```

interpretation $cdcl_{NOT}$:

conflict-driven-clause-learning-ops *trail* *clauses* *prepend-trail* *tl-trail* *add-cls_{NOT}* *remove-cls_{NOT}*
propagate-conds *inv* *backjump-conds* ($\lambda C \cdot \text{distinct-mset } C \wedge \neg \text{tautology } C$) *forget-conds*
by *unfold-locales*

interpretation $cdcl_{NOT}$:

conflict-driven-clause-learning *trail* *clauses* *prepend-trail* *tl-trail* *add-cls_{NOT}* *remove-cls_{NOT}*
propagate-conds *inv* *backjump-conds* ($\lambda C \cdot \text{distinct-mset } C \wedge \neg \text{tautology } C$) *forget-conds*
apply *unfold-locales*
using $cdcl_{NOT}$ -merged-bj-learn-forget_{NOT} $cdcl$ -merged-inv *learn-inv*
by (*auto simp* *add*: $cdcl_{NOT}$.*simps* *dpll-bj-inv*)

definition *not-simplified-cl* $A = \{\#C \in \# A. \text{tautology } C \vee \neg \text{distinct-mset } C\}$

lemma *build-all-simple-clss-or-not-simplified-cl*:

assumes *atms-of-msu* (*clauses* S) \subseteq *atms-of-ms* A **and**

$x \in \#$ *clauses* S **and** *finite* A

shows $x \in \text{build-all-simple-clss } (\text{atms-of-ms } A) \vee x \in \# \text{not-simplified-cl } (\text{clauses } S)$

proof –

consider

(*simpl*) $\neg \text{tautology } x$ **and** *distinct-mset* x

| (*n-simp*) *tautology* $x \vee \neg \text{distinct-mset } x$

by *auto*

then show ?thesis

proof *cases*

case *simpl*

then have $x \in \text{build-all-simple-clss } (\text{atms-of-ms } A)$

by (*meson* *assms* *atms-of-atms-of-ms-mono* *atms-of-ms-finite* *build-all-simple-clss-mono*
distinct-mset-not-tautology-implies-in-build-all-simple-clss *finite-subset*
mem-set-mset-iff subsetCE)

then show ?thesis **by** *blast*

next

case *n-simp*

then have $x \in \# \text{not-simplified-cl } (\text{clauses } S)$

using ($x \in \#$ *clauses* S) **unfolding** *not-simplified-cl*-def **by** *auto*

then show ?thesis **by** *blast*

qed

qed

lemma $cdcl_{NOT}$ -merged-bj-learn-clauses-bound:

assumes

$cdcl_{NOT}$ -merged-bj-learn S T **and**

inv: *inv* S **and**

atms-clss: *atms-of-msu* (*clauses* S) \subseteq *atms-of-ms* A **and**

atms-trail: *atm-of* (*lits-of* (*trail* S)) \subseteq *atms-of-ms* A **and**

n-d: *no-dup* (*trail* S) **and**

fin-A[*simp*]: *finite* A

shows *set-mset* (*clauses* T) \subseteq *set-mset* (*not-simplified-cl* (*clauses* S))
 \cup *build-all-simple-clss* (*atms-of-ms* A)

using *assms*

proof (*induction rule*: $cdcl_{NOT}$ -merged-bj-learn.induct)

case $cdcl_{NOT}$ -merged-bj-learn-decide_{NOT}

then show ?case **using** *dpll-bj-clauses* **by** (*force* *dest*!: *build-all-simple-clss-or-not-simplified-cl*)

next

case $cdcl_{NOT}$ -merged-bj-learn-propagate_{NOT}

then show *?case using dpll-bj-clauses by (force dest!: build-all-simple-clss-or-not-simplified-cls)*
next
case *cdcl_{NOT}-merged-bj-learn-forget_{NOT}*
then show *?case using clauses-remove-cl_{NOT} unfolding state-eq_{NOT}-def*
by (force elim!: forgetE dest: build-all-simple-clss-or-not-simplified-cls)
next
case *(cdcl_{NOT}-merged-bj-learn-backjump-l T) note bj = this(1) and inv = this(2) and*
atms-clss = this(3) and atms-trail = this(4) and n-d = this(5)

have *cdcl_{NOT}** S T*
apply *(rule rtrancpl-cdcl_{NOT}-merged-bj-learn-is-rtrancpl-cdcl_{NOT})*
using *(backjump-l S T) inv cdcl_{NOT}-merged-bj-learn.simps n-d by blast+*
have *atm-of (lits-of (trail T)) ⊆ atms-of-ms A*
using *cdcl_{NOT}.rtrancpl-cdcl_{NOT}-trail-clauses-bound[OF (cdcl_{NOT}** S T) inv atms-trail atms-clss*
n-d by auto
have *atms-of-msu (clauses T) ⊆ atms-of-ms A*
using *cdcl_{NOT}.rtrancpl-cdcl_{NOT}-trail-clauses-bound[OF (cdcl_{NOT}** S T) inv n-d atms-clss atms-trail]*
by fast
moreover have *no-dup (trail T)*
using *cdcl_{NOT}.rtrancpl-cdcl_{NOT}-no-dup[OF (cdcl_{NOT}** S T) inv n-d] by fast*

obtain *F' K F L l C' C where*
tr-S: trail S = F' @ Marked K () # F and
T: T ~ prepend-trail (Propagated L l) (reduce-trail-to_{NOT} F (add-cl_{NOT} (C' + {#L#}) S)) and
C ∈# clauses S and
trail S ⊨_{as} CNot C and
undef: undefined-lit F L and
atm-of L = atm-of K ∨ atm-of L ∈ atms-of-msu (clauses S)
∨ atm-of L ∈ atm-of (lits-of F' ∪ lits-of F) and
clauses S ⊨_{pm} C' + {#L#} and
F ⊨_{as} CNot C' and
dist: distinct-mset (C' + {#L#}) and
tauto: ¬ tautology (C' + {#L#}) and
backjump-l-cond C C' L T
using *(backjump-l S T) apply (induction rule: backjump-l.induct) by auto*

have *atms-of C' ⊆ atm-of (lits-of F)*
using *(F ⊨_{as} CNot C') by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*
atms-of-def image-subset-iff in-CNot-implies-uminus(2))
then have *atms-of (C' + {#L#}) ⊆ atms-of-ms A*
using *T (atm-of (lits-of (trail T) ⊆ atms-of-ms A) tr-S undef n-d by auto*
then have *build-all-simple-clss (atms-of (C' + {#L#})) ⊆ build-all-simple-clss (atms-of-ms A)*
apply *– by (rule build-all-simple-clss-mono) (simp-all)*
then have *C' + {#L#} ∈ build-all-simple-clss (atms-of-ms A)*
using *distinct-mset-not-tautology-implies-in-build-all-simple-clss[OF dist tauto]*
by auto
then show *?case*
using *T inv atms-clss undef tr-S n-d*
by (force dest!: build-all-simple-clss-or-not-simplified-cls)
qed

lemma *cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing:*
assumes *cdcl_{NOT}-merged-bj-learn S T*
shows *(not-simplified-cls (clauses T)) ⊆# (not-simplified-cls (clauses S))*
using *assms apply induction*

prefer 4
unfolding *not-simplified-cls-def* **apply** (auto elim!: backjump-LE forgetE)[3]
by (elim backjump-LE) auto

lemma *rtrancp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl_{NOT}-merged-bj-learn** S T*
shows (*not-simplified-cls (clauses T)*) $\subseteq \#$ (*not-simplified-cls (clauses S)*)
using *assms* **apply** *induction*
apply *simp*
by (*drule cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*) auto

lemma *rtrancp-cdcl_{NOT}-merged-bj-learn-clauses-bound*:

assumes
*cdcl_{NOT}-merged-bj-learn** S T* **and**
inv S **and**
atms-of-msu (clauses S) \subseteq atms-of-ms A **and**
atm-of ' (lits-of (trail S)) \subseteq atms-of-ms A **and**
n-d: no-dup (trail S) **and**
finite[simp]: finite A
shows *set-mset (clauses T) \subseteq set-mset (not-simplified-cls (clauses S))*
 \cup *build-all-simple-clss (atms-of-ms A)*
using *assms(1-5)*
proof *induction*
case *base*
then show ?case **by** (auto dest!: *build-all-simple-clss-or-not-simplified-cls*)

next

case (*step T U*) **note** *st = this(1)* **and** *cdcl_{NOT} = this(2)* **and** *IH = this(3)[OF this(4-7)]* **and**
inv = this(4) **and** *atms-clss-S = this(5)* **and** *atms-trail-S = this(6)* **and** *finite-clss-S = this(7)*
have *st': cdcl_{NOT}** S T*
using *inv rtrancp-cdcl_{NOT}-merged-bj-learn-is-rtrancp-cdcl_{NOT}-and-inv st n-d* **by** *blast*
have *inv T*
using *inv rtrancp-cdcl_{NOT}-merged-bj-learn-inv st n-d* **by** *blast*
moreover
have *atms-of-msu (clauses T) \subseteq atms-of-ms A* **and**
atm-of ' lits-of (trail T) \subseteq atms-of-ms A
using *cdcl_{NOT}.rtrancp-cdcl_{NOT}-trail-clauses-bound[OF st'] inv atms-clss-S atms-trail-S n-d*
by *blast+*
moreover moreover have *no-dup (trail T)*
using *cdcl_{NOT}.rtrancp-cdcl_{NOT}-no-dup[OF <cdcl_{NOT}** S T> inv n-d]* **by** *fast*
ultimately have *set-mset (clauses U)*
 \subseteq *set-mset (not-simplified-cls (clauses T)) \cup build-all-simple-clss (atms-of-ms A)*
using *cdcl_{NOT} finite cdcl_{NOT}-merged-bj-learn-clauses-bound*
by (auto intro!: *cdcl_{NOT}-merged-bj-learn-clauses-bound*)
moreover have *set-mset (not-simplified-cls (clauses T))*
 \subseteq *set-mset (not-simplified-cls (clauses S))*
using *rtrancp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing[OF st]* **by** auto
ultimately show ?case **using** *IH inv atms-clss-S*
by (auto dest!: *build-all-simple-clss-or-not-simplified-cls*)

qed

abbreviation μ_{CDCL}' -bound **where**

μ_{CDCL}' -bound *A T* == ((2+card (atms-of-ms A)) \wedge (1+card (atms-of-ms A))) * 2
+ card (set-mset (not-simplified-cls (clauses T)))
+ 3 \wedge card (atms-of-ms A)

lemma *rtrancpl-cdcl_{NOT}-merged-bj-learn-clauses-bound-card*:

assumes

*cdcl_{NOT}-merged-bj-learn*** *S T* **and**

inv S **and**

atms-of-msu (*clauses S*) \subseteq *atms-of-ms A* **and**

atm-of ‘(*lits-of* (*trail S*)) \subseteq *atms-of-ms A* **and**

n-d: *no-dup* (*trail S*) **and**

finite: *finite A*

shows $\mu_{CDCL}'\text{-merged } A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$

proof –

have *set-mset* (*clauses T*) \subseteq *set-mset* (*not-simplified-cls*(*clauses S*))

\cup *build-all-simple-clss* (*atms-of-ms A*)

using *rtrancpl-cdcl_{NOT}-merged-bj-learn-clauses-bound*[*OF assms*] .

moreover have *card* (*set-mset* (*not-simplified-cls*(*clauses S*))

\cup *build-all-simple-clss* (*atms-of-ms A*))

\leq *card* (*set-mset* (*not-simplified-cls*(*clauses S*))) + 3 \wedge *card* (*atms-of-ms A*)

by (*meson Nat.le-trans atms-of-ms-finite build-all-simple-clss-card card-Un-le finite nat-add-left-cancel-le*)

ultimately have *card* (*set-mset* (*clauses T*))

\leq *card* (*set-mset* (*not-simplified-cls*(*clauses S*))) + 3 \wedge *card* (*atms-of-ms A*)

by (*meson build-all-simple-clss-finite card-mono dual-order.trans finite-UnI finite-set-mset*)

moreover have ((2 + *card* (*atms-of-ms A*)) \wedge (1 + *card* (*atms-of-ms A*)) – $\mu_C' A \ T$) * 2

\leq (2 + *card* (*atms-of-ms A*)) \wedge (1 + *card* (*atms-of-ms A*)) * 2

by *auto*

ultimately show *?thesis unfolding* $\mu_{CDCL}'\text{-merged-def}$ **by** *auto*

qed

sublocale *cdcl_{NOT}-increasing-restarts-ops* $\lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}] \ S$

cdcl_{NOT}-merged-bj-learn f

$\lambda A \ S. \ \text{atms-of-msu} \ (\text{clauses } S) \subseteq \text{atms-of-ms } A$

$\wedge \text{atm-of 'lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$

$\mu_{CDCL}'\text{-merged}$

$\lambda S. \ \text{inv } S \wedge \text{no-dup } (\text{trail } S)$

$\mu_{CDCL}'\text{-bound}$

apply *unfold-locales*

using *unbounded apply simp*

using *f-ge-1 apply force*

apply (*blast dest!*: *cdcl_{NOT}-merged-bj-learn-is-trancpl-cdcl_{NOT} trancpl-into-rtrancpl cdcl_{NOT}.rtrancpl-cdcl_{NOT}-trail-clauses-bound*)

apply (*simp add*: *cdcl_{NOT}-decreasing-measure'*)

using *rtrancpl-cdcl_{NOT}-merged-bj-learn-clauses-bound-card* **apply** *blast*

apply (*drule rtrancpl-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*)

apply (*auto dest!*: *simp: card-mono set-mset-mono*) []

apply *simp*

apply *auto* []

using *cdcl_{NOT}-merged-bj-learn-no-dup-inv cdcl-merged-inv* **apply** *blast*

apply (*auto simp: inv-restart*) []

done

lemma *cdcl_{NOT}-restart- $\mu_{CDCL}'\text{-merged-le-}\mu_{CDCL}'\text{-bound}$* :

assumes

cdcl_{NOT}-restart T V

inv (*fst T*) **and**

no-dup (*trail* (*fst T*)) **and**

atms-of-msu (*clauses* (*fst T*)) \subseteq *atms-of-ms A* **and**

$atm\text{-}of \text{ ' } lits\text{-}of (trail (fst T)) \subseteq atm\text{-}of\text{-}ms A$ and
 $finite A$
shows $\mu_{CDCL}'\text{-merged } A (fst V) \leq \mu_{CDCL}'\text{-bound } A (fst T)$
using *assms*
proof *induction*
case (*restart-full* $S T n$)
show *?case*
unfolding *fst-conv*
apply (*rule rtranclp-cdcl_{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**
 $n\text{-}d = this(5)$ **and** $atms\text{-}clss = this(6)$ **and** $atms\text{-}trail = this(7)$ **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 $inv\ n\text{-}d$ **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 $inv\ st'\ n\text{-}d$ **by** *auto*
then have $inv U$
using U **by** (*auto simp: inv-restart*)
have $atms\text{-}of\text{-}msu (clauses T) \subseteq atm\text{-}of\text{-}ms A$
using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-trail-clauses-bound*[*OF st'*] $inv\ atms\text{-}clss\ atms\text{-}trail\ n\text{-}d$
by *simp*
then have $atms\text{-}of\text{-}msu (clauses U) \subseteq atm\text{-}of\text{-}ms A$
using U **by** *simp*
have $not\text{-}simplified\text{-}cls (clauses U) \subseteq \# not\text{-}simplified\text{-}cls (clauses T)$
using $\langle U \sim reduce\text{-}trail\text{-}to_{NOT} \sqcup T \rangle$ **by** *auto*
moreover have $not\text{-}simplified\text{-}cls (clauses T) \subseteq \# not\text{-}simplified\text{-}cls (clauses S)$
apply (*rule rtranclp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*)
using $\langle (cdcl_{NOT}\text{-merged-bj-learn} \tilde{m}) S T \rangle$ **by** (*auto dest!: relpowp-imp-rtranclp*)
ultimately have $U\text{-}S: not\text{-}simplified\text{-}cls (clauses U) \subseteq \# not\text{-}simplified\text{-}cls (clauses S)$
by *auto*

have (*set-mset* ($clauses U$))
 $\subseteq set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}ms A)$
apply (*rule rtranclp-cdcl_{NOT}-merged-bj-learn-clauses-bound*)
apply *simp*
using $\langle inv U \rangle$ **apply** *simp*
using $\langle atms\text{-}of\text{-}msu (clauses U) \subseteq atm\text{-}of\text{-}ms A \rangle$ **apply** *simp*
using U **apply** *simp*
using U **apply** *simp*
using *finite* **apply** *simp*
done
then have $f1: card (set\text{-}mset (clauses U)) \leq card (set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U))$
 $\cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}ms A))$
by (*meson build-all-simple-clss-finite card-mono finite-UnI finite-set-mset*)

moreover have $set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}ms A)$
 $\subseteq set\text{-}mset (not\text{-}simplified\text{-}cls (clauses S)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}ms A)$
using $U\text{-}S$ **by** *auto*
then have $f2:$
 $card (set\text{-}mset (not\text{-}simplified\text{-}cls (clauses U)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}ms A))$
 $\leq card (set\text{-}mset (not\text{-}simplified\text{-}cls (clauses S)) \cup build\text{-}all\text{-}simple\text{-}clss (atms\text{-}of\text{-}ms A))$

by (meson build-all-simple-clss-finite card-mono finite-UnI finite-set-mset)

moreover have card (set-mset (not-simplified-cls (clauses S))
 \cup build-all-simple-clss (atms-of-ms A))
 \leq card (set-mset (not-simplified-cls (clauses S))) + card (build-all-simple-clss (atms-of-ms A))
 using card-Un-le by blast

moreover have card (build-all-simple-clss (atms-of-ms A)) $\leq 3 \wedge$ card (atms-of-ms A)
 using atms-of-ms-finite build-all-simple-clss-card local.finite by blast

ultimately have card (set-mset (clauses U))
 \leq card (set-mset (not-simplified-cls (clauses S))) + $3 \wedge$ card (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 not-simplified-cls (clauses T) $\subseteq \#$ not-simplified-cls (clauses S)
apply (rule rtrancpl-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing)
using (full1 cdcl_{NOT}-merged-bj-learn S T) **unfolding** full1-def
by (auto dest: trancpl-into-rtrancpl)
then show ?case by (auto simp: card-mono set-mset-mono)

next
case (restart-step m S T n U) **note** st = this(1) **and** U = this(3) **and** n-d = this(4) **and** inv = this(5)
then have st': cdcl_{NOT}-merged-bj-learn** S T
by (blast dest: relpowp-imp-rtrancpl)
then have st'': cdcl_{NOT}** S T
using inv n-d **apply** – **by** (rule rtrancpl-cdcl_{NOT}-merged-bj-learn-is-rtrancpl-cdcl_{NOT}) auto
have inv T
apply (rule rtrancpl-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 not-simplified-cls (clauses U) $\subseteq \#$ not-simplified-cls (clauses T)
using (U \sim reduce-trail-to_{NOT} [] T) by auto
moreover have not-simplified-cls (clauses T) $\subseteq \#$ not-simplified-cls (clauses S)
apply (rule rtrancpl-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing)
using (cdcl_{NOT}-merged-bj-learn $\widetilde{\sim}$ m) S T by (auto dest!: relpowp-imp-rtrancpl)
ultimately have U-S: not-simplified-cls (clauses U) $\subseteq \#$ not-simplified-cls (clauses S)
 by auto
then show ?case by (auto simp: card-mono set-mset-mono)

qed

sublocale cdcl_{NOT}-increasing-restarts - - - - - f $\lambda S T. T \sim$ reduce-trail-to_{NOT} ([::'a list) S
 $\lambda A S. \text{atms-of-msu (clauses S)} \subseteq \text{atms-of-ms A}$
 $\wedge \text{atm-of ' lits-of (trail S)} \subseteq \text{atms-of-ms A} \wedge \text{finite A}$

μ_{CDCL}' -merged $cdcl_{NOT}$ -merged-bj-learn
 $\lambda S. inv\ S \wedge no_dup\ (trail\ S)$
 $\lambda A\ T. ((2 + card\ (atms_of_ms\ A)) \wedge (1 + card\ (atms_of_ms\ A))) * 2$
 $+ card\ (set_mset\ (not_simplified_cls(c\lause s\ T)))$
 $+ 3 \wedge card\ (atms_of_ms\ A)$
apply *unfold-locales*
using $cdcl_{NOT}$ -restart- μ_{CDCL}' -merged-le- μ_{CDCL}' -bound **apply** *force*
using $cdcl_{NOT}$ -restart- μ_{CDCL}' -bound-le- μ_{CDCL}' -bound **by** *fastforce*

lemma $cdcl_{NOT}$ -restart-eq-sat-iff:
assumes
 $cdcl_{NOT}$ -restart $S\ T$ **and**
 $no_dup\ (trail\ (fst\ S))$
 $inv\ (fst\ S)$
shows $I \models_{sextm} clauses\ (fst\ S) \longleftrightarrow I \models_{sextm} clauses\ (fst\ T)$
using *assms*

proof (*induction rule: $cdcl_{NOT}$ -restart.induct*)
case (*restart-full* $S\ T\ n$)
then have $cdcl_{NOT}$ -merged-bj-learn** $S\ T$
by (*simp add: tranclp-into-rtranclp full1-def*)
then show ?*case*
using $cdcl_{NOT}$.*rtranclp-cdcl_{NOT}-bj-sat-ext-iff restart-full.prem*s(1,2)
 $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-is- $rtranclp$ - $cdcl_{NOT}$ **by** *auto*

next
case (*restart-step* $m\ S\ T\ n\ U$)
then have $cdcl_{NOT}$ -merged-bj-learn** $S\ T$
by (*auto simp: tranclp-into-rtranclp full1-def dest!: relpowp-imp-rtranclp*)
then have $I \models_{sextm} clauses\ S \longleftrightarrow I \models_{sextm} clauses\ T$
using $cdcl_{NOT}$.*rtranclp-cdcl_{NOT}-bj-sat-ext-iff restart-step.prem*s(1,2)
 $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-is- $rtranclp$ - $cdcl_{NOT}$ **by** *auto*
moreover have $I \models_{sextm} clauses\ T \longleftrightarrow I \models_{sextm} clauses\ U$
using *restart-step.hyps*(3) **by** *auto*
ultimately show ?*case* **by** *auto*

qed

lemma $rtranclp$ - $cdcl_{NOT}$ -restart-eq-sat-iff:
assumes
 $cdcl_{NOT}$ -restart** $S\ T$ **and**
 $inv: inv\ (fst\ S)$ **and** $n-d: no_dup(trail\ (fst\ S))$
shows $I \models_{sextm} clauses\ (fst\ S) \longleftrightarrow I \models_{sextm} clauses\ (fst\ T)$
using *assms*(1)

proof (*induction rule: $rtranclp$ -induct*)
case *base*
then show ?*case* **by** *simp*

next
case (*step* $T\ U$) **note** $st = this(1)$ **and** $cdcl = this(2)$ **and** $IH = this(3)$
have $inv\ (fst\ T)$ **and** $no_dup\ (trail\ (fst\ T))$
using $rtranclp$ - $cdcl_{NOT}$ -with-restart- $cdcl_{NOT}$ -inv **using** $st\ inv\ n-d$ **by** *blast+*
then have $I \models_{sextm} clauses\ (fst\ T) \longleftrightarrow I \models_{sextm} clauses\ (fst\ U)$
using $cdcl_{NOT}$ -restart-eq-sat-iff $cdcl$ **by** *blast*
then show ?*case* **using** IH **by** *blast*

qed

lemma $cdcl_{NOT}$ -restart-all-decomposition-implies-m:
assumes

```

  cdclNOT-restart  $S$   $T$  and
  inv: inv (fst  $S$ ) and  $n$ -d: no-dup(trail (fst  $S$ )) and
  all-decomposition-implies-m (clauses (fst  $S$ ))
    (get-all-marked-decomposition (trail (fst  $S$ )))
  shows all-decomposition-implies-m (clauses (fst  $T$ ))
    (get-all-marked-decomposition (trail (fst  $T$ )))
  using assms
proof (induction)
  case (restart-full  $S$   $T$   $n$ ) note full = this(1) and inv = this(2) and  $n$ -d = this(3) and
    decomp = this(4)
  have st: cdclNOT-merged-bj-learn**  $S$   $T$  and
     $n$ -s: no-step cdclNOT-merged-bj-learn  $T$ 
    using full unfolding full1-def by (fast dest: tranclp-into-rtranclp)+
  have st': cdclNOT**  $S$   $T$ 
    using inv rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT-and-inv st  $n$ -d by auto
  have inv  $T$ 
    using rtranclp-cdclNOT-cdclNOT-inv[OF st] inv  $n$ -d by auto
  then show ?case
    using cdclNOT.rtranclp-cdclNOT-all-decomposition-implies[OF - -  $n$ -d decomp] st' inv by auto
next
  case (restart-step  $m$   $S$   $T$   $n$   $U$ ) note st = this(1) and  $U$  = this(3) and inv = this(4) and
     $n$ -d = this(5) and decomp = this(6)
  show ?case using  $U$  by auto
qed

```

lemma rtranclp-cdcl_{NOT}-restart-all-decomposition-implies-m:

```

  assumes
    cdclNOT-restart**  $S$   $T$  and
    inv: inv (fst  $S$ ) and  $n$ -d: no-dup(trail (fst  $S$ )) and
    decomp: all-decomposition-implies-m (clauses (fst  $S$ ))
      (get-all-marked-decomposition (trail (fst  $S$ )))
  shows all-decomposition-implies-m (clauses (fst  $T$ ))
    (get-all-marked-decomposition (trail (fst  $T$ )))
  using assms
proof (induction)
  case base
  then show ?case using decomp by simp
next
  case (step  $T$   $U$ ) note st = this(1) and cdcl = this(2) and IH = this(3)[OF this(4-)] and
    inv = this(4) and  $n$ -d = this(5) and decomp = this(6)
  have inv (fst  $T$ ) and no-dup (trail (fst  $T$ ))
    using rtranclp-cdclNOT-with-restart-cdclNOT-inv using st inv  $n$ -d by blast+
  then show ?case
    using cdclNOT-restart-all-decomposition-implies-m[OF cdcl] IH by auto
qed

```

lemma full-cdcl_{NOT}-restart-normal-form:

```

  assumes
    full: full cdclNOT-restart  $S$   $T$  and
    inv: inv (fst  $S$ ) and  $n$ -d: no-dup(trail (fst  $S$ )) and
    decomp: all-decomposition-implies-m (clauses (fst  $S$ ))
      (get-all-marked-decomposition (trail (fst  $S$ ))) and
    atms-cls: atms-of-msu (clauses (fst  $S$ ))  $\subseteq$  atms-of-ms  $A$  and
    atms-trail: atm-of ' lits-of (trail (fst  $S$ ))  $\subseteq$  atms-of-ms  $A$  and
    fin: finite  $A$ 

```

shows *unsatisfiable* (*set-mset* (*clauses* (*fst* *S*)))
 \vee *lits-of* (*trail* (*fst* *T*)) \models_{sextm} *clauses* (*fst* *S*) \wedge *satisfiable* (*set-mset* (*clauses* (*fst* *S*)))
proof –
have *inv-T*: *inv* (*fst* *T*) **and** *n-d-T*: *no-dup* (*trail* (*fst* *T*))
using *rtrancpl-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv* **using** *full inv n-d unfolding full-def* **by** *blast+*
moreover have
atms-cls-T: *atms-of-msu* (*clauses* (*fst* *T*)) \subseteq *atms-of-ms* *A* **and**
atms-trail-T: *atm-of* ‘*lits-of* (*trail* (*fst* *T*)) \subseteq *atms-of-ms* *A*
using *rtrancpl-cdcl_{NOT}-with-restart-bound-inv*[*of S T A*] *full atms-cls atms-trail fin inv n-d*
unfolding *full-def* **by** *blast+*
ultimately have *no-step cdcl_{NOT}-merged-bj-learn* (*fst* *T*)
apply –
apply (*rule no-step-cdcl_{NOT}-restart-no-step-cdcl_{NOT}*[*of - A*])
using *full unfolding full-def* **apply** *simp*
apply *simp*
using *fin* **apply** *simp*
done
moreover have *all-decomposition-implies-m* (*clauses* (*fst* *T*))
(*get-all-marked-decomposition* (*trail* (*fst* *T*)))
using *rtrancpl-cdcl_{NOT}-restart-all-decomposition-implies-m*[*of S T*] *inv n-d decomp*
full unfolding full-def **by** *auto*
ultimately have *unsatisfiable* (*set-mset* (*clauses* (*fst* *T*)))
 \vee *trail* (*fst* *T*) \models_{asm} *clauses* (*fst* *T*) \wedge *satisfiable* (*set-mset* (*clauses* (*fst* *T*)))
apply –
apply (*rule cdcl_{NOT}-merged-bj-learn-final-state*)
using *atms-cls-T atms-trail-T fin n-d-T fin inv-T* **by** *blast+*
then consider
(*unsat*) *unsatisfiable* (*set-mset* (*clauses* (*fst* *T*)))
| (*sat*) *trail* (*fst* *T*) \models_{asm} *clauses* (*fst* *T*) **and** *satisfiable* (*set-mset* (*clauses* (*fst* *T*)))
by *auto*
then show *unsatisfiable* (*set-mset* (*clauses* (*fst* *S*)))
 \vee *lits-of* (*trail* (*fst* *T*)) \models_{sextm} *clauses* (*fst* *S*) \wedge *satisfiable* (*set-mset* (*clauses* (*fst* *S*)))
proof *cases*
case *unsat*
then have *unsatisfiable* (*set-mset* (*clauses* (*fst* *S*)))
unfolding *satisfiable-def* **apply** *auto*
using *rtrancpl-cdcl_{NOT}-restart-eq-sat-iff*[*of S T*] *full inv n-d*
consistent-true-clss-ext-satisfiable true-clss-imp-true-cls-ext
unfolding *satisfiable-def full-def* **by** *blast*
then show *?thesis* **by** *blast*
next
case *sat*
then have *lits-of* (*trail* (*fst* *T*)) \models_{sextm} *clauses* (*fst* *T*)
using *true-clss-imp-true-cls-ext* **by** (*auto simp: true-annots-true-cls*)
then have *lits-of* (*trail* (*fst* *T*)) \models_{sextm} *clauses* (*fst* *S*)
using *rtrancpl-cdcl_{NOT}-restart-eq-sat-iff*[*of S T*] *full inv n-d* **unfolding** *full-def* **by** *blast*
moreover then have *satisfiable* (*set-mset* (*clauses* (*fst* *S*)))
using *consistent-true-clss-ext-satisfiable distinctconsistent-interp n-d-T* **by** *fast*
ultimately show *?thesis* **by** *fast*
qed
qed

corollary *full-cdcl_{NOT}-restart-normal-form-init-state*:

assumes

init-state: *trail* *S* = [] *clauses* *S* = *N* **and**

```

    full: full cdclNOT-restart (S, 0) T and
    inv: inv S
shows unsatisfiable (set-mset N)
  ∨ lits-of (trail (fst T)) ⊨sextm N ∧ satisfiable (set-mset N)
using full-cdclNOT-restart-normal-form[of (S, 0) T] assms by auto

end

end
theory DPLL-NOT
imports CDCL-NOT
begin

```

15 DPLL as an instance of NOT

15.1 DPLL with simple backtrack

```

locale dpll-with-backtrack
begin
inductive backtrack :: ('v, unit, unit) marked-lit list × 'v clauses
  ⇒ ('v, unit, unit) marked-lit list × 'v clauses ⇒ bool where
  backtrack-split (fst S) = (M', L # M) ⇒ is-marked L ⇒ D ∈# snd S
  ⇒ fst S ⊨as CNot D ⇒ backtrack S (Propagated (− (lit-of L)) () # M, snd S)

inductive-cases backtrackE[elim]: backtrack (M, N) (M', N')
lemma backtrack-is-backjump:
  fixes M M' :: ('v, unit, unit) marked-lit list
  assumes
    backtrack: backtrack (M, N) (M', N') and
    no-dup: (no-dup ∘ fst) (M, N) and
    decomp: all-decomposition-implies-m N (get-all-marked-decomposition M)
  shows
    ∃ C F' K F L l C'.
      M = F' @ Marked K () # F ∧
      M' = Propagated L l # F ∧ N = N' ∧ C ∈# N ∧ F' @ Marked K d # F ⊨as CNot C ∧
      undefined-lit F L ∧ atm-of L ∈ atms-of-msu N ∪ atm-of ' lits-of (F' @ Marked K d # F) ∧
      N ⊨pm C' + {#L#} ∧ F ⊨as CNot C'

proof −
  let ?S = (M, N)
  let ?T = (M', N')
  obtain F F' P L D where
    b-sp: backtrack-split M = (F', L # F) and
    is-marked L and
    D ∈# snd ?S and
    M ⊨as CNot D and
    bt: backtrack ?S (Propagated (− (lit-of L)) P # F, N) and
    M': M' = Propagated (− (lit-of L)) P # F and
    [simp]: N' = N
  using backtrackE[OF backtrack] by (metis backtrack fstI sndI)
  let ?K = lit-of L
  let ?C = image-mset lit-of {#K ∈# mset M. is-marked K ∧ K ≠ L#} :: 'v literal multiset
  let ?C' = set-mset (image-mset single (?C + {#?K#}))
  obtain K where L: L = Marked K () using ⟨is-marked L⟩ by (cases L) auto

  have M: M = F' @ Marked K () # F

```

```

  using b-sp by (metis L backtrack-split-list-eq fst-conv snd-conv)
moreover have  $F' @ \text{Marked } K () \# F \models_{as} C \text{Not } D$ 
  using  $\langle M \models_{as} C \text{Not } D \rangle$  unfolding M .
moreover have undefined-lit F ( $- ?K$ )
  using no-dup unfolding M L by (simp add: defined-lit-map)
moreover have  $\text{atm-of } (-K) \in \text{atms-of-msu } N \cup \text{atm-of ' } \text{lit-of } (F' @ \text{Marked } K \text{ d } \# F)$ 
  by auto
moreover
  have  $\text{set-mset } N \cup ?C' \models_{ps} \{\{\#\}\}$ 
  proof -
    have A:  $\text{set-mset } N \cup ?C' \cup (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M =$ 
       $\text{set-mset } N \cup (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M$ 
    unfolding M L by auto
    have  $\text{set-mset } N \cup \{\{\#\text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
       $\models_{ps} (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M$ 
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
    moreover have  $C': ?C' = \{\{\#\text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
    unfolding M L apply standard
    apply force
    using IntI by auto
    ultimately have N-C-M:  $\text{set-mset } N \cup ?C' \models_{ps} (\lambda a. \{\#\text{lit-of } a\# \}) \text{ ' set } M$ 
    by auto
    have  $\text{set-mset } N \cup (\lambda L. \{\#\text{lit-of } L\# \}) \text{ ' (set } M) \models_{ps} \{\{\#\}\}$ 
    unfolding true-clss-clss-def
    proof (intro allI impI, goal-cases)
      case (1 I) note tot = this(1) and cons = this(2) and I-N-M = this(3)
      have I  $\models D$ 
      using I-N-M  $\langle D \in \# \text{ snd } ?S \rangle$  unfolding true-clss-def by auto
      moreover have I  $\models_s C \text{Not } D$ 
      using  $\langle M \models_{as} C \text{Not } D \rangle$  unfolding M by (metis 1(3)  $\langle M \models_{as} C \text{Not } D \rangle$ 
        true-annots-true-clss true-clss-mono-set-mset-l true-clss-def
        true-clss-singleton-lit-of-implies-incl true-clss-union)
      ultimately show ?case using cons consistent-CNot-not by blast
    qed
  then show ?thesis
    using true-clss-clss-left-right[OF N-C-M, of  $\{\{\#\}\}$ ] unfolding A by auto
  qed
have  $N \models_{pm} \text{image-mset } \text{uminus } ?C + \{\# - ?K\# \}$ 
unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
proof (intro allI impI)
  fix I
  assume
    tot: total-over-set I (atms-of-ms ( $\text{set-mset } N \cup \{\text{image-mset } \text{uminus } ?C + \{\# - ?K\# \}\}$ )) 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 total-over-set I (atm-of ' lit-of ' ( $\text{set } M \cap \{L. \text{is-marked } L \wedge L \neq \text{Marked } K \text{ d}\}$ ))
  using tot by (auto simp add: L atms-of-uminus-lit-atm-of-lit-of)

then have H:  $\bigwedge x.$ 
   $\text{lit-of } x \notin I \implies x \in \text{set } M \implies \text{is-marked } x$ 
   $\implies x \neq \text{Marked } K \text{ d} \implies -\text{lit-of } x \in I$ 

unfolding total-over-set-def atms-of-s-def

```



```

proof –
  fix  $x :: ('v, unit, unit) \text{ marked-lit}$ 
  assume  $a1: x \in \text{set } M$ 
  assume  $a2: \forall l \in \text{atm-of } ' \text{ lit-of } ' (\text{set } M \cap \{L. \text{ is-marked } L \wedge L \neq \text{Marked } K \ d\}).$ 
     $\text{Pos } l \in I \vee \text{Neg } l \in I$ 
  assume  $a3: \text{lit-of } x \notin I$ 
  assume  $a4: \text{is-marked } x$ 
  assume  $a5: x \neq \text{Marked } K \ d$ 
  have  $f6: \text{Neg } (\text{atm-of } (\text{lit-of } x)) = - \text{Pos } (\text{atm-of } (\text{lit-of } x))$ 
    by simp
  have  $\text{Pos } (\text{atm-of } (\text{lit-of } x)) \in I \vee \text{Neg } (\text{atm-of } (\text{lit-of } x)) \in I$ 
    using  $a5 \ a4 \ a2 \ a1$  by blast
  then show  $-\text{lit-of } x \in I$ 
    using  $f6 \ a3$  by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      literal.sel(1))
  qed
have  $\neg I \models_s ?C'$ 
  using  $\langle \text{set-mset } N \cup ?C' \models_{ps} \{\{\#\}\} \rangle \text{ tot cons } \langle I \models_{sm} N \rangle$ 
  unfolding true-clss-clss-def total-over-m-def
  by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-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-cls-def Bex-mset-def
  using  $\langle (K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I) \rangle$ 
  unfolding  $L$  by (auto dest!: H)
  qed
moreover
  have  $\text{set } F' \cap \{K. \text{ is-marked } K \wedge K \neq L\} = \{\}$ 
    using backtrack-split-fst-not-marked[of - M] b-sp by auto
  then have  $F \models_{as} \text{CNot } (\text{image-mset } \text{uminus } ?C)$ 
    unfolding  $M \ \text{CNot-def true-annots-def}$  by (auto simp add: L lits-of-def)
  ultimately show ?thesis
    using  $M' \langle D \in \# \text{ snd } ?S \rangle L$  by force
qed

lemma backtrack-is-backjump':
  fixes  $M \ M' :: ('v, unit, unit) \text{ marked-lit list}$ 
  assumes
    backtrack: backtrack S T and
    no-dup: (no-dup  $\circ$  fst) S and
    decomp: all-decomposition-implies-m (snd S) (get-all-marked-decomposition (fst S))
  shows
     $\exists C \ F' \ K \ F \ L \ l \ C'.$ 
     $\text{fst } S = F' @ \text{Marked } K \ () \ \# \ F \wedge$ 
     $T = (\text{Propagated } L \ l \ \# \ F, \text{snd } S) \wedge C \in \# \text{ snd } S \wedge \text{fst } S \models_{as} \text{CNot } C$ 
     $\wedge \text{undefined-lit } F \ L \wedge \text{atm-of } L \in \text{atms-of-msu } (\text{snd } S) \cup \text{atm-of } ' \text{ lits-of } (\text{fst } S) \wedge$ 
     $\text{snd } S \models_{pm} C' + \{\#L\#\} \wedge F \models_{as} \text{CNot } C'$ 
  apply (cases S, cases T)
  using backtrack-is-backjump[of fst S snd S fst T snd T] assms by fastforce

sublocale dpll-state fst snd  $\lambda L \ (M, N). (L \ \# \ M, N) \ \lambda(M, N). (tl \ M, N)$ 
   $\lambda C \ (M, N). (M, \{\#C\#\} + N) \ \lambda C \ (M, N). (M, \text{remove-mset } C \ N)$ 
by unfold-locales auto

sublocale backjumping-ops fst snd  $\lambda L \ (M, N). (L \ \# \ M, N) \ \lambda(M, N). (tl \ M, N)$ 
   $\lambda C \ (M, N). (M, \{\#C\#\} + N) \ \lambda C \ (M, N). (M, \text{remove-mset } C \ N) \ \lambda - - S \ T. \text{backtrack } S \ T$ 

```

by *unfold-locales*

lemma *backtrack-is-backjump''*:

fixes $M M' :: ('v, unit, unit) \text{ marked-lit list}$

assumes

backtrack: *backtrack* $S T$ **and**

no-dup: $(no\text{-}dup \circ fst) S$ **and**

decomp: *all-decomposition-implies-m* $(snd S)$ (*get-all-marked-decomposition* $(fst S)$)

shows *backjump* $S T$

proof –

obtain $C F' K F L l C'$ **where**

1: $fst S = F' @ \text{Marked } K () \# F$ **and**

2: $T = (\text{Propagated } L l \# F, snd S)$ **and**

3: $C \in \# snd S$ **and**

4: $fst S \models_{as} CNot C$ **and**

5: *undefined-lit* $F L$ **and**

6: $atm\text{-}of L \in atm\text{-}of\text{-}msu (snd S) \cup atm\text{-}of \text{ ' lits-}of (fst S)$ **and**

7: $snd S \models_{pm} C' + \{\#L\# \}$ **and**

8: $F \models_{as} CNot C'$

using *backtrack-is-backjump'*[*OF assms*] **by** *blast*

show *?thesis*

using *backjump.intros*[*OF 1 - 3 4 5 6 7 8*] 2 *backtrack 1 5*

by (*auto simp: state-eq_{NOT}-def simp del: state-simp_{NOT}*)

qed

lemma *can-do-bt-step*:

assumes

$M: fst S = F' @ \text{Marked } K d \# F$ **and**

$C \in \# snd S$ **and**

$C: fst S \models_{as} CNot C$

shows $\neg no\text{-}step \text{ backtrack } S$

proof –

obtain $L G' G$ **where**

backtrack-split $(fst S) = (G', L \# G)$

unfolding M **by** (*induction* F' *rule: marked-lit-list-induct*) *auto*

moreover then have *is-marked* L

by (*metis backtrack-split-snd-hd-marked list.distinct*(1) *list.sel*(1) *snd-conv*)

ultimately show *?thesis*

using *backtrack.intros*[*of S G' L G C*] $\langle C \in \# snd S \rangle C$ **unfolding** M **by** *auto*

qed

end

sublocale *dpll-with-backtrack* \subseteq *dpll-with-backjumping-ops* *fst snd* $\lambda L (M, N). (L \# M, N)$

$\lambda (M, N). (tl M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, \text{remove-mset } C N) \lambda - -. \text{True}$

$\lambda (M, N). no\text{-}dup M \wedge all\text{-}decomposition\text{-}implies\text{-}m N (\text{get-all-marked-decomposition } M)$

$\lambda - - S T. \text{backtrack } S T$

by *unfold-locales* (*metis* (*mono-tags, lifting*) *dpll-with-backtrack.backtrack-is-backjump''*

dpll-with-backtrack.can-do-bt-step prod.case-eq-if comp-apply)

sublocale *dpll-with-backtrack* \subseteq *dpll-with-backjumping* *fst snd* $\lambda L (M, N). (L \# M, N)$

$\lambda (M, N). (tl M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, \text{remove-mset } C N) \lambda - -. \text{True}$

$\lambda (M, N). no\text{-}dup M \wedge all\text{-}decomposition\text{-}implies\text{-}m N (\text{get-all-marked-decomposition } M)$

$\lambda - - S T. \text{backtrack } S T$

apply *unfold-locales*

using *dpll-bj-no-dup dpll-bj-all-decomposition-implies-inv* **apply** *fastforce*
done

sublocale *dpll-with-backtrack* \subseteq *conflict-driven-clause-learning-ops*

fst snd $\lambda L (M, N). (L \# M, N)$
 $\lambda(M, N). (tl\ M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, remove_mset\ C\ N) \lambda - -. True$
 $\lambda(M, N). no_dup\ M \wedge all_decomposition_implies_m\ N\ (get_all_marked_decomposition\ M)$
 $\lambda - - S\ T. backtrack\ S\ T \lambda - -. False \lambda - -. False$
by *unfold-locales*

sublocale *dpll-with-backtrack* \subseteq *conflict-driven-clause-learning*

fst snd $\lambda L (M, N). (L \# M, N)$
 $\lambda(M, N). (tl\ M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, remove_mset\ C\ N) \lambda - -. True$
 $\lambda(M, N). no_dup\ M \wedge all_decomposition_implies_m\ N\ (get_all_marked_decomposition\ M)$
 $\lambda - - S\ T. backtrack\ S\ T \lambda - -. False \lambda - -. False$
apply *unfold-locales*
using *cdcl_{NOT}.simps dpll-bj-inv forgetE learnE* **by** *blast*

context *dpll-with-backtrack*

begin

lemma *wf-tranclp-dpll-inital-state:*

assumes *fin: finite A*
shows *wf* $\{((M'::('v, unit, unit)\ marked_lits, N'::'v\ clauses), ([], N)) | M'\ N'\ N.$
 $dpll_bj^{++}\ ([], N)\ (M', N') \wedge atms_of_msu\ N \subseteq atms_of_ms\ A\}$
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* $(set_mset\ N) \vee (M' \models_{asm}\ N \wedge satisfiable\ (set_mset\ N))$
using *assms full-dpll-backjump-final-state[of ([],N) (M', N') set-mset N]* **by** *auto*

corollary *full-dpll-normal-form-from-init-state:*

fixes *M M' :: ('v, unit, unit) marked-lit list*
assumes
 $full: full\ dpll_bj\ ([], N)\ (M', N')$
shows $M' \models_{asm}\ N \longleftrightarrow satisfiable\ (set_mset\ N)$

proof –

have *no-dup M'*
using *rtranclp-dpll-bj-no-dup[of ([], N) (M', N')]*
 $full$ **unfolding** *full-def* **by** *auto*
then have $M' \models_{asm}\ N \implies satisfiable\ (set_mset\ N)$
using *distinctconsistent-interp satisfiable-carac' true-annots-true-cls* **by** *blast*
then show *?thesis*
using *full-dpll-final-state-conclusive[OF full]* **by** *auto*

qed

lemma *cdcl_{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* $\{(T, S). dpll_bj\ S\ T \wedge cdcl_{NOT}\ NOT_all_inv\ A\ S\}$
unfolding *cdcl_{NOT}-is-dpll[symmetric]*

```

by (rule wf-cdclNOT-no-learn-and-forget-infinite-chain)
(auto simp: learn.simps forgetNOT.simps)
end

```

15.2 Adding restarts

```

locale dpll-withbacktrack-and-restarts =
  dpll-with-backtrack +
  fixes f :: nat  $\Rightarrow$  nat
  assumes unbounded: unbounded f and f-ge-1:  $\bigwedge n. n \geq 1 \implies f\ n \geq 1$ 
begin
  sublocale cdclNOT-increasing-restarts fst snd  $\lambda L\ (M, N). (L \# M, N) \lambda(M, N). (tl\ M, N)$ 
     $\lambda C\ (M, N). (M, \{\#C\# \} + N) \lambda C\ (M, N). (M, remove-mset\ C\ N) f \lambda(-, N) S. S = ([], N)$ 
     $\lambda A\ (M, N). atms-of-msu\ N \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of\ M \subseteq atms-of-ms\ A \wedge finite\ A$ 
     $\wedge all-decomposition-implies-m\ N\ (get-all-marked-decomposition\ M)$ 
     $\lambda A\ T. (2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$ 
       $- \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ T)\ dpll-bj$ 
     $\lambda(M, N). no-dup\ M \wedge all-decomposition-implies-m\ N\ (get-all-marked-decomposition\ M)$ 
     $\lambda A\ -. (2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$ 
  apply unfold-locales
    apply (rule unbounded)
    using f-ge-1 apply fastforce
    apply (smt dpll-bj-all-decomposition-implies-inv dpll-bj-atms-in-trail-in-set
      dpll-bj-clauses dpll-bj-no-dup prod.case-eq-if)
    apply (rule dpll-bj-trail-mes-decreasing-prop; auto)
    apply (case-tac T, simp)
    apply (case-tac U, simp)
    using dpll-bj-clauses dpll-bj-all-decomposition-implies-inv dpll-bj-no-dup by fastforce+
end

end
theory DPLL-W
imports Main Partial-Clausal-Logic Partial-Annotated-Clausal-Logic List-More Wellfounded-More
  DPLL-NOT
begin

```

16 DPLL

16.1 Rules

```

type-synonym 'a dpllW-marked-lit = ('a, unit, unit) marked-lit
type-synonym 'a dpllW-marked-lits = ('a, unit, unit) marked-lits
type-synonym 'v dpllW-state = 'v dpllW-marked-lits  $\times$  'v clauses

```

```

abbreviation trail :: 'v dpllW-state  $\Rightarrow$  'v dpllW-marked-lits where
  trail  $\equiv$  fst
abbreviation clauses :: 'v dpllW-state  $\Rightarrow$  'v clauses where
  clauses  $\equiv$  snd

```

The definition of DPLL is given in figure 2.13 page 70 of CW.

```

inductive dpllW :: 'v dpllW-state  $\Rightarrow$  'v dpllW-state  $\Rightarrow$  bool where
  propagate:  $C + \{\#L\# \} \in \# clauses\ S \implies trail\ S \models_{as}\ CNot\ C \implies undefined-lit\ (trail\ S)\ L$ 
     $\implies dpll_W\ S\ (Propagated\ L\ ()\ \# trail\ S, clauses\ S) \mid$ 
  decided:  $undefined-lit\ (trail\ S)\ L \implies atm-of\ L \in atms-of-msu\ (clauses\ S)$ 
     $\implies dpll_W\ S\ (Marked\ L\ ()\ \# trail\ S, clauses\ S) \mid$ 

```

backtrack: $\text{backtrack-split } (\text{trail } S) = (M', L \# M) \implies \text{is-marked } L \implies D \in \# \text{ clauses } S$
 $\implies \text{trail } S \models_{\text{as}} \text{CNot } D \implies \text{dpll}_W S (\text{Propagated } (- (\text{lit-of } L)) () \# M, \text{ clauses } S)$

16.2 Invariants

lemma *dpll_W-distinct-inv*:

assumes *dpll_W S S'*
and *no-dup (trail S)*
shows *no-dup (trail S')*
using *assms*

proof (*induct rule: dpll_W.induct*)

case (*decided L S*)

then show *?case* **using** *defined-lit-map* **by force**

next

case (*propagate C L S*)

then show *?case* **using** *defined-lit-map* **by force**

next

case (*backtrack S M' L M D*) **note** *extracted = this(1)* **and** *no-dup = this(5)*

show *?case*

using *no-dup backtrack-split-list-eq[of trail S, symmetric]* **unfolding** *extracted* **by auto**

qed

lemma *dpll_W-consistent-interp-inv*:

assumes *dpll_W S S'*

and *consistent-interp (lits-of (trail S))*

and *no-dup (trail S)*

shows *consistent-interp (lits-of (trail S'))*

using *assms*

proof (*induct rule: dpll_W.induct*)

case (*backtrack S M' L M D*) **note** *extracted = this(1)* **and** *marked = this(2)* **and** *D = this(4)* **and**
cons = this(5) **and** *no-dup = this(6)*

have *no-dup'*: *no-dup M*

by (*metis (no-types) backtrack-split-list-eq distinct.simps(2) distinct-append extracted*
list.simps(9) map-append no-dup snd-conv)

then have *insert (lit-of L) (lits-of M) ⊆ lits-of (trail S)*

using *backtrack-split-list-eq[of trail S, symmetric]* **unfolding** *extracted* **by auto**

then have *cons*: *consistent-interp (insert (lit-of L) (lits-of M))*

using *consistent-interp-subset cons* **by blast**

moreover

have *lit-of L ∉ lits-of M*

using *no-dup backtrack-split-list-eq[of trail S, symmetric]* *extracted*
unfolding *lits-of-def* **by force**

moreover

have *atm-of (−lit-of L) ∉ (λm. atm-of (lit-of m))* ‘ *set M*

using *no-dup backtrack-split-list-eq[of trail S, symmetric]* **unfolding** *extracted* **by force**

then have *−lit-of L ∉ lits-of M*

unfolding *lits-of-def* **by force**

ultimately show *?case* **by simp**

qed (*auto intro: consistent-add-undefined-lit-consistent*)

lemma *dpll_W-vars-in-snd-inv*:

assumes *dpll_W S S'*

and *atm-of ‘ (lits-of (trail S)) ⊆ atms-of-msu (clauses S)*

shows *atm-of ‘ (lits-of (trail S')) ⊆ atms-of-msu (clauses S')*

using *assms*

proof (*induct rule: dpll_W.induct*)

```

case (backtrack S M' L M D)
then have atm-of (lit-of L) ∈ atms-of-msu (clauses S)
  using backtrack-split-list-eq[of trail S, symmetric] by auto
moreover
  have atm-of ' lits-of (trail S) ⊆ atms-of-msu (clauses S)
    using backtrack(5) by simp
  then have ∧xb. xb ∈ set M ⇒ atm-of (lit-of xb) ∈ atms-of-msu (clauses S)
    using backtrack-split-list-eq[symmetric, of trail S] backtrack.hyps(1)
    unfolding lits-of-def by auto
  ultimately show ?case by (auto simp : lits-of-def)
qed (auto simp: in-plus-implies-atm-of-on-atms-of-ms)

```

lemma *atms-of-ms-lit-of-atms-of*: *atms-of-ms* (($\lambda a. \{\# \text{lit-of } a \# \}$) ' c) = *atm-of* ' *lit-of* ' c
unfolding *atms-of-ms-def* **using** *image-iff* **by** *force*

Lemma theorem 2.8.2 page 71 of CW

lemma *dpll_W-propagate-is-conclusion*:

```

assumes dpllW S S'
and all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
and atm-of ' lits-of (trail S) ⊆ atms-of-msu (clauses S)
shows all-decomposition-implies-m (clauses S') (get-all-marked-decomposition (trail S'))
using assms

```

proof (*induct rule: dpll_W.induct*)

case (*decided L S*)

then show ?case **unfolding** *all-decomposition-implies-def* **by** *simp*

next

case (*propagate C L S*) **note** *inS* = *this*(1) **and** *cnot* = *this*(2) **and** *IH* = *this*(4) **and** *undef* = *this*(3) **and** *atms-incl* = *this*(5)

let ?*I* = *set* (*map* ($\lambda a. \{\# \text{lit-of } a \# \}$) (*trail S*)) ∪ *set-mset* (*clauses S*)

have ?*I* ⊢_p C + { $\#L\#$ } **by** (*auto simp add: inS*)

moreover have ?*I* ⊢_{ps} CNot C **using** *true-annots-true-clss-cls cnot* **by** *fastforce*

ultimately have ?*I* ⊢_p { $\#L\#$ } **using** *true-clss-cls-plus-CNot*[of ?*I* C L] *inS* **by** *blast*

```

{
  assume get-all-marked-decomposition (trail S) = []
  then have ?case by blast
}

```

moreover {

assume *n*: *get-all-marked-decomposition* (*trail S*) ≠ []

have 1: $\bigwedge a b. (a, b) \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } (\text{trail } S)))$

⇒ (($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* a ∪ *set-mset* (*clauses S*)) ⊢_{ps} ($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* b
using *IH* **unfolding** *all-decomposition-implies-def* **by** (*fastforce simp add: list.set-sel*(2) *n*)

moreover have 2: $\bigwedge a c. \text{hd } (\text{get-all-marked-decomposition } (\text{trail } S)) = (a, c)$

⇒ (($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* a ∪ *set-mset* (*clauses S*)) ⊢_{ps} (($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* c)

by (*metis IH all-decomposition-implies-cons-pair all-decomposition-implies-single list.collapse n*)

moreover have 3: $\bigwedge a c. \text{hd } (\text{get-all-marked-decomposition } (\text{trail } S)) = (a, c)$

⇒ (($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* a ∪ *set-mset* (*clauses S*)) ⊢_p { $\#L\#$ }

proof –

fix *a c*

assume *h*: *hd* (*get-all-marked-decomposition* (*trail S*)) = (*a*, *c*)

have *h'*: *trail S* = *c* @ *a* **using** *get-all-marked-decomposition-decomp h* **by** *blast*

have *I*: *set* (*map* ($\lambda a. \{\# \text{lit-of } a \# \}$) *a*) ∪ *set-mset* (*clauses S*)

∪ ($\lambda a. \{\# \text{lit-of } a \# \}$) ' *set* c ⊢_{ps} CNot C

using (?*I* ⊢_{ps} CNot C) **unfolding** *h'* **by** (*simp add: Un-commute Un-left-commute*)

have

```

atms-of-ms (CNot C) ⊆ atms-of-ms (set (map (λa. {#lit-of a#}) a) ∪ set-mset (clauses S))
and
atms-of-ms ((λa. {#lit-of a#}) ‘ set c) ⊆ atms-of-ms (set (map (λa. {#lit-of a#}) a)
  ∪ set-mset (clauses S))
apply (metis CNot-plus Un-subset-iff atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of
  atms-of-ms-union inS mem-set-mset-iff sup.coboundedI2)
using inS atms-of-atms-of-ms-mono atms-incl by (fastforce simp: h′)

then have (λa. {#lit-of a#}) ‘ set a ∪ set-mset (clauses S) ⊨ps CNot C
  using true-clss-clss-left-right[OF - I] h 2 by auto
then show (λa. {#lit-of a#}) ‘ set a ∪ set-mset (clauses S) ⊨p {#L#}
  by (metis (no-types) Un-insert-right inS insertI1 mk-disjoint-insert inS mem-set-mset-iff
    true-clss-clss-in true-clss-clss-plus-CNot)
qed
ultimately have ?case
  by (case-tac hd (get-all-marked-decomposition (trail S)))
    (auto simp add: all-decomposition-implies-def)
}
ultimately show ?case by auto
next
case (backtrack S M′ L M D) note extracted = this(1) and marked = this(2) and D = this(3) and
  cnot = this(4) and cons = this(4) and IH = this(5) and atms-incl = this(6)
have S: trail S = M′ @ L # M
  using backtrack-split-list-eq[of trail S] unfolding extracted by auto
have M′: ∀ l ∈ set M′. ¬is-marked l
  using extracted backtrack-split-fst-not-marked[of - trail S] by simp
have n: get-all-marked-decomposition (trail S) ≠ [] by auto
then have all-decomposition-implies-m (clauses S) ((L # M, M′)
  # tl (get-all-marked-decomposition (trail S)))
  by (metis (no-types) IH extracted get-all-marked-decomposition-backtrack-split list.exhaust-sel)
then have 1: (λa. {#lit-of a#}) ‘ set (L # M) ∪ set-mset (clauses S) ⊨ps(λa. {#lit-of a#}) ‘ set
M′
  by simp
moreover
have (λa. {#lit-of a#}) ‘ set (L # M) ∪ (λa. {#lit-of a#}) ‘ set M′ ⊨ps CNot D
  by (metis (mono-tags, lifting) S Un-commute cons image-Un set-append
    true-annots-true-clss-clss)
then have 2: (λa. {#lit-of a#}) ‘ set (L # M) ∪ set-mset (clauses S) ∪ (λa. {#lit-of a#}) ‘ set
M′
  ⊨ps CNot D
  by (metis (no-types, lifting) Un-assoc Un-left-commute true-clss-clss-union-l-r)
ultimately
have set (map (λa. {#lit-of a#}) (L # M)) ∪ set-mset (clauses S) ⊨ps CNot D
  using true-clss-clss-left-right by fastforce
then have set (map (λa. {#lit-of a#}) (L # M)) ∪ set-mset (clauses S) ⊨p {#}
  by (metis (mono-tags, lifting) D Un-def mem-Collect-eq set-mset-def
    true-clss-clss-contradiction-true-clss-clss-false)
then have IL: (λa. {#lit-of a#}) ‘ set M ∪ set-mset (clauses S) ⊨p {#-lit-of L#}
  using true-clss-clss-false-left-right by auto
show ?case unfolding S all-decomposition-implies-def
proof
fix x P level
assume x: x ∈ set (get-all-marked-decomposition
  (fst (Propagated (− lit-of L) P # M, clauses S)))
let ?M′ = Propagated (− lit-of L) P # M

```

```

let ?hd = hd (get-all-marked-decomposition ?M')
let ?tl = tl (get-all-marked-decomposition ?M')
have x = ?hd  $\vee$  x  $\in$  set ?tl
  using x
  by (cases get-all-marked-decomposition ?M')
    auto
moreover {
  assume x': x  $\in$  set ?tl
  have L': Marked (lit-of L) () = L using marked by (case-tac L, auto)
  have x  $\in$  set (get-all-marked-decomposition (M' @ L # M))
    using x' get-all-marked-decomposition-except-last-choice-equal[of M' lit-of L P M]
    L' by (metis (no-types) M' list.set-sel(2) tl-Nil)
  then have case x of (Ls, seen)  $\Rightarrow$  ( $\lambda a.$  {#lit-of a#}) ' set Ls  $\cup$  set-mset (clauses S)
     $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set seen
    using marked IH by (case-tac L) (auto simp add: S all-decomposition-implies-def)
}
moreover {
  assume x': x = ?hd
  have tl: tl (get-all-marked-decomposition (M' @ L # M))  $\neq$  []
  proof -
    have f1:  $\bigwedge ms.$  length (get-all-marked-decomposition (M' @ ms))
      = length (get-all-marked-decomposition ms)
    by (simp add: M' get-all-marked-decomposition-remove-unmarked-length)
    have Suc (length (get-all-marked-decomposition M))  $\neq$  Suc 0
    by blast
    then show ?thesis
      using f1 marked by (metis (no-types) get-all-marked-decomposition.simps(1) length-tl
        list.sel(3) list.size(3) marked-lit.collapse(1))
  qed
  obtain M0' M0 where
    L0: hd (tl (get-all-marked-decomposition (M' @ L # M))) = (M0, M0')
    by (cases hd (tl (get-all-marked-decomposition (M' @ L # M))))
  have x'': x = (M0, Propagated ( $-$ lit-of L) P # M0')
    unfolding x' using get-all-marked-decomposition-last-choice tl M' L0
    by (metis marked marked-lit.collapse(1))
  obtain l-get-all-marked-decomposition where
    get-all-marked-decomposition (trail S) = (L # M, M') # (M0, M0') #
    l-get-all-marked-decomposition
    using get-all-marked-decomposition-backtrack-split extracted by (metis (no-types) L0 S
      hd-Cons-tl n tl)
  then have M = M0' @ M0 using get-all-marked-decomposition-hd-hd by fastforce
  then have IL': ( $\lambda a.$  {#lit-of a#}) ' set M0  $\cup$  set-mset (clauses S)
     $\cup$  ( $\lambda a.$  {#lit-of a#}) ' set M0'  $\models_{ps}$  {#- lit-of L#}
    using IL by (simp add: Un-commute Un-left-commute image-Un)
  moreover have H: ( $\lambda a.$  {#lit-of a#}) ' set M0  $\cup$  set-mset (clauses S)
     $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set M0'
    using IH x'' unfolding all-decomposition-implies-def by (metis (no-types, lifting) L0 S
      list.set-sel(1) list.set-sel(2) old.prod.case tl tl-Nil)
  ultimately have case x of (Ls, seen)  $\Rightarrow$  ( $\lambda a.$  {#lit-of a#}) ' set Ls  $\cup$  set-mset (clauses S)
     $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set seen
    using true-clss-clss-left-right unfolding x'' by auto
}
ultimately show case x of (Ls, seen)  $\Rightarrow$ 
  ( $\lambda a.$  {#lit-of a#}) ' set Ls  $\cup$  set-mset (snd (?M', clauses S))
   $\models_{ps}$  ( $\lambda a.$  {#lit-of a#}) ' set seen

```


unfolding *snd-conv* by *blast*
 qed
 qed

Lemma theorem 2.8.3 page 72 of CW

theorem *dpll_W-propagate-is-conclusion-of-decided*:
 assumes *dpll_W S S'*
 and *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*
 and *atm-of ' lits-of (trail S) \subseteq atms-of-msu (clauses S)*
 shows *set-mset (clauses S') $\cup \{\{\#lit-of L\# \mid L. is-marked L \wedge L \in set (trail S')\}\}$*
 $\models_{ps} (\lambda a. \{\#lit-of a\# \}) ' \bigcup (set ' snd ' set (get-all-marked-decomposition (trail S')))$
 using *all-decomposition-implies-trail-is-implied[OF dpll_W-propagate-is-conclusion[OF assms]]* .

Lemma theorem 2.8.4 page 72 of CW

lemma *only-propagated-vars-unsat*:
 assumes *marked: $\forall x \in set M. \neg is-marked x$*
 and *DN: $D \in N$ and $D: M \models_{as} CNot D$*
 and *inv: all-decomposition-implies N (get-all-marked-decomposition M)*
 and *atm-incl: atm-of ' lits-of M \subseteq atms-of-ms N*
 shows *unsatisfiable N*
proof (rule *ccontr*)
 assume $\neg unsatisfiable N$
 then obtain *I* where
 $I: I \models_s N$ and
cons: consistent-interp I and
tot: total-over-m I N
 unfolding *satisfiable-def* by *auto*
 then have *I-D: $I \models D$*
 using *DN* unfolding *true-clss-def* by *auto*

 have *l0: $\{\{\#lit-of L\# \mid L. is-marked L \wedge L \in set M\} = \{\}\}$* using *marked* by *auto*
 have *atms-of-ms (N $\cup (\lambda a. \{\#lit-of a\# \}) ' set M) = atms-of-ms N$*
 using *atm-incl* unfolding *atms-of-ms-def lits-of-def* by *auto*

 then have *total-over-m I (N $\cup (\lambda a. \{\#lit-of a\# \}) ' (set M)$)*
 using *tot* unfolding *total-over-m-def* by *auto*
 then have $I \models_s (\lambda a. \{\#lit-of a\# \}) ' (set M)$
 using *all-decomposition-implies-propagated-lits-are-implied[OF inv] cons I*
 unfolding *true-clss-clss-def l0* by *auto*
 then have *IM: $I \models_s (\lambda a. \{\#lit-of a\# \}) ' set M$* by *auto*
 {
 fix *K*
 assume $K \in \# D$
 then have $-K \in lits-of M$
 by (auto split: *split-if-asm*)
 intro: *allE[OF D[unfolded true-annots-def Ball-def], of $\{\#-K\# \}$]*
 then have $-K \in I$ using *IM true-clss-singleton-lit-of-implies-incl* by *fastforce*
 }
 then have $\neg I \models D$ using *cons* unfolding *true-clss-def consistent-interp-def* by *auto*
 then show *False* using *I-D* by *blast*
 qed

lemma *dpll_W-same-clauses*:
 assumes *dpll_W S S'*
 shows *clauses S = clauses S'*

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

lemma *rtrancpl-dpll_W-inv*:

assumes *rtrancpl dpll_W S S'*

and *inv*: *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))*

and *atm-incl*: *atm-of ' lits-of (trail S) \subseteq atms-of-msu (clauses S)*

and *consistent-interp (lits-of (trail S))*

and *no-dup (trail S)*

shows *all-decomposition-implies-m (clauses S') (get-all-marked-decomposition (trail S'))*

and *atm-of ' lits-of (trail S') \subseteq atms-of-msu (clauses S')*

and *clauses S = clauses S'*

and *consistent-interp (lits-of (trail S'))*

and *no-dup (trail S')*

using *assms*

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

case *base*

show

all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S)) and

atm-of ' lits-of (trail S) \subseteq atms-of-msu (clauses S) and

clauses S = clauses S and

consistent-interp (lits-of (trail S)) and

no-dup (trail S) using assms by auto

next

case (*step S' S''*) **note** *dpll_WStar = this(1)* **and** *IH = this(3,4,5,6,7)* **and**

dpll_W = this(2)

moreover

assume

inv: *all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S)) and*

atm-incl: *atm-of ' lits-of (trail S) \subseteq atms-of-msu (clauses S) and*

cons: *consistent-interp (lits-of (trail S)) and*

no-dup (trail S)

ultimately have *decomp*: *all-decomposition-implies-m (clauses S')*

(get-all-marked-decomposition (trail S')) and

atm-incl': *atm-of ' lits-of (trail S') \subseteq atms-of-msu (clauses S') and*

snd: *clauses S = clauses S' and*

cons': *consistent-interp (lits-of (trail S')) and*

no-dup': *no-dup (trail S') by blast+*

show *clauses S = clauses S'' using dpll_W-same-clauses[OF dpll_W] snd by metis*

show *all-decomposition-implies-m (clauses S'') (get-all-marked-decomposition (trail S''))*

using *dpll_W-propagate-is-conclusion[OF dpll_W] decomp atm-incl' by auto*

show *atm-of ' lits-of (trail S'') \subseteq atms-of-msu (clauses S'')*

using *dpll_W-vars-in-snd-inv[OF dpll_W] atm-incl atm-incl' by auto*

show *no-dup (trail S'') using dpll_W-distinct-inv[OF dpll_W] no-dup' dpll_W by auto*

show *consistent-interp (lits-of (trail S''))*

using *cons' no-dup' dpll_W-consistent-interp-inv[OF dpll_W] by auto*

qed

definition *dpll_W-all-inv S* \equiv

(all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S)))

\wedge *atm-of ' lits-of (trail S) \subseteq atms-of-msu (clauses S)*

\wedge *consistent-interp (lits-of (trail S))*

\wedge *no-dup (trail S)*

lemma *dpll_W-all-inv-dest[dest]*:

assumes $dpll_W\text{-all-inv } S$
shows $\text{all-decomposition-implies-}m \text{ (clauses } S) \text{ (get-all-marked-decomposition (trail } S))}$
and $\text{atm-of ' lits-of (trail } S) \subseteq \text{atms-of-msu (clauses } S)}$
and $\text{consistent-interp (lits-of (trail } S)) \wedge \text{no-dup (trail } S)}$
using *assms* **unfolding** $dpll_W\text{-all-inv-def lits-of-def}$ **by** *auto*

lemma $rtranclp\text{-}dpll_W\text{-all-inv}$:
assumes $rtranclp\ dpll_W\ S\ S'$
and $dpll_W\text{-all-inv } S$
shows $dpll_W\text{-all-inv } S'$
using *assms* $rtranclp\text{-}dpll_W\text{-inv}[OF\ assms(1)]$ **unfolding** $dpll_W\text{-all-inv-def lits-of-def}$ **by** *blast*

lemma $dpll_W\text{-all-inv}$:
assumes $dpll_W\ S\ S'$
and $dpll_W\text{-all-inv } S$
shows $dpll_W\text{-all-inv } S'$
using *assms* $rtranclp\text{-}dpll_W\text{-all-inv}$ **by** *blast*

lemma $rtranclp\text{-}dpll_W\text{-inv-starting-from-0}$:
assumes $rtranclp\ dpll_W\ S\ S'$
and $\text{inv: trail } S = []$
shows $dpll_W\text{-all-inv } S'$
proof –
have $dpll_W\text{-all-inv } S$
using *assms* **unfolding** $\text{all-decomposition-implies-def } dpll_W\text{-all-inv-def}$ **by** *auto*
then show $?thesis$ **using** $rtranclp\text{-}dpll_W\text{-all-inv}[OF\ assms(1)]$ **by** *blast*
qed

lemma $dpll_W\text{-can-do-step}$:
assumes $\text{consistent-interp (set } M)$
and $\text{distinct } M$
and $\text{atm-of ' (set } M) \subseteq \text{atms-of-msu } N$
shows $rtranclp\ dpll_W\ ([], N) (\text{map } (\lambda M. \text{Marked } M ())\ M, N)$
using *assms*
proof (*induct* M)
case *Nil*
then show $?case$ **by** *auto*
next
case (*Cons* $L\ M$)
then have $\text{undefined-lit (map } (\lambda M. \text{Marked } M ())\ M) L$
unfolding $\text{defined-lit-def consistent-interp-def}$ **by** *auto*
moreover have $\text{atm-of } L \in \text{atms-of-msu } N$ **using** $\text{Cons.prem}(3)$ **by** *auto*
ultimately have $dpll_W\ (\text{map } (\lambda M. \text{Marked } M ())\ M, N) (\text{map } (\lambda M. \text{Marked } M ())\ (L \# M), N)$
using $dpll_W.\text{decided}$ **by** *auto*
moreover have $\text{consistent-interp (set } M)$ **and** $\text{distinct } M$ **and** $\text{atm-of ' set } M \subseteq \text{atms-of-msu } N$
using Cons.prem **unfolding** $\text{consistent-interp-def}$ **by** *auto*
ultimately show $?case$ **using** Cons.hyps **by** *auto*
qed

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

lemma $dpll_W\text{-strong-completeness}$:

```

assumes set  $M \models_{sm} N$ 
and consistent-interp (set  $M$ )
and distinct  $M$ 
and atm-of ' (set  $M$ )  $\subseteq$  atms-of-msu  $N$ 
shows  $dpll_W^{**} ([], N) (map (\lambda M. \text{Marked } M ()) M, N)$ 
and conclusive-dpllW-state (map ( $\lambda M. \text{Marked } M ()$ )  $M, N$ )
proof –
  show rtrancpl  $dpll_W ([], N) (map (\lambda M. \text{Marked } M ()) M, N)$  using dpllW-can-do-step assms by auto
  have map ( $\lambda M. \text{Marked } M ()$ )  $M \models_{asm} N$  using assms(1) true-annots-marked-true-cls by auto
  then show conclusive-dpllW-state (map ( $\lambda M. \text{Marked } M ()$ )  $M, N$ )
    unfolding conclusive-dpllW-state-def by auto
qed

```

lemma *dpll_W-sound*:

```

assumes
  rtrancpl  $dpll_W ([], N) (M, N)$  and
   $\forall S. \neg dpll_W (M, N) S$ 
shows  $M \models_{asm} N \longleftrightarrow \text{satisfiable } (set-mset\ N)$  (is  $?A \longleftrightarrow ?B$ )
proof
  let  $?M' = \text{lits-of } M$ 
  assume  $?A$ 
  then have  $?M' \models_{sm} N$  by (simp add: true-annots-true-cls)
  moreover have consistent-interp  $?M'$ 
    using rtrancpl-dpllW-inv-starting-from-0[OF assms(1)] by auto
  ultimately show  $?B$  by auto
next
  assume  $?B$ 
  show  $?A$ 
  proof (rule ccontr)
    assume  $n: \neg ?A$ 
    have  $(\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of-msu } N) \vee (\exists D \in \#N. M \models_{as} CNot\ D)$ 
    proof –
      obtain  $D :: 'a\ \text{clause}$  where  $D: D \in \# N$  and  $\neg M \models_a D$ 
      using  $n$  unfolding true-annots-def Ball-def by auto
      then have  $(\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of } D) \vee M \models_{as} CNot\ D$ 
      unfolding true-annots-def Ball-def CNot-def true-annot-def
      using atm-of-lit-in-atms-of true-annot-iff-marked-or-true-lit true-cls-def by blast
      then show  $?thesis$ 

      using  $D$  apply auto by (meson atms-of-atms-of-ms-mono mem-set-mset-iff subset-eq)
    qed
  moreover {
    assume  $\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of-msu } N$ 
    then have False using assms(2) decided by fastforce
  }
  moreover {
    assume  $\exists D \in \#N. M \models_{as} CNot\ D$ 
    then obtain  $D$  where  $DN: D \in \# N$  and  $MD: M \models_{as} CNot\ D$  by auto
    {
      assume  $\forall l \in set\ M. \neg \text{is-marked } l$ 
      moreover have dpllW-all-inv ( $[], N$ )
      using assms unfolding all-decomposition-implies-def dpllW-all-inv-def by auto
      ultimately have unsatisfiable (set-mset  $N$ )
      using only-propagated-vars-unsat[of  $M\ D\ set-mset\ N$ ]  $DN\ MD$ 
    }
  }

```

```

      rtrancp-dpllW-all-inv[OF assms(1)] by force
    then have False using ⟨?B⟩ by blast
  }
  moreover {
    assume l: ∃ l ∈ set M. is-marked l
    then have False
      using backtrack[of (M, N) - - - D ] DN MD assms(2)
        backtrack-split-some-is-marked-then-snd-has-hd[OF l]
      by (metis backtrack-split-snd-hd-marked fst-conv list.distinct(1) list.sel(1) snd-conv)
  }
  ultimately have False by blast
}
ultimately show False by blast
qed
qed

```

16.3 Termination

definition *dpll_W-mes* $M\ n =$

map ($\lambda l.$ if is-marked l then 2 else (1::nat)) (rev M) @ replicate ($n - \text{length } M$) 3

lemma *length-dpll_W-mes*:

assumes *length* $M \leq n$

shows *length* (*dpll_W-mes* $M\ n$) = n

using *assms* **unfolding** *dpll_W-mes-def* **by** *auto*

lemma *distinctcard-atm-of-lits-of-eq-length*:

assumes *no-dup* S

shows *card* (*atm-of* ‘ *lits-of* S) = *length* S

using *assms* **by** (*induct* S) (*auto simp add: image-image lits-of-def*)

lemma *dpll_W-card-decrease*:

assumes *dpll*: *dpll_W* $S\ S'$ **and** *length* (*trail* S') ≤ *card vars*

and *length* (*trail* S) ≤ *card vars*

shows (*dpll_W-mes* (*trail* S') (*card vars*), *dpll_W-mes* (*trail* S) (*card vars*))
 ∈ *lexn* {(a, b). $a < b$ } (*card vars*)

using *assms*

proof (*induct rule: dpll_W.induct*)

case (*propagate* $C\ L\ S$)

have m : *map* ($\lambda l.$ if is-marked l then 2 else 1) (rev (*trail* S))

@ replicate (*card vars* − *length* (*trail* S)) 3

= *map* ($\lambda l.$ if is-marked l then 2 else 1) (rev (*trail* S)) @ 3

replicate (*card vars* − Suc (*length* (*trail* S))) 3

using *propagate.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*

then show ?*case*

using *propagate.prem[s(1)]* **unfolding** *dpll_W-mes-def* **by** (*fastforce simp add: lexn-conv assms(2)*)

next

case (*decided* $S\ L$)

have m : *map* ($\lambda l.$ if is-marked l then 2 else 1) (rev (*trail* S))

@ replicate (*card vars* − *length* (*trail* S)) 3

= *map* ($\lambda l.$ if is-marked l then 2 else 1) (rev (*trail* S)) @ 3

replicate (*card vars* − Suc (*length* (*trail* S))) 3

using *decided.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*

then show ?*case*

using *decided.prem[s]* **unfolding** *dpll_W-mes-def* **by** (*force simp add: lexn-conv assms(2)*)

next

```

case (backtrack S M' L M D)
have L: 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 dpllW-card-decrease':
  assumes dpll: dpllW S S'
  and atm-incl: atm-of ' lits-of (trail S) ⊆ atms-of-msu (clauses S)
  and no-dup: no-dup (trail S)
  shows (dpllW-mes (trail S') (card (atms-of-msu (clauses S'))),
        dpllW-mes (trail S) (card (atms-of-msu (clauses S)))) ∈ lex {(a, b). a < b}
proof -
  have finite (atms-of-msu (clauses S)) unfolding atms-of-ms-def by auto
  then have 1: length (trail S) ≤ card (atms-of-msu (clauses S))
    using distinctcard-atm-of-lit-of-eq-length[OF no-dup] atm-incl card-mono by metis

```

moreover

```

  have no-dup': no-dup (trail S') using dpll dpllW-distinct-inv no-dup by blast
  have SS': clauses S' = clauses S using dpll by (auto dest!: dpllW-same-clauses)
  have atm-incl': atm-of ' lits-of (trail S') ⊆ atms-of-msu (clauses S')
    using atm-incl dpll dpllW-vars-in-snd-inv[OF dpll] by force
  have finite (atms-of-msu (clauses S'))
    unfolding atms-of-ms-def by auto
  then have 2: length (trail S') ≤ card (atms-of-msu (clauses S'))
    using distinctcard-atm-of-lit-of-eq-length[OF no-dup'] atm-incl' card-mono SS' by metis

```

```

ultimately have (dpllW-mes (trail S') (card (atms-of-msu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-msu (clauses S))))
  ∈ lexn {(a, b). a < b} (card (atms-of-msu (clauses S)))
  using dpllW-card-decrease[OF assms(1), of atms-of-msu (clauses S)] by blast
then have (dpllW-mes (trail S') (card (atms-of-msu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-msu (clauses S)))) ∈ lex {(a, b). a < b}
  unfolding lex-def by auto
then show (dpllW-mes (trail S') (card (atms-of-msu (clauses S'))),
  dpllW-mes (trail S) (card (atms-of-msu (clauses S)))) ∈ lex {(a, b). a < b}
  using dpllW-same-clauses[OF assms(1)] by auto
qed

```

```

lemma wf-lexn: wf (lexn {(a, b). (a::nat) < b} (card (atms-of-msu (clauses S))))

```

proof -

```

  have m: {(a, b). a < b} = measure id by auto
  show ?thesis apply (rule wf-lexn) unfolding m by auto

```

qed

```

lemma dpllW-wf:

```

```

  wf {(S', S). dpllW-all-inv S ∧ dpllW S S'}
  apply (rule wf-wf-if-measure'[OF wf-lex-less, of - -
    λS. dpllW-mes (trail S) (card (atms-of-msu (clauses S)))]])
  using dpllW-card-decrease' by fast

```

lemma *dpll_W-trancpl-star-commute*:
 $\{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+ = \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{trancpl } \text{dpll}_W S S'\}$
 (is ?A = ?B)

proof

```

{ fix S S'
  assume (S, S') ∈ ?A
  then have (S, S') ∈ ?B
    by (induct rule: trancpl.induct, auto)
}
then show ?A ⊆ ?B by blast
{ fix S S'
  assume (S, S') ∈ ?B
  then have dpllW++ S' S and dpllW-all-inv S' by auto
  then have (S, S') ∈ ?A
    proof (induct rule: trancpl.induct)
      case r-into-trancpl
      then show ?case by (simp-all add: r-into-trancpl')
    next
      case (trancpl-into-trancpl S S' S'')
      then have (S', S) ∈ {a. case a of (S', S) ⇒ dpllW-all-inv S ∧ dpllW S S'}+ by blast
      moreover have dpllW-all-inv S'
        using rtrancpl-dpllW-all-inv[OF trancpl-into-rtrancpl[OF trancpl-into-trancpl.hyps(1)]]
        trancpl-into-trancpl.prem by auto
      ultimately have (S'', S') ∈ {(pa, p). dpllW-all-inv p ∧ dpllW p pa}+
        using ⟨dpllW-all-inv S'⟩ trancpl-into-trancpl.hyps(3) by blast
      then show ?case
        using ⟨(S', S) ∈ {a. case a of (S', S) ⇒ dpllW-all-inv S ∧ dpllW S S'}+⟩ by auto
    qed
  }
then show ?B ⊆ ?A by blast
qed

```

lemma *dpll_W-wf-trancpl*: wf {(S', S). dpll_W-all-inv S ∧ dpll_W⁺⁺ S S'}
 unfolding dpll_W-trancpl-star-commute[symmetric] by (simp add: dpll_W-wf wf-trancpl)

lemma *dpll_W-wf-plus*:
 shows wf {(S', (□, N)) | S'. dpll_W⁺⁺ (□, N) S'} (is wf ?P)
 apply (rule wf-subset[OF dpll_W-wf-trancpl, of ?P])
 using assms unfolding dpll_W-all-inv-def by auto

16.4 Final States

lemma *dpll_W-no-more-step-is-a-conclusive-state*:

assumes $\forall S'. \neg \text{dpll}_W S S'$
 shows conclusive-dpll_W-state S

proof —

```

have vars: ∀ s ∈ atms-of-msu (clauses S). s ∈ atm-of ' lits-of (trail S)
proof (rule ccontr)
  assume ¬ (∀ s ∈ atms-of-msu (clauses S). s ∈ atm-of ' lits-of (trail S))
  then obtain L where
    L-in-atms: L ∈ atms-of-msu (clauses S) and
    L-notin-trail: L ∉ atm-of ' lits-of (trail S) by metis
  obtain L' where L': atm-of L' = L by (meson literal.sel(2))
  then have undefined-lit (trail S) L'
    unfolding Marked-Propagated-in-iff-in-lits-of by (metis L-notin-trail atm-of-uminus imageI)
  then show False using dpllW.decided assms(1) L-in-atms L' by blast

```

```

qed
show ?thesis
proof (rule ccontr)
  assume not-final:  $\neg$  ?thesis
  then have
     $\neg$  trail S  $\models_{asm}$  clauses S and
     $(\exists L \in set (trail S). is\_marked L) \vee (\forall C \in \# clauses S. \neg trail S \models_{as} CNot C)$ 
    unfolding conclusive-dpllW-state-def by auto
  moreover {
    assume  $\exists L \in set (trail S). is\_marked L$ 
    then obtain L M' M where L: backtrack-split (trail S) = (M', L # M)
      using backtrack-split-some-is-marked-then-snd-has-hd by blast
    obtain D where D  $\in \# clauses S$  and  $\neg trail S \models_a D$ 
      using  $\langle \neg trail S \models_{asm} clauses S \rangle$  unfolding true-annots-def by auto
    then have  $\forall s \in atms\_of\_ms \{D\}. s \in atm\_of \text{ ' lits-of } (trail S)$ 
      using vars unfolding atms-of-ms-def by auto
    then have trail S  $\models_{as} CNot D$ 
      using all-variables-defined-not-imply-cnot[of D]  $\langle \neg trail S \models_a D \rangle$  by auto
    moreover have is-marked L
      using L by (metis backtrack-split-snd-hd-marked list.distinct(1) list.sel(1) snd-conv)
    ultimately have False
      using assms(1) dpllW.backtrack L  $\langle D \in \# clauses S \rangle \langle trail S \models_{as} CNot D \rangle$  by blast
  }
  moreover {
    assume tr:  $\forall C \in \# clauses S. \neg trail S \models_{as} CNot C$ 
    obtain C where C-in-cls: C  $\in \# clauses S$  and trC:  $\neg trail S \models_a C$ 
      using  $\langle \neg trail S \models_{asm} clauses S \rangle$  unfolding true-annots-def by auto
    have  $\forall s \in atms\_of\_ms \{C\}. s \in atm\_of \text{ ' lits-of } (trail S)$ 
      using vars  $\langle C \in \# clauses S \rangle$  unfolding atms-of-ms-def by auto
    then have trail S  $\models_{as} CNot C$ 
      by (meson C-in-cls tr trC all-variables-defined-not-imply-cnot)
    then have False using tr C-in-cls by auto
  }
  ultimately show False by blast
qed
qed

lemma dpllW-conclusive-state-correct:
  assumes dpllW** ([], N) (M, N) and conclusive-dpllW-state (M, N)
  shows M  $\models_{asm} N \longleftrightarrow$  satisfiable (set-mset N) (is ?A  $\longleftrightarrow$  ?B)
proof
  let ?M' = lits-of M
  assume ?A
  then have ?M'  $\models_{sm} N$  by (simp add: true-annots-true-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:  $\forall L \in set M. \neg is\_marked L \exists C \in \# N. M \models_{as} CNot C$ 
      using n assms(2) unfolding conclusive-dpllW-state-def by auto
    moreover obtain D where DN: D  $\in \# N$  and MD: M  $\models_{as} CNot D$  using no-mark by auto

```



```

ultimately have unsatisfiable (set-mset N)
  using only-propagated-vars-unsat rtrancpl-dpllW-all-inv[OF assms(1)]
  unfolding dpllW-all-inv-def by force
then show False using ⟨?B⟩ by blast
qed
qed

```

16.5 Link with NOT's DPLL

interpretation $dpll_{W-NOT}$: *dpll-with-backtrack* .

```

lemma state-eqNOT-iff-eq[iff, simp]: dpllW-NOT.state-eqNOT S T  $\longleftrightarrow$  S = T
  unfolding dpllW-NOT.state-eqNOT-def by (cases S, cases T) auto

```

```

declare dpllW-NOT.state-simpNOT[simp del]

```

```

lemma dpllW-dpllW-bj:
  assumes inv: dpllW-all-inv S and dpll: dpllW S T
  shows dpllW-NOT.dpll-bj S T
  using dpll inv
  apply (induction rule: dpllW.induct)
    using dpllW-NOT.dpll-bj.simps apply fastforce
    using dpllW-NOT.bj-decideNOT apply fastforce
  apply (frule dpllW-NOT.backtrack.intros[of - - - -], simp-all)
  apply (rule dpllW-NOT.dpll-bj.bj-backjump)
  apply (rule dpllW-NOT.backtrack-is-backjump'',
    simp-all add: dpllW-all-inv-def)
done

```

```

lemma dpllW-bj-dpll:
  assumes inv: dpllW-all-inv S and dpll: dpllW-NOT.dpll-bj S T
  shows dpllW S T
  using dpll
  apply (induction rule: dpllW-NOT.dpll-bj.induct)
    apply (elim dpllW-NOT.decideE, cases S)
    using decided apply fastforce
    apply (elim dpllW-NOT.propagateE, cases S)
    using dpllW.simps apply fastforce
    apply (elim dpllW-NOT.backjumpE, cases S)
  by (simp add: dpllW.simps dpll-with-backtrack.backtrack.simps)

```

```

lemma rtrancpl-dpllW-rtrancpl-dpllW-NOT:
  assumes dpllW** S T and dpllW-all-inv S
  shows dpllW-NOT.dpll-bj** S T
  using assms apply (induction)
  apply simp
  by (auto intro: rtrancpl-dpllW-all-inv dpllW-dpllW-bj rtrancpl.rtrancpl-into-rtrancpl)

```

```

lemma rtrancpl-dpll-rtrancpl-dpllW:
  assumes dpllW-NOT.dpll-bj** S T and dpllW-all-inv S
  shows dpllW** S T
  using assms apply (induction)
  apply simp
  by (auto intro: dpllW-bj-dpll rtrancpl.rtrancpl-into-rtrancpl rtrancpl-dpllW-all-inv)

```

```

lemma dpll-conclusive-state-correctness:

```

```

assumes  $dpll_W\text{-}NOT.dpll\text{-}bj^{**} ([], N) (M, N)$  and  $conclusive\text{-}dpll_W\text{-}state (M, N)$ 
shows  $M \models_{asm} N \longleftrightarrow satisfiable (set\text{-}mset N)$ 
proof -
  have  $dpll_W\text{-}all\text{-}inv ([], N)$ 
    unfolding  $dpll_W\text{-}all\text{-}inv\text{-}def$  by auto
  show ?thesis
    apply (rule  $dpll_W\text{-}conclusive\text{-}state\text{-}correct$ )
      apply (simp add:  $\langle dpll_W\text{-}all\text{-}inv ([], N) \rangle assms(1) rtranclp\text{-}dpll\text{-}rtranclp\text{-}dpll_W$ )
      using  $assms(2)$  by simp
qed

end
theory CDCL-W-Level
imports Partial-Annotated-Clausal-Logic
begin

```

16.5.1 Level of literals and clauses

Getting the level of a variable, implies that the list has to be reversed. Here is the funtion after reversing.

```

fun  $get\text{-}rev\text{-}level :: 'v\ literal \Rightarrow nat \Rightarrow ('v, nat, 'a) marked\text{-}lits \Rightarrow nat$  where
 $get\text{-}rev\text{-}level - [] = 0$  |
 $get\text{-}rev\text{-}level L n (Marked\ l\ level\ \# Ls) =$ 
   $(if\ atm\text{-}of\ l = atm\text{-}of\ L\ then\ level\ else\ get\text{-}rev\text{-}level\ L\ level\ Ls) |$ 
 $get\text{-}rev\text{-}level L n (Propagated\ l - \# Ls) =$ 
   $(if\ atm\text{-}of\ l = atm\text{-}of\ L\ then\ n\ else\ get\text{-}rev\text{-}level\ L\ n\ Ls)$ 

```

abbreviation $get\text{-}level L M \equiv get\text{-}rev\text{-}level L 0 (rev\ M)$

lemma $get\text{-}rev\text{-}level\text{-}uminus[simp]$: $get\text{-}rev\text{-}level (-L) n M = get\text{-}rev\text{-}level L n M$
by (*induct* M *arbitrary*: n *rule*: $get\text{-}rev\text{-}level.induct$) *auto*

lemma $atm\text{-}of\text{-}notin\text{-}get\text{-}rev\text{-}level\text{-}eq\text{-}0[simp]$:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ ' lits\text{-}of\ M$
shows $get\text{-}rev\text{-}level L n M = 0$
using $assms$ **apply** (*induct* M *arbitrary*: n , *simp*)
by (*case-tac* a) *auto*

lemma $get\text{-}rev\text{-}level\text{-}ge\text{-}0\text{-}atm\text{-}of\text{-}in$:
assumes $get\text{-}rev\text{-}level L n M > n$
shows $atm\text{-}of\ L \in atm\text{-}of\ ' lits\text{-}of\ M$
using $assms$ **apply** (*induct* M *arbitrary*: n , *simp*)
by (*case-tac* a) *fastforce*+

In $get\text{-}rev\text{-}level$ (resp. $get\text{-}level$), the beginning (resp. the end) can be skipped if the literal is not in the beginning (resp. the end).

lemma $get\text{-}rev\text{-}level\text{-}skip[simp]$:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ ' lits\text{-}of\ M$
shows $get\text{-}rev\text{-}level L n (M @ Marked\ K\ i \# M') = get\text{-}rev\text{-}level L i (Marked\ K\ i \# M')$
using $assms$ **apply** (*induct* M *arbitrary*: $n\ i$, *simp*)
by (*case-tac* a) *auto*

lemma $get\text{-}rev\text{-}level\text{-}notin\text{-}end[simp]$:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ ' lits\text{-}of\ M'$
shows $get\text{-}rev\text{-}level L n (M @ M') = get\text{-}rev\text{-}level L n M$

using *assms* **apply** (*induct M arbitrary: n, simp*)
by (*case-tac a*) *auto*

If the literal is at the beginning, then the end can be skipped

lemma *get-rev-level-skip-end*[*simp*]:
assumes *atm-of L* \in *atm-of* ' *lits-of M*
shows *get-rev-level L n* (*M* @ *M'*) = *get-rev-level L n M*
using *assms* **apply** (*induct M arbitrary: n, simp*)
by (*case-tac a*) *auto*

lemma *get-level-skip-beginning*:
assumes *atm-of L'* \neq *atm-of* (*lit-of K*)
shows *get-level L'* (*K* # *M*) = *get-level L' M*
using *assms* **by** *auto*

lemma *get-level-skip-beginning-not-marked-rev*:
assumes *atm-of L* \notin *atm-of* ' *lit-of* ' (*set S*)
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows *get-level L* (*M* @ *rev S*) = *get-level L M*
using *assms* **by** (*induction S rule: marked-lit-list-induct*) *auto*

lemma *get-level-skip-beginning-not-marked*[*simp*]:
assumes *atm-of L* \notin *atm-of* ' *lit-of* ' (*set S*)
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows *get-level L* (*M* @ *S*) = *get-level L M*
using *get-level-skip-beginning-not-marked-rev*[*of L rev S M*] *assms* **by** *auto*

lemma *get-rev-level-skip-beginning-not-marked*[*simp*]:
assumes *atm-of L* \notin *atm-of* ' *lit-of* ' (*set S*)
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows *get-rev-level L 0* (*rev S* @ *rev M*) = *get-level L M*
using *get-level-skip-beginning-not-marked-rev*[*of L rev S M*] *assms* **by** *auto*

lemma *get-level-skip-in-all-not-marked*:
fixes *M* :: ('a, nat, 'b) *marked-lit list* **and** *L* :: 'a *literal*
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
and *atm-of L* \in *atm-of* ' *lit-of* ' (*set M*)
shows *get-rev-level L n M* = *n*

proof –
show ?*thesis*
using *assms* **by** (*induction M rule: marked-lit-list-induct*) *auto*
qed

lemma *get-level-skip-all-not-marked*[*simp*]:
fixes *M*
defines *M'* \equiv *rev M*
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows *get-level L M* = 0

proof –
have *M*: *M* = *rev M'*
unfolding *M'-def* **by** *auto*
show ?*thesis*
using *assms* **unfolding** *M* **by** (*induction M' rule: marked-lit-list-induct*) *auto*
qed

abbreviation $MMax\ M \equiv Max\ (set-mset\ M)$

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

definition $get-maximum-level :: 'a\ literal\ multiset \Rightarrow ('a, nat, 'b)\ marked-lit\ list \Rightarrow nat$
where

$get-maximum-level\ D\ M = MMax\ (\{\#0\#\} + image-mset\ (\lambda L. get-level\ L\ M)\ D)$

lemma $get-maximum-level-ge-get-level$:

$L \in \# D \implies get-maximum-level\ D\ M \geq get-level\ L\ M$

unfolding $get-maximum-level-def$ **by** $auto$

lemma $get-maximum-level-empty[simp]$:

$get-maximum-level\ \{\#\}\ M = 0$

unfolding $get-maximum-level-def$ **by** $auto$

lemma $get-maximum-level-exists-lit-of-max-level$:

$D \neq \{\#\} \implies \exists L \in \# D. get-level\ L\ M = get-maximum-level\ D\ M$

unfolding $get-maximum-level-def$

apply $(induct\ D)$

apply $simp$

by $(case-tac\ D = \{\#\})\ (auto\ simp\ add: max-def)$

lemma $get-maximum-level-empty-list[simp]$:

$get-maximum-level\ D\ [] = 0$

unfolding $get-maximum-level-def$ **by** $(simp\ add: image-constant-conv)$

lemma $get-maximum-level-single[simp]$:

$get-maximum-level\ \{\#L\#\}\ M = get-level\ L\ M$

unfolding $get-maximum-level-def$ **by** $simp$

lemma $get-maximum-level-plus$:

$get-maximum-level\ (D + D')\ M = max\ (get-maximum-level\ D\ M)\ (get-maximum-level\ D'\ M)$

by $(induct\ D)\ (auto\ simp\ add: get-maximum-level-def)$

lemma $get-maximum-level-exists-lit$:

assumes $n: n > 0$

and $max: get-maximum-level\ D\ M = n$

shows $\exists L \in \# D. get-level\ L\ M = n$

proof $-$

have $f: finite\ (insert\ 0\ ((\lambda L. get-level\ L\ M)\ ' set-mset\ D))$ **by** $auto$

hence $n \in ((\lambda L. get-level\ L\ M)\ ' set-mset\ D)$

using $n\ max\ Max-in[OF\ f]$ **unfolding** $get-maximum-level-def$ **by** $simp$

thus $\exists L \in \# D. get-level\ L\ M = n$ **by** $auto$

qed

lemma $get-maximum-level-skip-first[simp]$:

assumes $atm-of\ L \notin atms-of\ D$

shows $get-maximum-level\ D\ (Propagated\ L\ C\ \# M) = get-maximum-level\ D\ M$

using $assms$ **unfolding** $get-maximum-level-def\ atms-of-def$

$atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set$

by $(smt\ atm-of-in-atm-of-set-in-uminus\ get-level-skip-beginning\ image-iff\ marked-lit.sel(2)\ multiset.map-cong0)$

lemma *get-maximum-level-skip-beginning*:
assumes DH : $\text{atms-of } D \subseteq \text{atm-of 'lits-of } H$
shows $\text{get-maximum-level } D (c @ \text{Marked } Kh \ i \ \# \ H) = \text{get-maximum-level } D \ H$
proof –
have $(\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } H @ \text{Marked } Kh \ i \ \# \ \text{rev } c)) \text{ 'set-mset } D$
 $= (\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } H)) \text{ 'set-mset } D$
using DH **unfolding** *atms-of-def*
by (*metis* (*no-types*, *lifting*) *get-rev-level-skip-end image-cong image-subset-iff lits-of-rev*) +
thus ?thesis **using** DH **unfolding** *get-maximum-level-def* **by** *auto*
qed

lemma *get-maximum-level-D-single-propagated*:
 $\text{get-maximum-level } D [\text{Propagated } x21 \ x22] = 0$
proof –
have A : $\text{insert } 0 \ ((\lambda L. \ 0) \text{ 'set-mset } D \cap \{L. \text{atm-of } x21 = \text{atm-of } L\})$
 $\cup (\lambda L. \ 0) \text{ 'set-mset } D \cap \{L. \text{atm-of } x21 \neq \text{atm-of } L\}) = \{0\}$
by *auto*
show ?thesis **unfolding** *get-maximum-level-def* **by** (*simp add: A*)
qed

lemma *get-maximum-level-skip-notin*:
assumes D : $\forall L \in \#D. \text{atm-of } L \in \text{atm-of 'lits-of } M$
shows $\text{get-maximum-level } D \ M = \text{get-maximum-level } D (\text{Propagated } x21 \ x22 \ \# \ M)$
proof –
have A : $(\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } M @ [\text{Propagated } x21 \ x22])) \text{ 'set-mset } D$
 $= (\lambda L. \text{get-rev-level } L \ 0 \ (\text{rev } M)) \text{ 'set-mset } D$
using D **by** (*auto intro!: image-cong simp add: lits-of-def*)
show ?thesis **unfolding** *get-maximum-level-def* **by** (*auto simp add: A*)
qed

lemma *get-maximum-level-skip-un-marked-not-present*:
assumes $\forall L \in \#D. \text{atm-of } L \in \text{atm-of 'lits-of } aa$ **and**
 $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\text{get-maximum-level } D \ aa = \text{get-maximum-level } D (M @ aa)$
using *assms* **apply** (*induction M*)
apply *simp*
by (*case-tac a*) (*auto intro!: get-maximum-level-skip-notin[of D - @ aa] simp add: image-Un*)

fun *get-maximum-possible-level:: ('b, nat, 'c) marked-lit list \Rightarrow nat* **where**
 $\text{get-maximum-possible-level } [] = 0 \mid$
 $\text{get-maximum-possible-level } (\text{Marked } K \ i \ \# \ l) = \max i (\text{get-maximum-possible-level } l) \mid$
 $\text{get-maximum-possible-level } (\text{Propagated } - \ - \ \# \ l) = \text{get-maximum-possible-level } l$

lemma *get-maximum-possible-level-append[simp]*:
 $\text{get-maximum-possible-level } (M @ M')$
 $= \max (\text{get-maximum-possible-level } M) (\text{get-maximum-possible-level } M')$
apply (*induct M, simp*) **by** (*case-tac a, auto*)

lemma *get-maximum-possible-level-rev[simp]*:
 $\text{get-maximum-possible-level } (\text{rev } M) = \text{get-maximum-possible-level } M$
apply (*induct M, simp*) **by** (*case-tac a, auto*)

lemma *get-maximum-possible-level-ge-get-rev-level*:
 $\max (\text{get-maximum-possible-level } M) \ i \geq \text{get-rev-level } L \ i \ M$
apply (*induct M arbitrary: i*)

```

  apply simp
by (case-tac a) (auto simp add: le-max-iff-disj)

lemma get-maximum-possible-level-ge-get-level[simp]:
  get-maximum-possible-level M ≥ get-level L M
  using get-maximum-possible-level-ge-get-rev-level[of - 0 rev -] by auto

lemma get-maximum-possible-level-ge-get-maximum-level[simp]:
  get-maximum-possible-level M ≥ get-maximum-level D M
  using get-maximum-level-exists-lit-of-max-level unfolding Bex-mset-def
  by (metis get-maximum-level-empty get-maximum-possible-level-ge-get-level le0)

fun get-all-mark-of-propagated where
  get-all-mark-of-propagated [] = [] |
  get-all-mark-of-propagated (Marked - - # L) = get-all-mark-of-propagated L |
  get-all-mark-of-propagated (Propagated - mark # L) = mark # get-all-mark-of-propagated L

lemma get-all-mark-of-propagated-append[simp]: get-all-mark-of-propagated (A @ B) = get-all-mark-of-propagated
A @ get-all-mark-of-propagated B
  apply (induct A, simp)
  by (case-tac a) auto



### 16.5.2 Properties about the levels



fun get-all-levels-of-marked :: ('b, 'a, 'c) marked-lit list ⇒ 'a list where
  get-all-levels-of-marked [] = [] |
  get-all-levels-of-marked (Marked l level # Ls) = level # get-all-levels-of-marked Ls |
  get-all-levels-of-marked (Propagated - - # Ls) = get-all-levels-of-marked Ls

lemma get-all-levels-of-marked-nil-iff-not-is-marked:
  get-all-levels-of-marked xs = [] ⟷ (∀ x ∈ set xs. ¬is-marked x)
  using assms by (induction xs rule: marked-lit-list-induct) auto

lemma get-all-levels-of-marked-cons:
  get-all-levels-of-marked (a # b) =
    (if is-marked a then [level-of a] else []) @ get-all-levels-of-marked b
  by (case-tac a) simp-all

lemma get-all-levels-of-marked-append[simp]:
  get-all-levels-of-marked (a @ b) = get-all-levels-of-marked a @ get-all-levels-of-marked b
  by (induct a) (simp-all add: get-all-levels-of-marked-cons)

lemma in-get-all-levels-of-marked-iff-decomp:
  i ∈ set (get-all-levels-of-marked M) ⟷ (∃ c K c'. M = c @ Marked K i # c') (is ?A ⟷ ?B)
proof
  assume ?B
  thus ?A by auto
next
  assume ?A
  thus ?B
  apply (induction M rule: marked-lit-list-induct)
  apply auto[]
  apply (metis append-Cons append-Nil get-all-levels-of-marked.simps(2) set-ConsD)
  by (metis append-Cons get-all-levels-of-marked.simps(3))
qed

```

lemma *get-rev-level-less-max-get-all-levels-of-marked*:

get-rev-level $L \ n \ M \leq \text{Max} \ (\text{set} \ (n \ \# \ \text{get-all-levels-of-marked} \ M))$
by (*induct* M *arbitrary*: n *rule*: *get-all-levels-of-marked.induct*)
(simp-all add: max.coboundedI2)

lemma *get-rev-level-ge-min-get-all-levels-of-marked*:

assumes *atm-of* $L \in \text{atm-of} \ ' \ \text{lits-of} \ M$
shows *get-rev-level* $L \ n \ M \geq \text{Min} \ (\text{set} \ (n \ \# \ \text{get-all-levels-of-marked} \ M))$
using *assms* **by** (*induct* M *arbitrary*: n *rule*: *get-all-levels-of-marked.induct*)
(auto simp add: min-le-iff-disj)

lemma *get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked[simp]*:

get-all-levels-of-marked (*rev* M) = *rev* (*get-all-levels-of-marked* M)
by (*induct* M *rule*: *get-all-levels-of-marked.induct*)
(simp-all add: max.coboundedI2)

lemma *get-maximum-possible-level-max-get-all-levels-of-marked*:

get-maximum-possible-level $M = \text{Max} \ (\text{insert} \ 0 \ (\text{set} \ (\text{get-all-levels-of-marked} \ M)))$
apply (*induct* M , *simp*)
by (*case-tac* a) (*case-tac* $\text{set} \ (\text{get-all-levels-of-marked} \ M) = \{\}, \text{auto}$)

lemma *get-rev-level-in-levels-of-marked*:

get-rev-level $L \ n \ M \in \{0, n\} \cup \text{set} \ (\text{get-all-levels-of-marked} \ M)$
apply (*induction* M *arbitrary*: n)
apply *auto*[1]
by (*case-tac* a)
(force simp add: atm-of-eq-atm-of)+

lemma *get-rev-level-in-atms-in-levels-of-marked*:

atm-of $L \in \text{atm-of} \ ' \ (\text{lits-of} \ M) \implies \text{get-rev-level} \ L \ n \ M \in \{n\} \cup \text{set} \ (\text{get-all-levels-of-marked} \ M)$
apply (*induction* M *arbitrary*: n , *simp*)
by (*case-tac* a)
(auto simp add: atm-of-eq-atm-of)

lemma *get-all-levels-of-marked-no-marked*:

$(\forall l \in \text{set} \ Ls. \neg \text{is-marked} \ l) \iff \text{get-all-levels-of-marked} \ Ls = []$
by (*induction* Ls) (*auto simp add: get-all-levels-of-marked-cons*)

lemma *get-level-in-levels-of-marked*:

get-level $L \ M \in \{0\} \cup \text{set} \ (\text{get-all-levels-of-marked} \ M)$
using *get-rev-level-in-levels-of-marked[of* $L \ 0 \ \text{rev} \ M]$ **by** *auto*

The zero is here to avoid empty-list issues with *last*:

lemma *get-level-get-rev-level-get-all-levels-of-marked*:

assumes *atm-of* $L \notin \text{atm-of} \ ' \ (\text{lits-of} \ M)$
shows *get-level* $L \ (K \ @ \ M) = \text{get-rev-level} \ L \ (\text{last} \ (0 \ \# \ \text{get-all-levels-of-marked} \ (\text{rev} \ M)))$
(rev K)

using *assms*

proof (*induct* M *arbitrary*: K)

case *Nil*

thus *?case* **by** *auto*

next

case (*Cons* $a \ M$)

hence $H: \bigwedge K. \text{get-level} \ L \ (K \ @ \ M)$

```

    = get-rev-level L (last (0 # get-all-levels-of-marked (rev M))) (rev K)
  by auto
  have get-level L ((K @ [a])@ M)
    = get-rev-level L (last (0 # get-all-levels-of-marked (rev M))) (a # rev K)
    using H[of K @ [a]] by simp
  thus ?case using Cons(2) by (case-tac a) auto
qed

```

lemma *get-rev-level-can-skip-correctly-ordered:*

```

  assumes no-dup M
  and atm-of L  $\notin$  atm-of ' (lits-of M)
  and get-all-levels-of-marked M = rev [Suc 0.. $\leq$ Suc (length (get-all-levels-of-marked M))]
  shows get-rev-level L 0 (rev M @ K) = get-rev-level L (length (get-all-levels-of-marked M)) K
  using assms
proof (induct M arbitrary: K)
  case Nil
  thus ?case by simp
next
  case (Cons a M K)
  show ?case
  proof (case-tac a)
    fix L' i
    assume a: a = Marked L' i
    have i: i = Suc (length (get-all-levels-of-marked M))
    and get-all-levels-of-marked M = rev [Suc 0.. $\leq$ Suc (length (get-all-levels-of-marked M))]
      using Cons.prem(3) unfolding a by auto
    hence get-rev-level L 0 (rev M @ (a # K))
      = get-rev-level L (length (get-all-levels-of-marked M)) (a # K)
      using Cons.hyps Cons.prem by auto
    thus ?case using Cons.prem(2) unfolding a i by auto
  next
    fix L' D
    assume a: a = Propagated L' D
    have get-all-levels-of-marked M = rev [Suc 0.. $\leq$ Suc (length (get-all-levels-of-marked M))]
      using Cons.prem(3) unfolding a by auto
    hence get-rev-level L 0 (rev M @ (a # K))
      = get-rev-level L (length (get-all-levels-of-marked M)) (a # K)
      using Cons by auto
    thus ?case using Cons.prem(2) unfolding a by auto
  qed
qed

```

lemma *get-level-skip-beginning-hd-get-all-levels-of-marked:*

```

  assumes atm-of L  $\notin$  atm-of ' lits-of S
  and get-all-levels-of-marked S  $\neq$  []
  shows get-level L (M @ S) = get-rev-level L (hd (get-all-levels-of-marked S)) (rev M)
  using assms
proof (induction S arbitrary: M rule: marked-lit-list-induct)
  case nil
  thus ?case by (auto simp add: lits-of-def)
next
  case (marked K m)
  note notin = this(2)
  thus ?case by (auto simp add: lits-of-def)
next
  case (proped L l)
  note IH = this(1) and L = this(2) and neq = this(3)

```


show *?case* **using** *IH[of M@[Propagated L l]] L neq* **by** (*auto simp add: atm-of-eq-atm-of*)
qed

end

theory *CDCL-W*

imports *Partial-Annotated-Clausal-Logic List-More CDCL-W-Level Wellfounded-More*

begin

declare *set-mset-minus-replicate-mset[simp]*

lemma *Bex-set-set-Bex-set[iff]*: $(\exists x \in \text{set-mset } C. P) \longleftrightarrow (\exists x \in \#C. P)$
by *auto*

17 Weidenbach's CDCL

sledgehammer-params[*verbose, e spass cvc4 z3 verit*]

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

datatype *'a conflicting-clause* = *C-True* | *C-Clause 'a*

17.1 The State

locale *state_W* =

fixes

trail :: *'st* \Rightarrow (*'v*, *nat*, *'v clause*) *marked-lits* **and**

init-clss :: *'st* \Rightarrow *'v clauses* **and**

learned-clss :: *'st* \Rightarrow *'v clauses* **and**

backtrack-lvl :: *'st* \Rightarrow *nat* **and**

conflicting :: *'st* \Rightarrow *'v clause conflicting-clause* **and**

cons-trail :: (*'v*, *nat*, *'v clause*) *marked-lit* \Rightarrow *'st* \Rightarrow *'st* **and**

tl-trail :: *'st* \Rightarrow *'st* **and**

add-init-cls :: *'v clause* \Rightarrow *'st* \Rightarrow *'st* **and**

add-learned-cls :: *'v clause* \Rightarrow *'st* \Rightarrow *'st* **and**

remove-cls :: *'v clause* \Rightarrow *'st* \Rightarrow *'st* **and**

update-backtrack-lvl :: *nat* \Rightarrow *'st* \Rightarrow *'st* **and**

update-conflicting :: *'v clause conflicting-clause* \Rightarrow *'st* \Rightarrow *'st* **and**

init-state :: *'v clauses* \Rightarrow *'st* **and**

restart-state :: *'st* \Rightarrow *'st*

assumes

trail-cons-trail[simp]:

$\bigwedge L \text{ st. } \text{undefined-lit } (\text{trail st}) (\text{lit-of } L) \Longrightarrow \text{trail } (\text{cons-trail } L \text{ st}) = L \# \text{trail st}$ **and**

trail-tl-trail[simp]: $\bigwedge \text{st. trail } (\text{tl-trail st}) = \text{tl } (\text{trail st})$ **and**

trail-add-init-cls[simp]:

$\bigwedge \text{st } C. \text{no-dup } (\text{trail st}) \Longrightarrow \text{trail } (\text{add-init-cls } C \text{ st}) = \text{trail st}$ **and**

trail-add-learned-cls[simp]:

$\bigwedge C \text{ st. no-dup } (\text{trail st}) \Longrightarrow \text{trail } (\text{add-learned-cls } C \text{ st}) = \text{trail st}$ **and**

trail-remove-cls[simp]:

$\bigwedge C \text{ st. trail } (\text{remove-cls } C \text{ st}) = \text{trail st}$ **and**

trail-update-backtrack-lvl[simp]: $\bigwedge C \text{ st. trail } (\text{update-backtrack-lvl } C \text{ st}) = \text{trail st}$ **and**

trail-update-conflicting[simp]: $\bigwedge C \text{ st. trail } (\text{update-conflicting } C \text{ st}) = \text{trail st}$ **and**

init-clss-cons-trail[simp]:

$\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies init\text{-clss } (cons\text{-trail } M \text{ st}) = init\text{-clss } st \text{ and }$
 $init\text{-clss-tl-trail}[simp]:$
 $\bigwedge st. init\text{-clss } (tl\text{-trail } st) = init\text{-clss } st \text{ and }$
 $init\text{-clss-add-init-cls}[simp]:$
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies init\text{-clss } (add\text{-init-cls } C \text{ st}) = \{\#C\# \} + init\text{-clss } st \text{ and }$
 $init\text{-clss-add-learned-cls}[simp]:$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies init\text{-clss } (add\text{-learned-cls } C \text{ st}) = init\text{-clss } st \text{ and }$
 $init\text{-clss-remove-cls}[simp]:$
 $\bigwedge C \text{ st. init-clss } (remove\text{-cls } C \text{ st}) = remove\text{-mset } C (init\text{-clss } st) \text{ and }$
 $init\text{-clss-update-backtrack-lvl}[simp]:$
 $\bigwedge st \ C. init\text{-clss } (update\text{-backtrack-lvl } C \text{ st}) = init\text{-clss } st \text{ and }$
 $init\text{-clss-update-conflicting}[simp]:$
 $\bigwedge C \text{ st. init-clss } (update\text{-conflicting } C \text{ st}) = init\text{-clss } st \text{ and }$

$learned\text{-clss-cons-trail}[simp]:$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $learned\text{-clss } (cons\text{-trail } M \text{ st}) = learned\text{-clss } st \text{ and }$
 $learned\text{-clss-tl-trail}[simp]:$
 $\bigwedge st. learned\text{-clss } (tl\text{-trail } st) = learned\text{-clss } st \text{ and }$
 $learned\text{-clss-add-init-cls}[simp]:$
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies learned\text{-clss } (add\text{-init-cls } C \text{ st}) = learned\text{-clss } st \text{ and }$
 $learned\text{-clss-add-learned-cls}[simp]:$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies learned\text{-clss } (add\text{-learned-cls } C \text{ st}) = \{\#C\# \} + learned\text{-clss } st$
 and
 $learned\text{-clss-remove-cls}[simp]:$
 $\bigwedge C \text{ st. learned-clss } (remove\text{-cls } C \text{ st}) = remove\text{-mset } C (learned\text{-clss } st) \text{ and }$
 $learned\text{-clss-update-backtrack-lvl}[simp]:$
 $\bigwedge st \ C. learned\text{-clss } (update\text{-backtrack-lvl } C \text{ st}) = learned\text{-clss } st \text{ and }$
 $learned\text{-clss-update-conflicting}[simp]:$
 $\bigwedge C \text{ st. learned-clss } (update\text{-conflicting } C \text{ st}) = learned\text{-clss } st \text{ and }$

$backtrack\text{-lvl-cons-trail}[simp]:$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $backtrack\text{-lvl } (cons\text{-trail } M \text{ st}) = backtrack\text{-lvl } st \text{ and }$
 $backtrack\text{-lvl-tl-trail}[simp]:$
 $\bigwedge st. backtrack\text{-lvl } (tl\text{-trail } st) = backtrack\text{-lvl } st \text{ and }$
 $backtrack\text{-lvl-add-init-cls}[simp]:$
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies backtrack\text{-lvl } (add\text{-init-cls } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$
 $backtrack\text{-lvl-add-learned-cls}[simp]:$
 $\bigwedge C \text{ st. no-dup } (trail \text{ st}) \implies backtrack\text{-lvl } (add\text{-learned-cls } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$
 $backtrack\text{-lvl-remove-cls}[simp]:$
 $\bigwedge C \text{ st. backtrack-lvl } (remove\text{-cls } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$
 $backtrack\text{-lvl-update-backtrack-lvl}[simp]:$
 $\bigwedge st \ k. backtrack\text{-lvl } (update\text{-backtrack-lvl } k \text{ st}) = k \text{ and }$
 $backtrack\text{-lvl-update-conflicting}[simp]:$
 $\bigwedge C \text{ st. backtrack-lvl } (update\text{-conflicting } C \text{ st}) = backtrack\text{-lvl } st \text{ and }$

$conflicting\text{-cons-trail}[simp]:$
 $\bigwedge M \text{ st. undefined-lit } (trail \text{ st}) (lit\text{-of } M) \implies$
 $conflicting (cons\text{-trail } M \text{ st}) = conflicting \text{ st and }$
 $conflicting\text{-tl-trail}[simp]:$
 $\bigwedge st. conflicting (tl\text{-trail } st) = conflicting \text{ st and }$
 $conflicting\text{-add-init-cls}[simp]:$
 $\bigwedge st \ C. no\text{-dup } (trail \text{ st}) \implies conflicting (add\text{-init-cls } C \text{ st}) = conflicting \text{ st and }$
 $conflicting\text{-add-learned-cls}[simp]:$

$\bigwedge C \text{ st. no-dup } (\text{trail } st) \implies \text{conflicting } (\text{add-learned-cls } C \text{ st}) = \text{conflicting } st \text{ and}$
 $\text{conflicting-remove-cls}[simp]:$

$\bigwedge C \text{ st. conflicting } (\text{remove-cls } C \text{ st}) = \text{conflicting } st \text{ and}$
 $\text{conflicting-update-backtrack-lvl}[simp]:$

$\bigwedge st \ C. \text{conflicting } (\text{update-backtrack-lvl } C \text{ st}) = \text{conflicting } st \text{ and}$
 $\text{conflicting-update-conflicting}[simp]:$

$\bigwedge C \text{ st. conflicting } (\text{update-conflicting } C \text{ st}) = C \text{ and}$

$\text{init-state-trail}[simp]: \bigwedge N. \text{trail } (\text{init-state } N) = [] \text{ and}$

$\text{init-state-clss}[simp]: \bigwedge N. \text{init-clss } (\text{init-state } N) = N \text{ and}$

$\text{init-state-learned-clss}[simp]: \bigwedge N. \text{learned-clss } (\text{init-state } N) = \{\#\} \text{ and}$

$\text{init-state-backtrack-lvl}[simp]: \bigwedge N. \text{backtrack-lvl } (\text{init-state } N) = 0 \text{ and}$

$\text{init-state-conflicting}[simp]: \bigwedge N. \text{conflicting } (\text{init-state } N) = C\text{-True and}$

$\text{trail-restart-state}[simp]: \text{trail } (\text{restart-state } S) = [] \text{ and}$

$\text{init-clss-restart-state}[simp]: \text{init-clss } (\text{restart-state } S) = \text{init-clss } S \text{ and}$

$\text{learned-clss-restart-state}[intro]: \text{learned-clss } (\text{restart-state } S) \subseteq \# \text{ learned-clss } S \text{ and}$

$\text{backtrack-lvl-restart-state}[simp]: \text{backtrack-lvl } (\text{restart-state } S) = 0 \text{ and}$

$\text{conflicting-restart-state}[simp]: \text{conflicting } (\text{restart-state } S) = C\text{-True}$

begin

definition $\text{clauses} :: 'st \Rightarrow 'v \text{ clauses}$ **where**

$\text{clauses } S = \text{init-clss } S + \text{learned-clss } S$

lemma

shows

$\text{clauses-cons-trail}[simp]:$

$\text{undefined-lit } (\text{trail } S) \ (\text{lit-of } M) \implies \text{clauses } (\text{cons-trail } M \ S) = \text{clauses } S \text{ and}$

$\text{clss-tl-trail}[simp]: \text{clauses } (\text{tl-trail } S) = \text{clauses } S \text{ and}$

$\text{clauses-add-learned-clss-unfolded}: \text{no-dup } (\text{trail } S) \implies$

$\text{clauses } (\text{add-learned-clss } U \ S) = \{\#U\# \} + \text{learned-clss } S + \text{init-clss } S$
and

$\text{clauses-add-init-clss}[simp]:$

$\text{no-dup } (\text{trail } S) \implies \text{clauses } (\text{add-init-clss } N \ S) = \{\#N\# \} + \text{init-clss } S + \text{learned-clss } S \text{ and}$

$\text{clauses-update-backtrack-lvl}[simp]: \text{clauses } (\text{update-backtrack-lvl } k \ S) = \text{clauses } S \text{ and}$

$\text{clauses-update-conflicting}[simp]: \text{clauses } (\text{update-conflicting } D \ S) = \text{clauses } S \text{ and}$

$\text{clauses-remove-clss}[simp]:$

$\text{clauses } (\text{remove-clss } C \ S) = \text{clauses } S - \text{replicate-mset } (\text{count } (\text{clauses } S) \ C) \ C \text{ and}$

$\text{clauses-add-learned-clss}[simp]:$

$\text{no-dup } (\text{trail } S) \implies \text{clauses } (\text{add-learned-clss } C \ S) = \{\#C\# \} + \text{clauses } S \text{ and}$

$\text{clauses-restart}[simp]: \text{clauses } (\text{restart-state } S) \subseteq \# \text{ clauses } S \text{ and}$

$\text{clauses-init-state}[simp]: \bigwedge N. \text{clauses } (\text{init-state } N) = N$

prefer 9 using $\text{clauses-def learned-clss-restart-state}$ **apply** fastforce

by $(\text{auto simp: ac-simps replicate-mset-plus clauses-def intro: multiset-eqI})$

abbreviation $\text{state} :: 'st \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit list} \times 'v \text{ clauses} \times 'v \text{ clauses}$

$\times \text{nat} \times 'v \text{ clause conflicting-clause}$ **where**

$\text{state } S \equiv (\text{trail } S, \text{init-clss } S, \text{learned-clss } S, \text{backtrack-lvl } S, \text{conflicting } S)$

abbreviation $\text{incr-lvl} :: 'st \Rightarrow 'st$ **where**

$\text{incr-lvl } S \equiv \text{update-backtrack-lvl } (\text{backtrack-lvl } S + 1) \ S$

definition $\text{state-eq} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **(infix \sim 50)** **where**

$S \sim T \iff \text{state } S = \text{state } T$

lemma *state-eq-ref*[*simp*, *intro*]:
 $S \sim S$
unfolding *state-eq-def* **by** *auto*

lemma *state-eq-sym*:
 $S \sim T \longleftrightarrow T \sim S$
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: $S \sim T \implies \text{trail } S = \text{trail } T$ **and**
state-eq-init-clss: $S \sim T \implies \text{init-clss } S = \text{init-clss } T$ **and**
state-eq-learned-clss: $S \sim T \implies \text{learned-clss } S = \text{learned-clss } T$ **and**
state-eq-backtrack-lvl: $S \sim T \implies \text{backtrack-lvl } S = \text{backtrack-lvl } T$ **and**
state-eq-conflicting: $S \sim T \implies \text{conflicting } S = \text{conflicting } T$ **and**
state-eq-clauses: $S \sim T \implies \text{clauses } S = \text{clauses } T$ **and**
state-eq-undefined-lit: $S \sim T \implies \text{undefined-lit } (\text{trail } S) L = \text{undefined-lit } (\text{trail } T) L$
unfolding *state-eq-def* *clauses-def* **by** *auto*

lemmas *state-simp*[*simp*] = *state-eq-trail* *state-eq-init-clss* *state-eq-learned-clss*
state-eq-backtrack-lvl *state-eq-conflicting* *state-eq-clauses* *state-eq-undefined-lit*

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*[*intro*]:
 $x \in \text{atms-of-msu } (\text{learned-clss } (\text{restart-state } S)) \implies x \in \text{atms-of-msu } (\text{learned-clss } S)$
by (*meson* *atms-of-ms-mono* *learned-clss-restart-state* *set-mset-mono* *subsetCE*)

function *reduce-trail-to* :: '*a list* \Rightarrow '*st* \Rightarrow '*st* **where**
reduce-trail-to *F* *S* =
 (*if* *length* (*trail* *S*) = *length* *F* \vee *trail* *S* = [] *then* *S* *else* *reduce-trail-to* *F* (*tl-trail* *S*))
by *fast+*
termination
by (*relation* *measure* ($\lambda(-, S). \text{length } (\text{trail } S)$)) *simp-all*

declare *reduce-trail-to.simps*[*simp* *del*]

lemma
shows
reduce-trail-to-nil[*simp*]: $\text{trail } S = [] \implies \text{reduce-trail-to } F S = S$ **and**
reduce-trail-to-eq-length[*simp*]: $\text{length } (\text{trail } S) = \text{length } F \implies \text{reduce-trail-to } F S = S$
by (*auto* *simp*: *reduce-trail-to.simps*)

lemma *reduce-trail-to-length-ne*:
 $\text{length } (\text{trail } S) \neq \text{length } F \implies \text{trail } S \neq [] \implies$
 $\text{reduce-trail-to } F S = \text{reduce-trail-to } F (\text{tl-trail } S)$
by (*auto* *simp*: *reduce-trail-to.simps*)

lemma *trail-reduce-trail-to-length-le*:
assumes $\text{length } F > \text{length } (\text{trail } S)$
shows $\text{trail } (\text{reduce-trail-to } F S) = []$

using *assms* **apply** (*induction* $F\ S$ *rule: reduce-trail-to.induct*)
by (*metis* (*no-types*, *hide-lams*) *length-tl less-imp-diff-less less-irrefl trail-tl-trail*
reduce-trail-to.simps)

lemma *trail-reduce-trail-to-nil[simp]*:
trail (*reduce-trail-to* $[]\ S$) = $[]$
apply (*induction* $[]$: ($'v$, *nat*, $'v$ *clause*) *marked-lits* S *rule: reduce-trail-to.induct*)
by (*metis* *length-0-conv reduce-trail-to-length-ne reduce-trail-to-nil*)

lemma *clauses-reduce-trail-to-nil*:
clauses (*reduce-trail-to* $[]\ S$) = *clauses* S
proof (*induction* $[]\ S$ *rule: reduce-trail-to.induct*)
case ($1\ Sa$)
then have *clauses* (*reduce-trail-to* ($[]$: $'a$ *list*) (*tl-trail* Sa)) = *clauses* (*tl-trail* Sa)
 \vee *trail* Sa = $[]$
by *fastforce*
then show *clauses* (*reduce-trail-to* ($[]$: $'a$ *list*) Sa) = *clauses* Sa
by (*metis* (*no-types*) *length-0-conv reduce-trail-to-eq-length clss-tl-trail*
reduce-trail-to-length-ne)
qed

lemma *reduce-trail-to-skip-beginning*:
assumes *trail* $S = F' @ F$
shows *trail* (*reduce-trail-to* $F\ S$) = F
using *assms* **by** (*induction* F' *arbitrary: S*) (*auto simp: reduce-trail-to-length-ne*)

lemma *clauses-reduce-trail-to[simp]*:
clauses (*reduce-trail-to* $F\ S$) = *clauses* S
apply (*induction* $F\ S$ *rule: reduce-trail-to.induct*)
by (*metis* *clss-tl-trail reduce-trail-to.simps*)

lemma *conflicting-update-trial[simp]*:
conflicting (*reduce-trail-to* $F\ S$) = *conflicting* S
apply (*induction* $F\ S$ *rule: reduce-trail-to.induct*)
by (*metis* *conflicting-tl-trail reduce-trail-to.simps*)

lemma *backtrack-lvl-update-trial[simp]*:
backtrack-lvl (*reduce-trail-to* $F\ S$) = *backtrack-lvl* S
apply (*induction* $F\ S$ *rule: reduce-trail-to.induct*)
by (*metis* *backtrack-lvl-tl-trail reduce-trail-to.simps*)

lemma *init-clss-update-trial[simp]*:
init-clss (*reduce-trail-to* $F\ S$) = *init-clss* S
apply (*induction* $F\ S$ *rule: reduce-trail-to.induct*)
by (*metis* *init-clss-tl-trail reduce-trail-to.simps*)

lemma *learned-clss-update-trial[simp]*:
learned-clss (*reduce-trail-to* $F\ S$) = *learned-clss* S
apply (*induction* $F\ S$ *rule: reduce-trail-to.induct*)
by (*metis* *learned-clss-tl-trail reduce-trail-to.simps*)

lemma *trail-eq-reduce-trail-to-eq*:
trail $S = \text{trail } T \implies \text{trail } (\text{reduce-trail-to } F\ S) = \text{trail } (\text{reduce-trail-to } F\ T)$
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-eqNOT-compatible:
  assumes ST:  $S \sim T$ 
  shows reduce-trail-to  $F S \sim \text{reduce-trail-to } F T$ 
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 -  $F' @ \text{Marked } K d \# []$ ])
  by (cases  $F'$ ) (auto simp add:tl-append reduce-trail-to-skip-beginning)

lemma reduce-trail-to-add-learned-cls[simp]:
  no-dup (trail S)  $\implies$ 
    trail (reduce-trail-to  $F$  (add-learned-cls  $C S$ )) = trail (reduce-trail-to  $F S$ )
  by (rule trail-eq-reduce-trail-to-eq) auto

lemma reduce-trail-to-add-init-cls[simp]:
  no-dup (trail S)  $\implies$ 
    trail (reduce-trail-to  $F$  (add-init-cls  $C S$ )) = trail (reduce-trail-to  $F S$ )
  by (rule trail-eq-reduce-trail-to-eq) auto

lemma reduce-trail-to-remove-learned-cls[simp]:
  trail (reduce-trail-to  $F$  (remove-cls  $C S$ )) = trail (reduce-trail-to  $F S$ )
  by (rule trail-eq-reduce-trail-to-eq) auto

lemma reduce-trail-to-update-conflicting[simp]:
  trail (reduce-trail-to  $F$  (update-conflicting  $C S$ )) = trail (reduce-trail-to  $F S$ )
  by (rule trail-eq-reduce-trail-to-eq) auto

lemma reduce-trail-to-update-backtrack-lvl[simp]:
  trail (reduce-trail-to  $F$  (update-backtrack-lvl  $C S$ )) = trail (reduce-trail-to  $F S$ )
  by (rule trail-eq-reduce-trail-to-eq) auto

lemma in-get-all-marked-decomposition-marked-or-empty:
  assumes  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
  shows  $a = [] \vee (\text{is-marked } (\text{hd } a))$ 
  using assms
proof (induct  $M$  arbitrary: a b)
  case Nil then show ?case by simp
next
  case (Cons  $m M$ )
  show ?case
  proof (cases  $m$ )
  case (Marked  $l \text{ mark}$ )
  then show ?thesis using Cons by auto
  next
  case (Propagated  $l \text{ mark}$ )
  then show ?thesis using Cons by (cases get-all-marked-decomposition  $M$ ) force+
qed
qed

```

```

lemma in-get-all-marked-decomposition-trail-update-trail[simp]:
  assumes  $H: (L \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
  shows  $\text{trail } (\text{reduce-trail-to } M1 \ S) = M1$ 
proof –
  obtain  $K \ \text{mark}$  where
     $L: L = \text{Marked } K \ \text{mark}$ 
  using  $H$  by  $(\text{cases } L) \ (\text{auto } \text{dest!}: \text{in-get-all-marked-decomposition-marked-or-empty})$ 
  obtain  $c$  where
     $\text{tr-}S: \text{trail } S = c @ M2 @ L \# M1$ 
  using  $H$  by  $\text{auto}$ 
  show  $?thesis$ 
  by  $(\text{rule } \text{reduce-trail-to-trail-tl-trail-decomp}[\text{of } - \ c @ M2 \ K \ \text{mark}])$ 
   $(\text{auto } \text{simp}: \text{tr-}S \ L)$ 
qed

```

```

fun append-trail where
   $\text{append-trail } [] \ S = S \mid$ 
   $\text{append-trail } (L \# M) \ S = \text{append-trail } M \ (\text{cons-trail } L \ S)$ 

```

```

lemma trail-append-trail[simp]:
   $\text{no-dup } (M @ \text{trail } S) \implies \text{trail } (\text{append-trail } M \ S) = \text{rev } M @ \text{trail } S$ 
  by  $(\text{induction } M \ \text{arbitrary}: S) \ (\text{auto } \text{simp}: \text{defined-lit-map})$ 

```

```

lemma learned-clss-append-trail[simp]:
   $\text{no-dup } (M @ \text{trail } S) \implies \text{learned-clss } (\text{append-trail } M \ S) = \text{learned-clss } S$ 
  by  $(\text{induction } M \ \text{arbitrary}: S) \ (\text{auto } \text{simp}: \text{defined-lit-map})$ 

```

```

lemma init-clss-append-trail[simp]:
   $\text{no-dup } (M @ \text{trail } S) \implies \text{init-clss } (\text{append-trail } M \ S) = \text{init-clss } S$ 
  by  $(\text{induction } M \ \text{arbitrary}: S) \ (\text{auto } \text{simp}: \text{defined-lit-map})$ 

```

```

lemma conflicting-append-trail[simp]:
   $\text{no-dup } (M @ \text{trail } S) \implies \text{conflicting } (\text{append-trail } M \ S) = \text{conflicting } S$ 
  by  $(\text{induction } M \ \text{arbitrary}: S) \ (\text{auto } \text{simp}: \text{defined-lit-map})$ 

```

```

lemma backtrack-lvl-append-trail[simp]:
   $\text{no-dup } (M @ \text{trail } S) \implies \text{backtrack-lvl } (\text{append-trail } M \ S) = \text{backtrack-lvl } S$ 
  by  $(\text{induction } M \ \text{arbitrary}: S) \ (\text{auto } \text{simp}: \text{defined-lit-map})$ 

```

```

lemma clauses-append-trail[simp]:
   $\text{no-dup } (M @ \text{trail } S) \implies \text{clauses } (\text{append-trail } M \ S) = \text{clauses } S$ 
  by  $(\text{induction } M \ \text{arbitrary}: S) \ (\text{auto } \text{simp}: \text{defined-lit-map})$ 

```

This function is useful for proofs to speak of a global trail change, but is a bad for programs and code in general.

```

fun delete-trail-and-rebuild where
   $\text{delete-trail-and-rebuild } M \ S = \text{append-trail } (\text{rev } M) \ (\text{reduce-trail-to } ([:: 'v \ \text{list}] \ S))$ 

```

end

17.2 Special Instantiation: using Triples as State

17.3 CDCL Rules

Because of the strategy we will later use, we distinguish propagate, conflict from the other rules

locale

cdcl_W-ops =
state_W *trail* *init-clss* *learned-clss* *backtrack-lvl* *conflicting* *cons-trail* *tl-trail* *add-init-cls*
add-learned-cls *remove-cls* *update-backtrack-lvl* *update-conflicting* *init-state*
restart-state

for

trail :: '*st* ⇒ (*v*, *nat*, '*v* clause') marked-lits **and**
init-clss :: '*st* ⇒ '*v* clauses **and**
learned-clss :: '*st* ⇒ '*v* clauses **and**
backtrack-lvl :: '*st* ⇒ *nat* **and**
conflicting :: '*st* ⇒ '*v* clause conflicting-clause **and**

cons-trail :: (*v*, *nat*, '*v* clause') marked-lit ⇒ '*st* ⇒ '*st* **and**
tl-trail :: '*st* ⇒ '*st* **and**
add-init-cls :: '*v* clause ⇒ '*st* ⇒ '*st* **and**
add-learned-cls :: '*v* clause ⇒ '*st* ⇒ '*st* **and**
remove-cls :: '*v* clause ⇒ '*st* ⇒ '*st* **and**
update-backtrack-lvl :: *nat* ⇒ '*st* ⇒ '*st* **and**
update-conflicting :: '*v* clause conflicting-clause ⇒ '*st* ⇒ '*st* **and**

init-state :: '*v* clauses ⇒ '*st* **and**
restart-state :: '*st* ⇒ '*st*

begin

inductive *propagate* :: '*st* ⇒ '*st* ⇒ *bool* **where**

propagate-rule[intro]:

state *S* = (*M*, *N*, *U*, *k*, *C-True*) ⇒ *C* + {*#L#*} ∈ # *clauses* *S* ⇒ *M* ⊨_{as} *CNot C*
⇒ *undefined-lit* (*trail* *S*) *L*
⇒ *T* ∼ *cons-trail* (*Propagated* *L* (*C* + {*#L#*})) *S*
⇒ *propagate* *S* *T*

inductive-cases *propagateE*[elim]: *propagate* *S* *T*

thm *propagateE*

inductive *conflict* :: '*st* ⇒ '*st* ⇒ *bool* **where**

conflict-rule[intro]: *state* *S* = (*M*, *N*, *U*, *k*, *C-True*) ⇒ *D* ∈ # *clauses* *S* ⇒ *M* ⊨_{as} *CNot D*
⇒ *T* ∼ *update-conflicting* (*C-Clause* *D*) *S*
⇒ *conflict* *S* *T*

inductive-cases *conflictE*[elim]: *conflict* *S* *S'*

inductive *backtrack* :: '*st* ⇒ '*st* ⇒ *bool* **where**

backtrack-rule[intro]: *state* *S* = (*M*, *N*, *U*, *k*, *C-Clause* (*D* + {*#L#*}))
⇒ (*Marked* *K* (*i*+1) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* *M*)
⇒ *get-level* *L* *M* = *k*
⇒ *get-level* *L* *M* = *get-maximum-level* (*D*+{*#L#*}) *M*
⇒ *get-maximum-level* *D* *M* = *i*
⇒ *T* ∼ *cons-trail* (*Propagated* *L* (*D*+{*#L#*}))
(*reduce-trail-to* *M1*
(*add-learned-cls* (*D* + {*#L#*}))
(*update-backtrack-lvl* *i*
(*update-conflicting* *C-True* *S*))))
⇒ *backtrack* *S* *T*

inductive-cases *backtrackE*[elim]: *backtrack* *S* *S'*

thm *backtrackE*

inductive *decide* :: 'st \Rightarrow 'st \Rightarrow bool **where**
decide-rule[intro]: state $S = (M, N, U, k, C\text{-True})$
 \Rightarrow undefined-lit $M\ L \Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$
 $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L\ (k+1))\ (\text{incr-lvl } S)$
 $\Rightarrow \text{decide } S\ T$
inductive-cases *decideE*[elim]: *decide* $S\ S'$
thm *decideE*

inductive *skip* :: 'st \Rightarrow 'st \Rightarrow bool **where**
skip-rule[intro]: state $S = (\text{Propagated } L\ C' \# M, N, U, k, C\text{-Clause } D) \Rightarrow -L \notin \# D \Rightarrow D \neq \{\#\}$
 $\Rightarrow T \sim \text{tl-trail } S$
 $\Rightarrow \text{skip } S\ T$
inductive-cases *skipE*[elim]: *skip* $S\ S'$
thm *skipE*

get-maximum-level $D\ (\text{Propagated } L\ (C + \{\#L\#\}) \# M) = k \vee k = 0$ is equivalent to
get-maximum-level $D\ (\text{Propagated } L\ (C + \{\#L\#\}) \# M) = k$

inductive *resolve* :: 'st \Rightarrow 'st \Rightarrow bool **where**
resolve-rule[intro]:
state $S = (\text{Propagated } L\ ((C + \{\#L\#\})) \# M, N, U, k, C\text{-Clause } (D + \{\#-L\#\}))$
 $\Rightarrow \text{get-maximum-level } D\ (\text{Propagated } L\ (C + \{\#L\#\}) \# M) = k$
 $\Rightarrow T \sim \text{update-conflicting } (C\text{-Clause } (D \# \cup C))\ (\text{tl-trail } S)$
 $\Rightarrow \text{resolve } S\ T$
inductive-cases *resolveE*[elim]: *resolve* $S\ S'$
thm *resolveE*

inductive *restart* :: 'st \Rightarrow 'st \Rightarrow bool **where**
restart: state $S = (M, N, U, k, C\text{-True}) \Rightarrow \neg M \models \text{asm clauses } S$
 $\Rightarrow T \sim \text{restart-state } S$
 $\Rightarrow \text{restart } S\ T$
inductive-cases *restartE*[elim]: *restart* $S\ T$
thm *restartE*

We add the condition $C \notin \# \text{init-clss } S$, to maintain consistency even without the strategy.

inductive *forget* :: 'st \Rightarrow 'st \Rightarrow bool **where**
forget-rule: state $S = (M, N, \{\#C\#\} + U, k, C\text{-True})$
 $\Rightarrow \neg M \models \text{asm clauses } S$
 $\Rightarrow C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$
 $\Rightarrow C \notin \# \text{init-clss } S$
 $\Rightarrow C \in \# \text{learned-clss } S$
 $\Rightarrow T \sim \text{remove-cl } C\ S$
 $\Rightarrow \text{forget } S\ T$
inductive-cases *forgetE*[elim]: *forget* $S\ T$

inductive *cdcl_W-rf* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S :: 'st$ **where**
restart: *restart* $S\ T \Rightarrow \text{cdcl}_W\text{-rf } S\ T$ |
forget: *forget* $S\ T \Rightarrow \text{cdcl}_W\text{-rf } S\ T$

inductive *cdcl_W-bj* :: 'st \Rightarrow 'st \Rightarrow bool **where**
skip[intro]: *skip* $S\ S' \Rightarrow \text{cdcl}_W\text{-bj } S\ S'$ |
resolve[intro]: *resolve* $S\ S' \Rightarrow \text{cdcl}_W\text{-bj } S\ S'$ |
backtrack[intro]: *backtrack* $S\ S' \Rightarrow \text{cdcl}_W\text{-bj } S\ S'$

inductive-cases *cdcl_W-bjE*: *cdcl_W-bj* $S\ T$

inductive $cdcl_W\text{-}o:: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
decide[intro]: $\text{decide } S \ S' \Longrightarrow cdcl_W\text{-}o \ S \ S' \mid$
bj[intro]: $cdcl_W\text{-}bj \ S \ S' \Longrightarrow cdcl_W\text{-}o \ S \ S'$

inductive $cdcl_W :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
propagate: $\text{propagate } S \ S' \Longrightarrow cdcl_W \ S \ S' \mid$
conflict: $\text{conflict } S \ S' \Longrightarrow cdcl_W \ S \ S' \mid$
other: $cdcl_W\text{-}o \ S \ S' \Longrightarrow cdcl_W \ S \ S' \mid$
rf: $cdcl_W\text{-}rf \ S \ S' \Longrightarrow cdcl_W \ S \ S'$

lemma *rtrancpl-propagate-is-rtrancpl-cdcl_W*:
 $\text{propagate}^{**} \ S \ S' \Longrightarrow cdcl_W^{**} \ S \ S'$
by (*induction rule: rtrancpl-induct*) (*fastforce dest!: propagate*)+

lemma *cdcl_W-all-rules-induct*[*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

fixes $S :: 'st$
assumes
 $cdcl_W: cdcl_W \ S \ S'$ **and**
propagate: $\bigwedge T. \text{propagate } S \ T \Longrightarrow P \ S \ T$ **and**
conflict: $\bigwedge T. \text{conflict } S \ T \Longrightarrow P \ S \ T$ **and**
forget: $\bigwedge T. \text{forget } S \ T \Longrightarrow P \ S \ T$ **and**
restart: $\bigwedge T. \text{restart } S \ T \Longrightarrow P \ S \ T$ **and**
decide: $\bigwedge T. \text{decide } S \ T \Longrightarrow P \ S \ T$ **and**
skip: $\bigwedge T. \text{skip } S \ T \Longrightarrow P \ S \ T$ **and**
resolve: $\bigwedge T. \text{resolve } S \ T \Longrightarrow P \ S \ T$ **and**
backtrack: $\bigwedge T. \text{backtrack } S \ T \Longrightarrow P \ S \ T$
shows $P \ S \ S'$
using *assms*(1)
proof (*induct S' rule: cdcl_W.induct*)
case (*propagate S'*) **note** *propagate = this*(1)
then show ?case **using** *assms*(2) **by** *auto*
next
case (*conflict S'*)
then show ?case **using** *assms*(3) **by** *auto*
next
case (*other S'*)
then show ?case
proof (*induct rule: cdcl_W-o.induct*)
case (*decide U*)
then show ?case **using** *assms*(6) **by** *auto*
next
case (*bj S'*)
then show ?case **using** *assms*(7–9) **by** (*induction rule: cdcl_W-bj.induct*) *auto*
qed
next
case (*rf S'*)
then show ?case
by (*induct rule: cdcl_W-rf.induct*) (*fast dest: forget restart*)
qed

lemma *cdcl_W-all-induct*[*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

fixes $S :: 'st$
assumes

$cdcl_W: cdcl_W S S' \text{ and}$
 $propagateH: \bigwedge C L T. C + \{\#L\# \} \in \# \text{ clauses } S \implies trail S \models_{as} CNot C$
 $\implies undefined-lit (trail S) L \implies conflicting S = C-True$
 $\implies T \sim cons-trail (Propagated L (C + \{\#L\# \})) S$
 $\implies P S T \text{ and}$
 $conflictH: \bigwedge D T. D \in \# \text{ clauses } S \implies conflicting S = C-True \implies trail S \models_{as} CNot D$
 $\implies T \sim update-conflicting (C-Clause D) S$
 $\implies P S T \text{ and}$
 $forgetH: \bigwedge C T. \neg trail S \models_{asm} clauses S$
 $\implies C \notin set (get-all-mark-of-propagated (trail S))$
 $\implies C \notin \# init-clss S$
 $\implies C \in \# learned-clss S$
 $\implies conflicting S = C-True$
 $\implies T \sim remove-cls C S$
 $\implies P S T \text{ and}$
 $restartH: \bigwedge T. \neg trail S \models_{asm} clauses S$
 $\implies conflicting S = C-True$
 $\implies T \sim restart-state S$
 $\implies P S T \text{ and}$
 $decideH: \bigwedge L T. conflicting S = C-True \implies undefined-lit (trail S) L$
 $\implies atm-of L \in atms-of-msu (init-clss S)$
 $\implies T \sim cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S)$
 $\implies P S T \text{ and}$
 $skipH: \bigwedge L C' M D T. trail S = Propagated L C' \# M$
 $\implies conflicting S = C-Clause D \implies -L \notin \# D \implies D \neq \{\#\}$
 $\implies T \sim tl-trail S$
 $\implies P S T \text{ and}$
 $resolveH: \bigwedge L C M D T.$
 $trail S = Propagated L ((C + \{\#L\# \})) \# M$
 $\implies conflicting S = C-Clause (D + \{\#-L\# \})$
 $\implies get-maximum-level D (Propagated L ((C + \{\#L\# \})) \# M) = backtrack-lvl S$
 $\implies T \sim (update-conflicting (C-Clause (D \# \cup C)) (tl-trail S))$
 $\implies P S T \text{ and}$
 $backtrackH: \bigwedge K i M1 M2 L D T.$
 $(Marked K (Suc i) \# M1, M2) \in set (get-all-marked-decomposition (trail S))$
 $\implies get-level L (trail S) = backtrack-lvl S$
 $\implies conflicting S = C-Clause (D + \{\#L\# \})$
 $\implies get-maximum-level (D + \{\#L\# \}) (trail S) = get-level L (trail S)$
 $\implies get-maximum-level D (trail S) \equiv i$
 $\implies T \sim cons-trail (Propagated L (D + \{\#L\# \}))$
 $\quad (reduce-trail-to M1$
 $\quad \quad (add-learned-cls (D + \{\#L\# \})$
 $\quad \quad \quad (update-backtrack-lvl i$
 $\quad \quad \quad \quad (update-conflicting C-True S))))$
 $\implies P S T$
shows $P S S'$
using $cdcl_W$
proof (induct $S S'$ rule: $cdcl_W$ -all-rules-induct)
case (propagate S')
then show ?case by (elim propagateE) (frule propagateH; simp)
next
case (conflict S')
then show ?case by (elim conflictE) (frule conflictH; simp)
next
case (restart S')

```

    then show ?case by (elim restartE) (frule restartH; simp)
next
  case (decide T)
  then show ?case by (elim decideE) (frule decideH; simp)
next
  case (backtrack S')
  then show ?case by (elim backtrackE) (frule backtrackH; simp del: state-simp add: state-eq-def)
next
  case (forget S')
  then show ?case using forgetH by auto
next
  case (skip S')
  then show ?case using skipH by auto
next
  case (resolve S')
  then show ?case by (elim resolveE) (frule resolveH; simp)
qed

```

lemma $cdcl_W\text{-}o\text{-induct}$ [consumes 1, case-names decide skip resolve backtrack]:

fixes $S :: 'st$

assumes $cdcl_W$: $cdcl_W\text{-}o\ S\ T$ **and**

$decideH$: $\bigwedge L\ T.$ $conflicting\ S = C\text{-}True \implies undefined\text{-}lit\ (trail\ S)\ L$
 $\implies atm\text{-}of\ L \in atm\text{-}of\text{-}msu\ (init\text{-}cls\ S)$
 $\implies T \sim cons\text{-}trail\ (Marked\ L\ (backtrack\text{-}lvl\ S + 1))\ (incr\text{-}lvl\ S)$
 $\implies P\ S\ T$ **and**

$skipH$: $\bigwedge L\ C'\ M\ D\ T.$ $trail\ S = Propagated\ L\ C' \# M$
 $\implies conflicting\ S = C\text{-}Clause\ D \implies -L \notin \# D \implies D \neq \{\#\}$
 $\implies T \sim tl\text{-}trail\ S$
 $\implies P\ S\ T$ **and**

$resolveH$: $\bigwedge L\ C\ M\ D\ T.$
 $trail\ S = Propagated\ L\ ((C + \{\#L\})) \# M$
 $\implies conflicting\ S = C\text{-}Clause\ (D + \{\#-L\})$
 $\implies get\text{-}maximum\text{-}level\ D\ (Propagated\ L\ (C + \{\#L\})) \# M = backtrack\text{-}lvl\ S$
 $\implies T \sim update\text{-}conflicting\ (C\text{-}Clause\ (D \# \cup C))\ (tl\text{-}trail\ S)$
 $\implies P\ S\ T$ **and**

$backtrackH$: $\bigwedge K\ i\ M1\ M2\ L\ D\ T.$
 $(Marked\ K\ (Suc\ i) \# M1, M2) \in set\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S))$
 $\implies get\text{-}level\ L\ (trail\ S) = backtrack\text{-}lvl\ S$
 $\implies conflicting\ S = C\text{-}Clause\ (D + \{\#L\})$
 $\implies get\text{-}level\ L\ (trail\ S) = get\text{-}maximum\text{-}level\ (D + \{\#L\})\ (trail\ S)$
 $\implies get\text{-}maximum\text{-}level\ D\ (trail\ S) \equiv i$
 $\implies T \sim cons\text{-}trail\ (Propagated\ L\ (D + \{\#L\}))$
 $\quad (reduce\text{-}trail\text{-}to\ M1$
 $\quad \quad (add\text{-}learned\text{-}cls\ (D + \{\#L\})$
 $\quad \quad \quad (update\text{-}backtrack\text{-}lvl\ i$
 $\quad \quad \quad \quad (update\text{-}conflicting\ C\text{-}True\ S))))$
 $\implies P\ S\ T$

shows $P\ S\ T$

using $cdcl_W$ **apply** (induct T rule: $cdcl_W\text{-}o.induct$)

using $assms(2)$ **apply** $auto[1]$

apply (elim $cdcl_W\text{-}bjE$ $skipE$ $resolveE$ $backtrackE$)

apply (frule $skipH$; simp)

apply (frule $resolveH$; simp)

apply (frule $backtrackH$; simp-all del: state-simp add: state-eq-def)

done

```

thm cdclW-o.induct
lemma cdclW-o-all-rules-induct[consumes 1, case-names decide backtrack skip resolve]:
  fixes S T :: 'st
  assumes
    cdclW-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: cdclW-o.induct) (auto simp: cdclW-bj.simps)

lemma cdclW-o-rule-cases[consumes 1, case-names decide backtrack skip resolve]:
  fixes S T :: 'st
  assumes
    cdclW-o S T and
    decide S T  $\implies P$  and
    backtrack S T  $\implies P$  and
    skip S T  $\implies P$  and
    resolve S T  $\implies P$ 
  shows P
  using assms by (auto simp: cdclW-o.simps cdclW-bj.simps)

```

17.4 Invariants

17.4.1 Properties of the trail

We here establish that: * the marks are exactly 1..k where k is the level * the consistency of the trail * the fact that there is no duplicate in the trail.

```

lemma backtrack-lit-skipped:
  assumes L: get-level L (trail S) = backtrack-lvl S
  and M1: (Marked K (i + 1) # M1, M2) ∈ set (get-all-marked-decomposition (trail S))
  and no-dup: no-dup (trail S)
  and bt-l: backtrack-lvl S = length (get-all-levels-of-marked (trail S))
  and order: get-all-levels-of-marked (trail S)
     $= \text{rev } ([1..<(1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))))]$ 
  shows atm-of L ∉ atm-of ' lits-of M1
proof
  let ?M = trail S
  assume L-in-M1: atm-of L ∈ atm-of ' lits-of M1
  obtain c where Mc: trail S = c @ M2 @ Marked K (i + 1) # M1 using M1 by blast
  have atm-of L ∉ atm-of ' lits-of c
    using L-in-M1 no-dup mk-disjoint-insert unfolding Mc lits-of-def by force
  have g-M-eq-g-M1: get-level L ?M = get-level L M1
    using L-in-M1 unfolding Mc by auto
  have g: get-all-levels-of-marked M1 = rev [1..<Suc i]
    using order unfolding Mc
    by (auto simp del: upt-simps dest!: append-cons-eq-upt-length-i
      simp add: rev-swap[symmetric])
  then have Max (set (0 # get-all-levels-of-marked (rev M1))) < Suc i by auto
  then have get-level L M1 < Suc i
    using get-rev-level-less-max-get-all-levels-of-marked[of L 0 rev M1] by linarith
  moreover have Suc i ≤ backtrack-lvl S using bt-l by (simp add: Mc g)
  ultimately show False using L g-M-eq-g-M1 by auto

```

qed

lemma *cdcl_W-distinctinv-1*:

assumes

cdcl_W S S' **and**

no-dup (trail S) **and**

backtrack-lvl S = length (get-all-levels-of-marked (trail S)) **and**

get-all-levels-of-marked (trail S) = rev [1.. $1 + \text{length (get-all-levels-of-marked (trail S))}$]

shows *no-dup (trail S')*

using *assms*

proof (*induct rule: cdcl_W-all-induct*)

case (*backtrack K i M1 M2 L D T*) **note** *decomp = this(1)* **and** *L = this(2)* **and** *T = this(6)* **and** *n-d = this(7)*

obtain *c* **where** *Mc: trail S = c @ M2 @ Marked K (i + 1) # M1*

using *decomp* **by** *auto*

have *no-dup (M2 @ Marked K (i + 1) # M1)*

using *Mc n-d* **by** *fastforce*

moreover **have** *atm-of L \notin ($\lambda l. \text{atm-of (lit-of l)}$)* ‘*set M1*

using *backtrack-lit-skipped[of L S K i M1 M2] L decomp backtrack.prem*

by (*fastforce simp add: lits-of-def*)

moreover **then** **have** *undefined-lit M1 L*

by (*simp add: defined-lit-map*)

ultimately **show** *?case* **using** *decomp T n-d* **by** *simp*

qed (*auto simp add: defined-lit-map*)

lemma *cdcl_W-consistent-inv-2*:

assumes

cdcl_W S S' **and**

no-dup (trail S) **and**

backtrack-lvl S = length (get-all-levels-of-marked (trail S)) **and**

get-all-levels-of-marked (trail S) = rev [1.. $1 + \text{length (get-all-levels-of-marked (trail S))}$]

shows *consistent-interp (lits-of (trail S'))*

using *cdcl_W-distinctinv-1[OF assms] distinctconsistent-interp* **by** *fast*

lemma *cdcl_W-o-bt*:

assumes

cdcl_W-o S S' **and**

backtrack-lvl S = length (get-all-levels-of-marked (trail S)) **and**

get-all-levels-of-marked (trail S) =

rev ([1.. $1 + \text{length (get-all-levels-of-marked (trail S))}$]) **and**

n-d[simp]: no-dup (trail S)

shows *backtrack-lvl S' = length (get-all-levels-of-marked (trail S'))*

using *assms*

proof (*induct rule: cdcl_W-o-induct*)

case (*backtrack K i M1 M2 L D T*) **note** *decomp = this(1)* **and** *T = this(6)* **and** *level = this(8)*

have [*simp*]: *trail (reduce-trail-to M1 S) = M1*

using *decomp* **by** *auto*

obtain *c* **where** *M: trail S = c @ M2 @ Marked K (i + 1) # M1* **using** *decomp* **by** *auto*

have *rev (get-all-levels-of-marked (trail S))*

= [1.. $1 + (\text{length (get-all-levels-of-marked (trail S))})$]

using *level* **by** (*auto simp: rev-swap[symmetric]*)

moreover **have** *atm-of L \notin ($\lambda l. \text{atm-of (lit-of l)}$)* ‘*set M1*

using *backtrack-lit-skipped[of L S K i M1 M2] backtrack(2,7,8,9) decomp*

by (*fastforce simp add: lits-of-def*)

moreover **then** **have** *undefined-lit M1 L*

```

    by (simp add: defined-lit-map)
  moreover then have no-dup (trail T)
    using T decomp n-d by (auto simp: defined-lit-map M)
  ultimately show ?case
    using T n-d unfolding M by (auto dest!: append-cons-eq-upt-length simp del: upt-simps)
qed auto

```

```

lemma cdclW-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.. $(1 + \text{length (get-all-levels-of-marked (trail S))})$ ]
  shows backtrack-lvl S' = length (get-all-levels-of-marked (trail S'))
  using assms by (induct rule: cdclW-rf.induct) auto

```

```

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)

```

```

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.premis by simp
  have k: ?k = length (get-all-levels-of-marked ?M)
    using decide.premis by auto
  have get-all-levels-of-marked ?M' = Suc ?k # get-all-levels-of-marked ?M by simp
  then have get-all-levels-of-marked ?M' = Suc ?k #
    rev [Suc 0.. $(1 + \text{length (get-all-levels-of-marked ?M)})$ ]
    using H by auto
  moreover have ... = rev [Suc 0.. $(1 + \text{length (get-all-levels-of-marked ?M)})$ ]
    unfolding k by simp
  finally show ?case using T undef by (auto simp add: defined-lit-map)
next
  case (backtrack K i M1 M2 L D T) note decomp = this(1) and confl = this(2) and T = this(6)
  and
    all-marked = this(8) and bt-lvl = this(7)

```

```

have atm-of  $L \notin (\lambda l. \text{atm-of } (\text{lit-of } l))$  ‘ set  $M1$ 
  using backtrack-lit-skipped[of  $L$   $S$   $K$   $i$   $M1$   $M2$ ] backtrack(2,7,8,9) decomp
  by (fastforce simp add: lits-of-def)
moreover then have undefined-lit  $M1$   $L$ 
  by (simp add: defined-lit-map)
then have [simp]: trail  $T = \text{Propagated } L (D + \{\#L\}) \# M1$ 
  using  $T$  decomp n-d by auto
obtain  $c$  where  $M$ : trail  $S = c @ M2 @ \text{Marked } K (i + 1) \# M1$  using decomp by auto
have get-all-levels-of-marked (rev (trail  $S$ ))
  = [Suc 0.. $2 + \text{length } (\text{get-all-levels-of-marked } c) + (\text{length } (\text{get-all-levels-of-marked } M2)$ 
    +  $\text{length } (\text{get-all-levels-of-marked } M1))$ ]
  using all-marked bt-lvl unfolding  $M$  by (auto simp add: rev-swap[symmetric] simp del: upt-simps)
then show ?case
  using  $T$  by (auto simp add: rev-swap  $M$  dest!: append-cons-eq-upt(1) simp del: upt-simps)
qed auto

```

We write $1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ instead of $\text{backtrack-lvl } S$ to avoid non termination of rewriting.

definition $\text{cdcl}_W\text{-}M\text{-level-inv } (S :: 'st) \longleftrightarrow$
 $\text{consistent-interp } (\text{lits-of } (\text{trail } S))$
 $\wedge \text{no-dup } (\text{trail } S)$
 $\wedge \text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$
 $\wedge \text{get-all-levels-of-marked } (\text{trail } S)$
 $= \text{rev } ([1.. $1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$])$

lemma $\text{cdcl}_W\text{-}M\text{-level-inv-decomp}$:
assumes $\text{cdcl}_W\text{-}M\text{-level-inv } S$
shows $\text{consistent-interp } (\text{lits-of } (\text{trail } S))$
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: $\text{cdcl}_W\text{-consistent-inv}$)

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: $\text{cdcl}_W\text{-consistent-inv}$)

lemma $\text{cdcl}_W\text{-}M\text{-level-inv-S0-cdcl}_W[\text{simp}]$:


```

cdclW-M-level-inv (init-state N)
unfolding cdclW-M-level-inv-def by auto

lemma cdclW-M-level-inv-get-level-le-backtrack-lvl:
  assumes inv: cdclW-M-level-inv S
  shows get-level L (trail S) ≤ backtrack-lvl S
proof –
  have get-all-levels-of-marked (trail S) = rev [1..1 + backtrack-lvl S]
    using inv unfolding cdclW-M-level-inv-def by auto
  then show ?thesis
    using get-rev-level-less-max-get-all-levels-of-marked[of L 0 rev (trail S)]
    by (auto simp: Max-n-upt)
qed

lemma backtrack-ex-decomp:
  assumes M-l: cdclW-M-level-inv S
  and i-S: i < backtrack-lvl S
  shows  $\exists K\ M1\ M2. (\text{Marked } K\ (i + 1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
proof –
  let ?M = trail S
  have
    g: get-all-levels-of-marked (trail S) = rev [Suc 0..Suc (backtrack-lvl S)]
    using M-l unfolding cdclW-M-level-inv-def by simp-all
  then have i+1 ∈ 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  $(\text{Marked } K\ (i + 1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
    unfolding tr-S apply (induct c rule: marked-lit-list-induct)
    apply auto[2]
    apply (case-tac hd (get-all-marked-decomposition (xs @ Marked K (Suc i) # c')))
    apply (case-tac get-all-marked-decomposition (xs @ Marked K (Suc i) # c'))
    by auto
  then show ?thesis by blast
qed

```

17.4.2 Better-Suited Induction Principle

Ew generalise the induction principle defined previously: the induction case for *backtrack* now includes the assumption that *undefined-lit M1 L*. This helps the simplifier and thus the automation.

```

lemma backtrack-induction-lev[consumes 1, case-names M-devel-inv backtrack]:
  assumes
    bt: backtrack S T and
    inv: cdclW-M-level-inv S and
    backtrackH:  $\bigwedge K\ i\ M1\ M2\ L\ D\ T.
      (\text{Marked } K\ (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))
      \implies \text{get-level } L\ (\text{trail } S) = \text{backtrack-lvl } S
      \implies \text{conflicting } S = \text{C-Clause } (D + \{\#L\# \})
      \implies \text{get-level } L\ (\text{trail } S) = \text{get-maximum-level } (D + \{\#L\# \})\ (\text{trail } S)
      \implies \text{get-maximum-level } D\ (\text{trail } S) \equiv i
      \implies \text{undefined-lit } M1\ L
      \implies T \sim \text{cons-trail } (\text{Propagated } L\ (D + \{\#L\# \}))$ 

```

(reduce-trail-to $M1$
 (add-learned-cls ($D + \{\#L\# \}$)
 (update-backtrack-lvl i
 (update-conflicting $C\text{-True } S$))))
 $\Rightarrow P S T$
shows $P S T$
proof –
obtain $K i M1 M2 L D$ **where**
 decomp: ($\text{Marked } K (\text{Suc } i) \# M1, M2 \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$) **and**
 L : $\text{get-level } L (\text{trail } S) = \text{backtrack-lvl } S$ **and**
 confl: $\text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ **and**
 lev-L: $\text{get-level } L (\text{trail } S) = \text{get-maximum-level } (D + \{\#L\# \}) (\text{trail } S)$ **and**
 lev-D: $\text{get-maximum-level } D (\text{trail } S) \equiv i$ **and**
 T : $T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$
 (reduce-trail-to $M1$
 (add-learned-cls ($D + \{\#L\# \}$)
 (update-backtrack-lvl i
 (update-conflicting $C\text{-True } S$))))
using bt **by** (elim backtrackE) *metis*

have $\text{atm-of } L \notin (\lambda l. \text{atm-of } (\text{lit-of } l))$ ‘ $\text{set } M1$
using $\text{backtrack-lit-skipped}[of L S K i M1 M2]$ L *decomp* bt *confl* *lev-L* *lev-D* *inv*
unfolding $\text{cdcl}_W\text{-M-level-inv-def}$
by (*fastforce simp add: lits-of-def*)
then have $\text{undefined-lit } M1 L$
by (*auto simp: defined-lit-map*)
then show *?thesis*
using $\text{backtrackH}[OF \text{decomp } L \text{confl lev-L lev-D - } T]$ **by** *simp*
qed

lemmas $\text{backtrack-induction-lev2} = \text{backtrack-induction-lev}[\text{consumes } 2, \text{case-names backtrack}]$

lemma $\text{cdcl}_W\text{-all-induct-lev-full}$:

fixes $S :: 'st$

assumes

cdcl_W : $\text{cdcl}_W S S'$ **and**

$\text{inv}[\text{simp}]$: $\text{cdcl}_W\text{-M-level-inv } S$ **and**

propagateH : $\bigwedge C L T. C + \{\#L\# \} \in \# \text{ clauses } S \Rightarrow \text{trail } S \models_{as} C\text{Not } C$

$\Rightarrow \text{undefined-lit } (\text{trail } S) L \Rightarrow \text{conflicting } S = C\text{-True}$

$\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S$

$\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$

$\Rightarrow P S T$ **and**

conflictH : $\bigwedge D T. D \in \# \text{ clauses } S \Rightarrow \text{conflicting } S = C\text{-True} \Rightarrow \text{trail } S \models_{as} C\text{Not } D$

$\Rightarrow T \sim \text{update-conflicting } (C\text{-Clause } D) S$

$\Rightarrow P S T$ **and**

forgetH : $\bigwedge C T. \neg \text{trail } S \models_{asm} \text{ clauses } S$

$\Rightarrow C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$

$\Rightarrow C \notin \# \text{ init-clss } S$

$\Rightarrow C \in \# \text{ learned-clss } S$

$\Rightarrow \text{conflicting } S = C\text{-True}$

$\Rightarrow T \sim \text{remove-cls } C S$

$\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$

$\Rightarrow P S T$ **and**

restartH : $\bigwedge T. \neg \text{trail } S \models_{asm} \text{ clauses } S$

$\Rightarrow \text{conflicting } S = C\text{-True}$

```

     $\Rightarrow T \sim \text{restart-state } S$ 
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T \text{ and}$ 
    decideH:  $\bigwedge L \ T. \text{conflicting } S = C\text{-True} \Rightarrow \text{undefined-lit } (\text{trail } S) \ L$ 
     $\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$ 
     $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L \ (\text{backtrack-lvl } S + 1)) \ (\text{incr-lvl } S)$ 
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T \text{ and}$ 
    skipH:  $\bigwedge L \ C' \ M \ D \ T. \text{trail } S = \text{Propagated } L \ C' \ \# \ M$ 
     $\Rightarrow \text{conflicting } S = C\text{-Clause } D \Rightarrow -L \notin \# \ D \Rightarrow D \neq \{\#\}$ 
     $\Rightarrow T \sim \text{tl-trail } S$ 
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T \text{ and}$ 
    resolveH:  $\bigwedge L \ C \ M \ D \ T.$ 
     $\text{trail } S = \text{Propagated } L \ ( (C + \{\#L\# \}) \ \# \ M$ 
     $\Rightarrow \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ 
     $\Rightarrow \text{get-maximum-level } D \ (\text{Propagated } L \ ( (C + \{\#L\# \}) \ \# \ M) = \text{backtrack-lvl } S$ 
     $\Rightarrow T \sim (\text{update-conflicting } (C\text{-Clause } (D \ \# \cup \ C)) \ (\text{tl-trail } S))$ 
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T \text{ and}$ 
    backtrackH:  $\bigwedge K \ i \ M1 \ M2 \ L \ D \ T.$ 
     $(\text{Marked } K \ (\text{Suc } i) \ \# \ M1, \ M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
     $\Rightarrow \text{get-level } L \ (\text{trail } S) = \text{backtrack-lvl } S$ 
     $\Rightarrow \text{conflicting } S = C\text{-Clause } (D + \{\#L\# \})$ 
     $\Rightarrow \text{get-maximum-level } (D + \{\#L\# \}) \ (\text{trail } S) = \text{get-level } L \ (\text{trail } S)$ 
     $\Rightarrow \text{get-maximum-level } D \ (\text{trail } S) \equiv i$ 
     $\Rightarrow \text{undefined-lit } M1 \ L$ 
     $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L \ (D + \{\#L\# \}))$ 
     $\quad (\text{reduce-trail-to } M1$ 
     $\quad \quad (\text{add-learned-cls } (D + \{\#L\# \})$ 
     $\quad \quad \quad (\text{update-backtrack-lvl } i$ 
     $\quad \quad \quad \quad (\text{update-conflicting } C\text{-True } S))))$ 
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T$ 
  shows  $P \ S \ S'$ 
  using  $\text{cdcl}_W$ 
  proof (induct  $S'$  rule:  $\text{cdcl}_W\text{-all-rules-induct}$ )
  case (propagate  $S'$ )
  then show ?case by (elim propagateE) (frule propagateH; simp)
next
  case (conflict  $S'$ )
  then show ?case by (elim conflictE) (frule conflictH; simp)
next
  case (restart  $S'$ )
  then show ?case by (elim restartE) (frule restartH; simp)
next
  case (decide  $T$ )
  then show ?case by (elim decideE) (frule decideH; simp)
next
  case (backtrack  $S'$ )
  then show ?case
  apply (induction rule: backtrack-induction-lev)
  apply (rule inv)
  by (rule backtrackH;
    fastforce simp del: state-simp simp add: state-eq-def dest!: HOL.meta-eq-to-obj-eq)

```

```

next
  case (forget S')
  then show ?case using forgetH by auto
next
  case (skip S')
  then show ?case using skipH by auto
next
  case (resolve S')
  then show ?case by (elim resolveE) (frule resolveH; simp)
qed

```

lemmas $cdcl_W\text{-all-induct-lev2} = cdcl_W\text{-all-induct-lev-full}[\text{consumes } 2, \text{case-names propagate conflict forget restart decide skip resolve backtrack}]$

lemmas $cdcl_W\text{-all-induct-lev} = cdcl_W\text{-all-induct-lev-full}[\text{consumes } 1, \text{case-names lev-inv propagate conflict forget restart decide skip resolve backtrack}]$

thm $cdcl_W\text{-o-induct}$

lemma $cdcl_W\text{-o-induct-lev}[\text{consumes } 1, \text{case-names } M\text{-lev decide skip resolve backtrack}]$:

fixes $S :: 'st$

assumes

$cdcl_W$: $cdcl_W\text{-o } S \ T \text{ and}$

$inv[simp]$: $cdcl_W\text{-M-level-inv } S \text{ and}$

$decideH$: $\bigwedge L \ T. \text{conflicting } S = C\text{-True} \implies \text{undefined-lit } (trail \ S) \ L$

$\implies atm\text{-of } L \in atm\text{-of-msu } (init\text{-cls } S)$

$\implies T \sim cons\text{-trail } (Marked \ L \ (backtrack\text{-lvl } S + 1)) \ (incr\text{-lvl } S)$

$\implies cdcl_W\text{-M-level-inv } S$

$\implies P \ S \ T \text{ and}$

$skipH$: $\bigwedge L \ C' \ M \ D \ T. trail \ S = Propagated \ L \ C' \ \# \ M$

$\implies \text{conflicting } S = C\text{-Clause } D \implies -L \notin \# \ D \implies D \neq \{\#\}$

$\implies T \sim tl\text{-trail } S$

$\implies cdcl_W\text{-M-level-inv } S$

$\implies P \ S \ T \text{ and}$

$resolveH$: $\bigwedge L \ C \ M \ D \ T.$

$trail \ S = Propagated \ L \ ((C + \{\#L\#\}) \ \# \ M$

$\implies \text{conflicting } S = C\text{-Clause } (D + \{\#-L\#\})$

$\implies get\text{-maximum-level } D \ (Propagated \ L \ (C + \{\#L\#\}) \ \# \ M) = backtrack\text{-lvl } S$

$\implies T \sim update\text{-conflicting } (C\text{-Clause } (D \ \# \cup \ C)) \ (tl\text{-trail } S)$

$\implies cdcl_W\text{-M-level-inv } S$

$\implies P \ S \ T \text{ and}$

$backtrackH$: $\bigwedge K \ i \ M1 \ M2 \ L \ D \ T.$

$(Marked \ K \ (Suc \ i) \ \# \ M1, \ M2) \in set \ (get\text{-all-marked-decomposition } (trail \ S))$

$\implies get\text{-level } L \ (trail \ S) = backtrack\text{-lvl } S$

$\implies \text{conflicting } S = C\text{-Clause } (D + \{\#L\#\})$

$\implies get\text{-level } L \ (trail \ S) = get\text{-maximum-level } (D + \{\#L\#\}) \ (trail \ S)$

$\implies get\text{-maximum-level } D \ (trail \ S) \equiv i$

$\implies \text{undefined-lit } M1 \ L$

$\implies T \sim cons\text{-trail } (Propagated \ L \ (D + \{\#L\#\}))$

$(reduce\text{-trail-to } M1$

$(add\text{-learned-cls } (D + \{\#L\#\})$

$(update\text{-backtrack-lvl } i$

$(update\text{-conflicting } C\text{-True } S))))$

$\implies cdcl_W\text{-M-level-inv } S$

$\implies P \ S \ T$

shows $P \ S \ T$

```

  using cdclW
proof (induct S T rule: cdclW-o-all-rules-induct)
  case (decide T)
  then show ?case by (elim decideE) (frule decideH; simp)
next
  case (backtrack S')
  then show ?case
    using inv apply (induction rule: backtrack-induction-lev2)
    by (rule backtrackH)
    (fastforce simp del: state-simp simp add: state-eq-def dest!: HOL.meta-eq-to-obj-eq)+
next
  case (skip S')
  then show ?case using skipH by auto
next
  case (resolve S')
  then show ?case by (elim resolveE) (frule resolveH; simp)
qed

lemmas cdclW-o-induct-lev2 = cdclW-o-induct-lev[consumes 2, case-names decide skip resolve
  backtrack]

```

17.4.3 Compatibility with $op \sim$

```

lemma propagate-state-eq-compatible:
  assumes
    propagate S T and
    S  $\sim$  S' and
    T  $\sim$  T'
  shows propagate S' T'
  using assms apply (elim propagateE)
  apply (rule propagate-rule)
  by (auto simp: state-eq-def clauses-def simp del: state-simp)

```

```

lemma conflict-state-eq-compatible:
  assumes
    conflict S T and
    S  $\sim$  S' and
    T  $\sim$  T'
  shows conflict S' T'
  using assms apply (elim conflictE)
  apply (rule conflict-rule)
  by (auto simp: state-eq-def clauses-def simp del: state-simp)

```

```

lemma backtrack-state-eq-compatible:
  assumes
    backtrack S T and
    S  $\sim$  S' and
    T  $\sim$  T' and
    inv: cdclW-M-level-inv S
  shows backtrack S' T'
  using assms apply (induction rule: backtrack-induction-lev)
  using inv apply simp
  apply (rule backtrack-rule)
  apply auto[5]
  by (auto simp: state-eq-def clauses-def cdclW-M-level-inv-def simp del: state-simp)

```

lemma *decide-state-eq-compatible*:

assumes
 decide S T and
 S ~ S' and
 T ~ T'
shows *decide S' T'*
using *assms apply (elim decideE)*
apply (*rule decide-rule*)
by (*auto simp: state-eq-def clauses-def simp del: state-simp*)

lemma *skip-state-eq-compatible*:

assumes
 skip S T and
 S ~ S' and
 T ~ T'
shows *skip S' T'*
using *assms apply (elim skipE)*
apply (*rule skip-rule*)
by (*auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of - # - trail -]*
 simp del: state-simp dest: arg-cong[of - # trail - trail - tl]))

lemma *resolve-state-eq-compatible*:

assumes
 resolve S T and
 S ~ S' and
 T ~ T'
shows *resolve S' T'*
using *assms apply (elim resolveE)*
apply (*rule resolve-rule*)
by (*auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of - # - trail -]*
 simp del: state-simp dest: arg-cong[of - # trail - trail - tl]))

lemma *forget-state-eq-compatible*:

assumes
 forget S T and
 S ~ S' and
 T ~ T'
shows *forget S' T'*
using *assms apply (elim forgetE)*
apply (*rule forget-rule*)
by (*auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of {#-#} + - -]*
 simp del: state-simp dest: arg-cong[of - # trail - trail - tl]))

lemma *cdcl_W-state-eq-compatible*:

assumes
 cdcl_W S T and ¬restart S T and
 S ~ S' and
 T ~ T' and
 inv: cdcl_W-M-level-inv S
shows *cdcl_W S' T'*
using *assms by (meson assms backtrack-state-eq-compatible bj cdcl_W.simps cdcl_W-bj.simps*
 cdcl_W-o-rule-cases cdcl_W-rf.cases cdcl_W-rf.restart conflict-state-eq-compatible decide
 decide-state-eq-compatible forget forget-state-eq-compatible
 propagate-state-eq-compatible resolve-state-eq-compatible
 skip-state-eq-compatible)

lemma *cdcl_W-bj-state-eq-compatible*:
assumes
 $cdcl_W\text{-bj } S \text{ } T$ **and** $cdcl_W\text{-M-level-inv } S$
 $S \sim S'$ **and**
 $T \sim T'$
shows $cdcl_W\text{-bj } S' \text{ } T'$
using *assms*
by induction (*auto*
intro: skip-state-eq-compatible backtrack-state-eq-compatible resolve-state-eq-compatible)

lemma *trancpl-cdcl_W-bj-state-eq-compatible*:
assumes
 $cdcl_W\text{-bj}^{++} S \text{ } T$ **and** *inv: cdcl_W-M-level-inv S* **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows $cdcl_W\text{-bj}^{++} S' \text{ } T'$
using *assms*
proof (*induction arbitrary: S' T'*)
case base
then show *?case*
using *cdcl_W-bj-state-eq-compatible* **by** *blast*
next
case (*step T U*) **note** $IH = \text{this}(3)[OF \text{this}(4-5)]$
have $cdcl_W^{++} S \text{ } T$
using *trancpl-mono[of cdcl_W-bj cdcl_W]* *other step.hyps(1)* **by** *blast*
then have $cdcl_W\text{-M-level-inv } T$
using *inv trancpl-cdcl_W-consistent-inv* **by** *blast*
then have $cdcl_W\text{-bj}^{++} T \text{ } T'$
using $\langle U \sim T' \rangle$ *cdcl_W-bj-state-eq-compatible[of T U]* $\langle cdcl_W\text{-bj } T \text{ } U \rangle$ **by** *auto*
then show *?case*
using $IH[of T]$ **by** *auto*
qed

17.4.4 Conservation of some Properties

lemma *level-of-marked-ge-1*:
assumes
 $cdcl_W S \text{ } S'$ **and**
inv: cdcl_W-M-level-inv S **and**
 $\forall L \text{ } l. \text{Marked } L \text{ } l \in \text{set } (\text{trail } S) \longrightarrow l > 0$
shows $\forall L \text{ } l. \text{Marked } L \text{ } l \in \text{set } (\text{trail } S') \longrightarrow l > 0$
using *assms* **apply** (*induct rule: cdcl_W-all-induct-lev2*)
by (*auto dest: union-in-get-all-marked-decomposition-is-subset simp: cdcl_W-M-level-inv-decomp*)

lemma *cdcl_W-o-no-more-init-clss*:
assumes
 $cdcl_W\text{-o } S \text{ } S'$ **and**
inv: cdcl_W-M-level-inv S
shows $\text{init-clss } S = \text{init-clss } S'$
using *assms* **by** (*induct rule: cdcl_W-o-induct-lev2*) (*auto simp: cdcl_W-M-level-inv-decomp*)

lemma *trancpl-cdcl_W-o-no-more-init-clss*:
assumes
 $cdcl_W\text{-o}^{++} S \text{ } S'$ **and**
inv: cdcl_W-M-level-inv S

shows $init-clss\ S = init-clss\ S'$
using *assms* **apply** (*induct rule: tranclp.induct*)
by (*auto dest: cdcl_W-o-no-more-init-clss*
dest!: tranclp-cdcl_W-consistent-inv dest: tranclp-mono-explicit[of cdcl_W-o - - cdcl_W]
simp: other)

lemma *rtranclp-cdcl_W-o-no-more-init-clss:*

assumes
*cdcl_W-o** S S' and*
inv: cdcl_W-M-level-inv S
shows $init-clss\ S = init-clss\ S'$
using *assms* **unfolding** *rtranclp-unfold* **by** (*auto intro: tranclp-cdcl_W-o-no-more-init-clss*)

lemma *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: cdcl_W-all-induct-lev2*) (*auto simp: cdcl_W-M-level-inv-def*)

lemma *rtranclp-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: rtranclp-induct*) (*auto dest: cdcl_W-init-clss rtranclp-cdcl_W-consistent-inv*)

lemma *tranclp-cdcl_W-init-clss:*

*cdcl_W** S T \implies cdcl_W-M-level-inv S \implies init-clss S = init-clss T*
using *rtranclp-cdcl_W-init-clss[of S T]* **unfolding** *rtranclp-unfold* **by** *auto*

17.4.5 Learned Clause

This invariant shows that:

- the learned clauses are entailed by the initial set of clauses.
- the conflicting clause is entailed by the initial set of clauses.
- the marks are entailed by the clauses. A more precise version would be to show that either these marked are learned or are in the set of clauses

definition *cdcl_W-learned-clause (S:: 'st) \longleftrightarrow*

(init-clss S \models_{psm} learned-clss S
 $\wedge (\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{init-clss } S \models_{pm} T)$
 $\wedge \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \subseteq \text{set-mset } (\text{clauses } S))$

lemma *cdcl_W-learned-clause-S0-cdcl_W[simp]:*

cdcl_W-learned-clause (init-state N)
unfolding *cdcl_W-learned-clause-def* **by** *auto*

lemma *cdcl_W-learned-clss:*

assumes
cdcl_W S S' and
learned: cdcl_W-learned-clause S and
lev-inv: cdcl_W-M-level-inv S
shows *cdcl_W-learned-clause S'*
using *assms(1) lev-inv learned*

proof (*induct rule: cdcl_W-all-induct-lev2*)

case (*backtrack K i M1 M2 L D T*) **note** *decomp = this(1) and confl = this(3) and undef = this(6)*


```

and  $T = \text{this}(\gamma)$ 
show ?case
  using decomp confl learned undef  $T$  lev-inv unfolding cdclW-learned-clause-def
  by (auto dest!: get-all-marked-decomposition-exists-prepend
      simp: clauses-def cdclW-M-level-inv-decomp dest: true-clss-clss-left-right)
next
case (resolve  $L$   $C$   $M$   $D$ ) note trail = this(1) and confl = this(2) and lvl = this(3) and
 $T = \text{this}(4)$ 
moreover
  have init-clss  $S \models_{\text{psm}}$  learned-clss  $S$ 
    using learned trail unfolding cdclW-learned-clause-def clauses-def by auto
  then have init-clss  $S \models_{\text{pm}}$   $C + \{\#L\# \}$ 
    using trail learned unfolding cdclW-learned-clause-def clauses-def
    by (auto dest: true-clss-clss-in-imp-true-clss-clss)
  ultimately show ?case
    using learned
    by (auto dest: mk-disjoint-insert true-clss-clss-left-right
        simp add: cdclW-learned-clause-def clauses-def
        intro: true-clss-clss-union-mset-true-clss-clss-or-not-true-clss-clss-or)
next
case (restart  $T$ )
then show ?case
  using learned-clss-restart-state[of  $T$ ]
  by (auto dest!: get-all-marked-decomposition-exists-prepend
      simp: clauses-def state-eq-def cdclW-learned-clause-def
      simp del: state-simp
      dest: true-clss-clssm-subsetE)
next
case propagate
then show ?case using learned by (auto simp: cdclW-learned-clause-def clauses-def)
next
case conflict
then show ?case using learned
  by (auto simp: cdclW-learned-clause-def clauses-def true-clss-clss-in-imp-true-clss-clss)
next
case forget
then show ?case
  using learned by (auto simp: cdclW-learned-clause-def clauses-def split: split-if-asm)
qed (auto simp: cdclW-learned-clause-def clauses-def)

lemma rtranclp-cdclW-learned-clss:
assumes
   $\text{cdcl}_W^{**} S S'$  and
   $\text{cdcl}_W\text{-M-level-inv } S$ 
   $\text{cdcl}_W\text{-learned-clause } S$ 
shows  $\text{cdcl}_W\text{-learned-clause } S'$ 
using assms by induction (auto dest: cdclW-learned-clss intro: rtranclp-cdclW-consistent-inv)

```

17.4.6 No alien atom in the state

This invariant means that all the literals are in the set of clauses.

definition *no-strange-atm* $S' \longleftrightarrow$ (
 $(\forall T. \text{conflicting } S' = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S'))$
 $\wedge (\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S') \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S'))$)

$\wedge \text{atms-of-msu } (\text{learned-clss } S') \subseteq \text{atms-of-msu } (\text{init-clss } S')$
 $\wedge \text{atm-of } ' (\text{lits-of } (\text{trail } S')) \subseteq \text{atms-of-msu } (\text{init-clss } S')$

lemma *no-strange-atm-decomp*:

assumes *no-strange-atm* S
shows *conflicting* $S = C\text{-Clause } T \implies \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S)$
and $(\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S))$
 $\longrightarrow \text{atms-of } (\text{ mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S))$
and $\text{atms-of-msu } (\text{learned-clss } S) \subseteq \text{atms-of-msu } (\text{init-clss } S)$
and $\text{atm-of } ' (\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-msu } (\text{init-clss } S)$
using *assms* **unfolding** *no-strange-atm-def* **by** *blast+*

lemma *no-strange-atm-S0* [*simp*]: *no-strange-atm* (*init-state* N)

unfolding *no-strange-atm-def* **by** *auto*

lemma *cdcl_W-no-strange-atm-explicit*:

assumes
 $\text{cdcl}_W S S'$ **and**
 $\text{lev: cdcl}_W\text{-M-level-inv } S$ **and**
 $\text{conf: } \forall T. \text{ conflicting } S = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S)$ **and**
 $\text{marked: } \forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S)$
 $\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-msu } (\text{init-clss } S)$ **and**
 $\text{learned: atms-of-msu } (\text{learned-clss } S) \subseteq \text{atms-of-msu } (\text{init-clss } S)$ **and**
 $\text{trail: atm-of } ' (\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-msu } (\text{init-clss } S)$
shows $(\forall T. \text{ conflicting } S' = C\text{-Clause } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S')) \wedge$
 $(\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S'))$
 $\longrightarrow \text{atms-of } (\text{ mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S')) \wedge$
 $\text{atms-of-msu } (\text{learned-clss } S') \subseteq \text{atms-of-msu } (\text{init-clss } S') \wedge$
 $\text{atm-of } ' (\text{lits-of } (\text{trail } S')) \subseteq \text{atms-of-msu } (\text{init-clss } S') \text{ (is } ?C S' \wedge ?M S' \wedge ?U S' \wedge ?V S')$
using *assms*(1,2)

proof (*induct rule: cdcl_W-all-induct-lev2*)

case (*propagate* $C L T$) **note** $C\text{-L} = \text{this}(1)$ **and** $\text{undef} = \text{this}(3)$ **and** $\text{confl} = \text{this}(4)$ **and** $T = \text{this}(5)$

have $?C (\text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S)$ **using** *confl undef* **by** *auto*

moreover

have $\text{atms-of } (C + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{init-clss } S)$

by (*metis* (*no-types*) *atms-of-atms-of-ms-mono* *atms-of-ms-union* *clauses-def* *mem-set-mset-iff* $C\text{-L}$ *learned set-mset-union sup.orderE*)

then have $?M (\text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S)$ **using** *undef*

by (*simp add: marked*)

moreover have $?U (\text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S)$

using *learned undef* **by** *auto*

moreover have $?V (\text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S)$

using $C\text{-L}$ *learned trail undef* **unfolding** *clauses-def*

by (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

ultimately show $?case$ **using** T **by** *auto*

next

case (*decide* L)

then show $?case$ **using** *learned marked confl trail* **unfolding** *clauses-def* **by** *auto*

next

case (*skip* $L C M D$)

then show $?case$ **using** *learned marked confl trail* **by** *auto*

next

case (*conflict* $D T$) **note** $T = \text{this}(4)$

have $D: \text{atm-of } ' \text{ set-mset } D \subseteq \bigcup (\text{atms-of } ' (\text{set-mset } (\text{clauses } S)))$

using $\langle D \in \# \text{ clauses } S \rangle$ **by** (*auto simp add: atms-of-def atms-of-ms-def*)

```

moreover {
  fix  $xa :: 'v \text{ literal}$ 
  assume  $a1: atm\text{-}of \text{ ' set-mset } D \subseteq (\bigcup x \in set\text{-}mset (init\text{-}clss \ S). \ atm\text{-}of \ x)$ 
     $\cup (\bigcup x \in set\text{-}mset (learned\text{-}clss \ S). \ atm\text{-}of \ x)$ 
  assume  $a2: (\bigcup x \in set\text{-}mset (learned\text{-}clss \ S). \ atm\text{-}of \ x) \subseteq (\bigcup x \in set\text{-}mset (init\text{-}clss \ S). \ atm\text{-}of \ x)$ 
  assume  $xa \in \# \ D$ 
  then have  $atm\text{-}of \ xa \in UNION \ (set\text{-}mset \ (init\text{-}clss \ S)) \ atm\text{-}of$ 
    using  $a2 \ a1$  by  $(metis \ (no\text{-}types) \ Un\text{-}iff \ atm\text{-}of\text{-}lit\text{-}in\text{-}atms\text{-}of \ atms\text{-}of\text{-}def \ subset\text{-}Un\text{-}eq)$ 
  then have  $\exists m \in set\text{-}mset \ (init\text{-}clss \ S). \ atm\text{-}of \ xa \in atms\text{-}of \ m$ 
    by  $blast$ 
} note  $H = this$ 
ultimately show  $?case \text{ using } conflict.premis \ T \text{ learned marked conf trail}$ 
  unfolding  $atms\text{-}of\text{-}def \ atms\text{-}of\text{-}ms\text{-}def \ clauses\text{-}def$ 
  by  $(auto \ simp \ add: \ H)$ 
next
  case  $(restart \ T)$ 
  then show  $?case \text{ using } learned \text{ marked conf trail by } auto$ 
next
  case  $(forget \ C \ T)$  note  $C = this(3)$  and  $C\text{-}le = this(4)$  and  $confl = this(5)$  and
     $T = this(6)$ 
  have  $H: \bigwedge L \text{ mark. } Propagated \ L \text{ mark} \in set \ (trail \ S) \implies atms\text{-}of \ mark \subseteq atms\text{-}of\text{-}msu \ (init\text{-}clss \ S)$ 
    using  $marked$  by  $simp$ 
  show  $?case \text{ unfolding } clauses\text{-}def \text{ apply } standard$ 
    using  $conf \ T \ trail \ C$  unfolding  $clauses\text{-}def$  apply  $(auto \ dest!: \ H)[]$ 
    apply  $standard$ 
    using  $T \ trail \ C$  apply  $(auto \ dest!: \ H)[]$ 
    apply  $standard$ 
    using  $T \text{ learned } C \ C\text{-}le \ atms\text{-}of\text{-}ms\text{-}remove\text{-}subset[of \ set\text{-}mset \ (learned\text{-}clss \ S)]$  apply  $(auto)[]$ 
    using  $T \ trail \ C$  apply  $(auto \ simp: \ clauses\text{-}def \ lits\text{-}of\text{-}def)[]$ 
  done
next
  case  $(backtrack \ K \ i \ M1 \ M2 \ L \ D \ T)$  note  $decomp = this(1)$  and  $confl = this(3)$  and  $undef = this(6)$ 
    and  $T = this(7)$ 
  have  $?C \ T$ 
    using  $conf \ T \ decomp \ undef \ lev$  by  $(auto \ simp: \ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}decomp)$ 
  moreover have  $set \ M1 \subseteq set \ (trail \ S)$ 
    using  $backtrack.hyps(1)$  by  $auto$ 
  then have  $M: ?M \ T$ 
    using  $marked \ conf \ undef \ confl \ T \ decomp \ lev$ 
    by  $(auto \ simp: \ image\text{-}subset\text{-}iff \ clauses\text{-}def \ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}decomp)$ 
  moreover have  $?U \ T$ 
    using  $learned \ decomp \ conf \ confl \ T \ undef \ lev$  unfolding  $clauses\text{-}def$ 
    by  $(auto \ simp: \ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}decomp)$ 
  moreover have  $?V \ T$ 
    using  $M \ conf \ confl \ trail \ T \ undef \ decomp \ lev$  by  $(force \ simp: \ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}decomp)$ 
  ultimately show  $?case \text{ by } blast$ 
next
  case  $(resolve \ L \ C \ M \ D \ T)$  note  $trail\text{-}S = this(1)$  and  $confl = this(2)$  and  $T = this(4)$ 
  let  $?T = update\text{-}conflicting \ (C\text{-}Clause \ (remdups\text{-}mset \ (D + C))) \ (tl\text{-}trail \ S)$ 
  have  $?C \ ?T$ 
    using  $confl \ trail\text{-}S \ conf \ marked$  by  $simp$ 
  moreover have  $?M \ ?T$ 
    using  $confl \ trail\text{-}S \ conf \ marked$  by  $auto$ 
  moreover have  $?U \ ?T$ 
    using  $trail \text{ learned by } auto$ 

```

moreover have ?V ?T
 using *confl trail-S trail* by auto
 ultimately show ?case using *T* by auto
 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*)

17.4.7 No duplicates all around

This invariant shows that there is no duplicate (no literal appearing twice in the formula). The last part could be proven using the previous invariant moreover.

definition *distinct-cdcl_W-state (S::'st)*
 $\longleftrightarrow ((\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{distinct-mset } T)$
 $\wedge \text{distinct-mset-mset (learned-clss } S)$
 $\wedge \text{distinct-mset-mset (init-clss } S)$
 $\wedge (\forall L \text{ mark. (Propagated } L \text{ mark} \in \text{set (trail } S) \longrightarrow \text{distinct-mset (mark)})))$

lemma *distinct-cdcl_W-state-decomp*:
 assumes *distinct-cdcl_W-state (S::'st)*
 shows $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{distinct-mset } T$
 and *distinct-mset-mset (learned-clss S)*
 and *distinct-mset-mset (init-clss S)*
 and $\forall L \text{ mark. (Propagated } L \text{ mark} \in \text{set (trail } S) \longrightarrow \text{distinct-mset (mark)})$
 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 = C-Clause 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 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
 cdcl_W-M-level-inv S and
 distinct-cdcl_W-state S
 shows *distinct-cdcl_W-state S'*
 using *assms*
proof (*induct rule: cdcl_W-all-induct-lev2*)
 case (*backtrack K i M1 M2 L D*)
 then show ?case
 unfolding *distinct-cdcl_W-state-def*
 by (*fastforce dest: get-all-marked-decomposition-incl simp: cdcl_W-M-level-inv-decomp*)
 next

```

case restart
then show ?case unfolding distinct-cdclW-state-def distinct-mset-set-def clauses-def
using learned-clss-restart-state[of S] by auto
next
case resolve
then show ?case
  by (auto simp add: distinct-cdclW-state-def distinct-mset-set-def clauses-def
    distinct-mset-single-add
    intro!: distinct-mset-union-mset)
qed (auto simp add: distinct-cdclW-state-def distinct-mset-set-def clauses-def)

lemma rtanclp-distinct-cdclW-state-inv:
assumes
  cdclW** S S' and
  cdclW-M-level-inv S and
  distinct-cdclW-state S
shows distinct-cdclW-state S'
using assms apply (induct rule: rtanclp-induct)
using distinct-cdclW-state-inv rtanclp-cdclW-consistent-inv by blast+

```

17.4.8 Conflicts and co

This invariant shows that each mark contains a contradiction only related to the previously defined variable.

abbreviation *every-mark-is-a-conflict :: 'st \Rightarrow bool where*

every-mark-is-a-conflict S \equiv

$\forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \ \text{mark} \ \# \ b = (\text{trail } S)$
 $\longrightarrow (b \models_{as} CNot \ (\ \text{mark} \ - \ \{\#L\# \}) \wedge L \in \# \ \text{mark})$

definition *cdcl_W-conflicting S \equiv*

$(\forall T. \ \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot \ T)$
 $\wedge \text{every-mark-is-a-conflict } S$

lemma *backtrack-atms-of-D-in-M1:*

fixes *M1 :: ('v, nat, 'v clause) marked-lits*

assumes

inv: cdcl_W-M-level-inv S and

undef: undefined-lit M1 L and

i: get-maximum-level D (trail S) = i and

decomp: (Marked K (Suc i) # M1, M2)

$\in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S)) \text{ and}$

S-lvl: backtrack-lvl S = get-maximum-level (D + {#L#}) (trail S) and

S-conf: conflicting S = C-Clause (D + {#L#}) and

undef: undefined-lit M1 L and

T: T \sim (cons-trail (Propagated L (D + {#L#})))

$(\text{reduce-trail-to } M1$

$(\text{add-learned-cls } (D + \{\#L\# \}))$

$(\text{update-backtrack-lvl } i$

$(\text{update-conflicting } C\text{-True } S)))) \text{ and}$

conf: $\forall T. \ \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot \ T$

shows *atms-of D \subseteq atm-of ' lits-of (tl (trail T))*

proof (*rule ccontr*)

let ?*k* = *get-maximum-level (D + {#L#}) (trail S)*

have *trail S $\models_{as} CNot \ D$ using conf S-conf by auto*

then have *vars-of-D: atms-of D \subseteq atm-of ' lits-of (trail S) unfolding atms-of-def*

by (*meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined*)

obtain $M0$ where M : trail $S = M0 @ M2 @ \text{Marked } K (Suc\ i) \# M1$
 using *decomp* by *auto*

have *max*: *get-maximum-level* ($D + \{\#L\# \}$) (trail S)
 = *length* (*get-all-levels-of-marked* ($M0 @ M2 @ \text{Marked } K (Suc\ i) \# M1$))
 using *inv unfolding* *cdcl_W-M-level-inv-def* $S\text{-lvl } M$ by *simp*

assume a : $\neg ?thesis$

then obtain L' where
 L' : $L' \in \text{atms-of } D$ and
 $L'\text{-notin-}M1$: $L' \notin \text{atm-of ' lits-of } M1$
 using T *undef decomp inv* by (*auto simp: cdcl_W-M-level-inv-decomp*)

then have $L'\text{-in}$: $L' \in \text{atm-of ' lits-of } (M0 @ M2 @ \text{Marked } K (i + 1) \# [])$
 using *vars-of-D unfolding* M by *force*

then obtain L'' where
 $L'' \in \# D$ and
 L'' : $L' = \text{atm-of } L''$
 using L' $L'\text{-notin-}M1$ *unfolding* *atms-of-def* by *auto*

have *get-level* L'' (trail S) = *get-rev-level* L'' ($Suc\ i$) ($\text{Marked } K (Suc\ i) \# \text{rev } M2 @ \text{rev } M0$)
 using $L'\text{-notin-}M1\ L''\ M$ by (*auto simp del: get-rev-level.simps*)

have *get-all-levels-of-marked* (trail S) = *rev* [$1..<1+?k$]
 using *inv S-lvl unfolding* *cdcl_W-M-level-inv-def* by *auto*

then have *get-all-levels-of-marked* ($M0 @ M2$)
 = *rev* [$Suc\ (Suc\ i)..<Suc\ (\text{get-maximum-level } (D + \{\#L\# \}) (trail\ S))$]
unfolding M by (*auto simp: rev-swap[symmetric] dest!: append-cons-eq-upt-length-i-end*)

then have M : *get-all-levels-of-marked* $M0 @ \text{get-all-levels-of-marked } M2$
 = *rev* [$Suc\ (Suc\ i)..<Suc\ (\text{length } (\text{get-all-levels-of-marked } (M0 @ M2 @ \text{Marked } K (Suc\ i) \# M1)))$]
unfolding *max* *unfolding* M by *simp*

have *get-rev-level* L'' ($Suc\ i$) ($\text{Marked } K (Suc\ i) \# \text{rev } (M0 @ M2)$)
 $\geq \text{Min } (\text{set } ((Suc\ i) \# \text{get-all-levels-of-marked } (\text{Marked } K (Suc\ i) \# \text{rev } (M0 @ M2))))$
 using *get-rev-level-ge-min-get-all-levels-of-marked*[*of* L''
 $\text{rev } (M0 @ M2 @ [\text{Marked } K (Suc\ i)])\ Suc\ i]\ L'\text{-in}$
unfolding L'' by (*fastforce simp add: lits-of-def*)

also have $\text{Min } (\text{set } ((Suc\ i) \# \text{get-all-levels-of-marked } (\text{Marked } K (Suc\ i) \# \text{rev } (M0 @ M2))))$
 = $\text{Min } (\text{set } ((Suc\ i) \# \text{get-all-levels-of-marked } (\text{rev } (M0 @ M2))))$ by *auto*

also have $\dots = \text{Min } (\text{set } ((Suc\ i) \# \text{get-all-levels-of-marked } M0 @ \text{get-all-levels-of-marked } M2))$
 by (*simp add: Un-commute*)

also have $\dots = \text{Min } (\text{set } ((Suc\ i) \# [Suc\ (Suc\ i)..<2 + \text{length } (\text{get-all-levels-of-marked } M0)$
 $+ (\text{length } (\text{get-all-levels-of-marked } M2) + \text{length } (\text{get-all-levels-of-marked } M1))]))$
unfolding M by (*auto simp add: Un-commute*)

also have $\dots = Suc\ i$ by (*auto intro: Min-eqI*)

finally have *get-rev-level* L'' ($Suc\ i$) ($\text{Marked } K (Suc\ i) \# \text{rev } (M0 @ M2)$) $\geq Suc\ i$.

then have *get-level* L'' (trail S) $\geq i + 1$
 using $\langle \text{get-level } L'' (trail\ S) = \text{get-rev-level } L'' (Suc\ i) (\text{Marked } K (Suc\ i) \# \text{rev } M2 @ \text{rev } M0) \rangle$
 by *simp*

then have *get-maximum-level* D (trail S) $\geq i + 1$
 using *get-maximum-level-ge-get-level*[*OF* $\langle L'' \in \# D \rangle$, *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 $a2: \text{atms-of } D \subseteq \text{atm-of ' lits-of } M'$
shows $\forall x \in \text{atms-of } D. x \notin \text{atm-of ' lits-of } M$
proof –
 { **fix** $aa :: 'a$
 have $ff1: \bigwedge l \text{ ms. undefined-lit ms } l \vee \text{atm-of } l$
 $\in \text{set (map (\lambda m. atm-of (lit-of (m::('a, 'b, 'c) marked-lit))) ms)$
 by (*simp add: defined-lit-map*)
 have $ff2: \bigwedge a. a \notin \text{atms-of } D \vee a \in \text{atm-of ' lits-of } M'$
 using $a2$ **by** (*meson subsetCE*)
 have $ff3: \bigwedge a. a \notin \text{set (map (\lambda m. atm-of (lit-of m)) } M')$
 $\vee a \notin \text{set (map (\lambda m. atm-of (lit-of m)) } M)$
 using $a1$ **by** (*metis (lifting) IntI distinct-append empty-iff map-append*)
 have $\forall L \text{ a f. } \exists l. ((a::'a) \notin f \text{ ' } L \vee (l::'a \text{ literal}) \in L) \wedge (a \notin f \text{ ' } L \vee f \text{ l} = a)$
 by *blast*
 then have $aa \notin \text{atms-of } D \vee aa \in \text{atm-of ' lits-of } M$
 using $ff3 \text{ ff2 } ff1$ **by** (*metis (no-types) Marked-Propagated-in-iff-in-lits-of*) }
then show *?thesis*
 by *blast*
qed

lemma *cdcl_W-propagate-is-conclusion:*

assumes
 $cdcl_W \text{ } S \text{ } S'$ **and**
 $inv: cdcl_W\text{-}M\text{-level-inv } S$ **and**
 $decomp: \text{all-decomposition-implies-}m \text{ (init-clss } S) \text{ (get-all-marked-decomposition (trail } S))$ **and**
 $learned: cdcl_W\text{-}learned\text{-clause } S$ **and**
 $confl: \forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot \text{ } T$ **and**
 $alien: \text{no-strange-atm } S$
shows $\text{all-decomposition-implies-}m \text{ (init-clss } S') \text{ (get-all-marked-decomposition (trail } S'))$
using *assms(1,2)*
proof (*induct rule: cdcl_W-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* $L \text{ } C \text{ } M \text{ } D$) **note** $tr = \text{this}(1)$ **and** $T = \text{this}(4)$
 let $?decomp = \text{get-all-marked-decomposition } M$
 have $M: \text{set } ?decomp = \text{insert (hd } ?decomp) (\text{set (tl } ?decomp))$
 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* $L \text{ } C' \text{ } M \text{ } D$) **note** $tr = \text{this}(1)$ **and** $T = \text{this}(5)$
 have $M: \text{set (get-all-marked-decomposition } M)$
 $= \text{insert (hd (get-all-marked-decomposition } M)) (\text{set (tl (get-all-marked-decomposition } M))}$
 by (*cases get-all-marked-decomposition M*) *auto*
 show *?case*
 using *decomp tr T* **unfolding** *all-decomposition-implies-def*

```

    by (cases hd (get-all-marked-decomposition M))
      (auto simp add: M)
next
case decide note S = this(1) and undef = this(2) and T = this(4)
show ?case using decomp T undef unfolding S all-decomposition-implies-def by auto
next
case (propagate C L T) note propa = this(2) and undef = this(3) and T = this(5)
obtain a y where ay: hd (get-all-marked-decomposition (trail S)) = (a, y)
  by (cases hd (get-all-marked-decomposition (trail S)))
then have M: trail S = y @ a using get-all-marked-decomposition-decomp by blast
have M': set (get-all-marked-decomposition (trail S))
  = insert (a, y) (set (tl (get-all-marked-decomposition (trail S))))
  using ay by (cases get-all-marked-decomposition (trail S)) auto
have (λa. {#lit-of a#}) ' set a ∪ set-mset (init-clss S) ⊨ps (λa. {#lit-of a#}) ' set y
  using decomp ay unfolding all-decomposition-implies-def
  by (cases get-all-marked-decomposition (trail S)) fastforce+
then have a-Un-N-M: (λa. {#lit-of a#}) ' set a ∪ set-mset (init-clss S)
  ⊨ps (λa. {#lit-of a#}) ' set (trail S)
  unfolding M by (auto simp add: all-in-true-clss-clss image-Un)

have (λa. {#lit-of a#}) ' set a ∪ set-mset (init-clss S) ⊨p {#L#} (is ?I ⊨p -)
proof (rule true-clss-clss-plus-CNot)
  show ?I ⊨p C + {#L#}
  using propa propagate.premis learned confl unfolding M
  by (metis Un-iff cdclw-learned-clause-def clauses-def mem-set-mset-iff propagate.hyps(1)
    set-mset-union true-clss-clss-in-imp-true-clss-clss true-clss-clss-mono-l2
    union-trus-clss-clss)
next
have (λm. {#lit-of m#}) ' set (trail S) ⊨ps CNot C
  using ⟨(trail S) ⊨as CNot C⟩ true-annots-true-clss-clss by blast
then show ?I ⊨ps CNot C
  using a-Un-N-M true-clss-clss-left-right true-clss-clss-union-l-r by blast
qed
moreover have ∧aa b.
  ∀ (Ls, seen) ∈ set (get-all-marked-decomposition (y @ a)).
    (λa. {#lit-of a#}) ' set Ls ∪ set-mset (init-clss S) ⊨ps (λa. {#lit-of a#}) ' set seen
  ⇒ (aa, b) ∈ set (tl (get-all-marked-decomposition (y @ a)))
  ⇒ (λa. {#lit-of a#}) ' set aa ∪ set-mset (init-clss S) ⊨ps (λa. {#lit-of a#}) ' set b
  by (metis (no-types, lifting) case-prod-conv get-all-marked-decomposition-never-empty-sym
    list.collapse list.set-intros(2))

ultimately show ?case
  using decomp T undef unfolding ay all-decomposition-implies-def
  using M ⟨(λa. {#lit-of a#}) ' set a ∪ set-mset (init-clss S) ⊨ps (λa. {#lit-of a#}) ' set y⟩
  ay by auto
next
case (backtrack K i M1 M2 L D T) note decomp' = this(1) and lev-L = this(2) and conf = this(3)
and
  undef = this(6) and T = this(7)
have ∀ l ∈ set M2. ¬is-marked l
  using get-all-marked-decomposition-snd-not-marked backtrack.hyps(1) by blast
obtain M0 where M: trail S = M0 @ M2 @ Marked K (i + 1) # M1
  using decomp' by auto
show ?case unfolding all-decomposition-implies-def
proof

```



```

fix x
assume x ∈ set (get-all-marked-decomposition (trail T))
then have x: x ∈ set (get-all-marked-decomposition (Propagated L ((D + {#L#})) # M1))
  using T decomp' undef inv by (simp add: cdclW-M-level-inv-decomp)
let ?m = get-all-marked-decomposition (Propagated L ((D + {#L#})) # M1)
let ?hd = hd ?m
let ?tl = tl ?m
have x = ?hd ∨ x ∈ set ?tl
  using x by (case-tac ?m) auto
moreover {
  assume x ∈ set ?tl
  then have x ∈ set (get-all-marked-decomposition (trail S))
    using tl-get-all-marked-decomposition-skip-some[of x] by (simp add: list.set-sel(2) M)
  then have case x of (Ls, seen) ⇒ (λa. {#lit-of a#}) ' set Ls
    ∪ set-mset (init-clss (T))
    ⊨ps (λa. {#lit-of a#}) ' set seen
    using decomp learned decomp confl alien inv T undef M
    unfolding all-decomposition-implies-def cdclW-M-level-inv-def
    by auto
}
moreover {
  assume x = ?hd
  obtain M1' M1'' where M1: hd (get-all-marked-decomposition M1) = (M1', M1'')
    by (cases hd (get-all-marked-decomposition M1))
  then have x': x = (M1', Propagated L ( (D + {#L#})) # 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: (λa. {#lit-of a#}) ' set M1' ∪ set-mset (init-clss S)
    ⊨ps (λa. {#lit-of a#}) ' set M1''
    using decomp unfolding all-decomposition-implies-def by auto
  moreover
    have trail S ⊨as CNot D using conf confl by auto
    then have vars-of-D: atms-of D ⊆ atm-of ' lits-of (trail S)
      unfolding atms-of-def
      by (meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)
    have vars-of-D: atms-of D ⊆ atm-of ' lits-of M1
      using backtrack-atms-of-D-in-M1[of S M1 L D i K M2 T] backtrack inv conf confl
      by (auto simp: cdclW-M-level-inv-decomp)
    have no-dup (trail S) using inv by (auto simp: cdclW-M-level-inv-decomp)
    then have vars-in-M1:
      ∀ x ∈ atms-of D. x ∉ atm-of ' lits-of (M0 @ M2 @ Marked K (i + 1) # [])
      using vars-of-D distinct-atms-of-incl-not-in-other[of M0 @ M2 @ Marked K (i + 1) # []
        M1]
      unfolding M by auto
    have M1 ⊨as CNot D
      using vars-in-M1 true-annots-remove-if-notin-vars[of M0 @ M2 @ Marked K (i + 1) # []
        M1 CNot D] ⟨trail S ⊨as CNot D⟩ unfolding M lits-of-def by simp
    have M1 = M1'' @ M1' by (simp add: M1 get-all-marked-decomposition-decomp)
    have TT: (λa. {#lit-of a#}) ' set M1' ∪ set-mset (init-clss S) ⊨ps CNot D
      using true-annots-true-clss-clc[OF ⟨M1 ⊨as CNot D⟩ true-clss-clss-left-right[OF 1,
        of CNot D] unfolding ⟨M1 = M1'' @ M1'⟩ by (auto simp add: inf-sup-aci(5,7))
    have init-clss S ⊨pm D + {#L#}
      using conf learned cdclW-learned-clause-def confl by blast

```

```

    then have  $T'$ :  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } M1' \cup \text{set-mset (init-clss } S) \models_p D + \{\#L\# \}$  by auto
    have  $\text{atms-of } (D + \{\#L\# \}) \subseteq \text{atms-of-msu (clauses } S)$ 
    using alien conf unfolding no-strange-atm-def clauses-def by auto
    then have  $(\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } M1' \cup \text{set-mset (init-clss } S) \models_p \{\#L\# \}$ 
    using true-clss-cls-plus-CNot[OF T' TT] by auto
  ultimately
    have case  $x$  of  $(Ls, \text{seen}) \Rightarrow (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } Ls$ 
       $\cup \text{set-mset (init-clss } T)$ 
       $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set seen}$  using  $T' T \text{ decomp' undef inv}$  unfolding  $x'$ 
      by  $(\text{simp add: cdcl}_W\text{-M-level-inv-decomp})$ 
    }
  ultimately show case  $x$  of  $(Ls, \text{seen}) \Rightarrow (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set } Ls \cup \text{set-mset (init-clss } T)$ 
     $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' set seen}$  using  $T$  by auto
qed
qed

lemma cdclW-propagate-is-false:
  assumes
     $\text{cdcl}_W S S'$  and
     $\text{lev: cdcl}_W\text{-M-level-inv } S$  and
     $\text{learned: cdcl}_W\text{-learned-clause } S$  and
     $\text{decomp: all-decomposition-implies-m (init-clss } S) (\text{get-all-marked-decomposition (trail } S))$  and
     $\text{confl: } \forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot T$  and
     $\text{alien: no-strange-atm } S$  and
     $\text{mark-confl: every-mark-is-a-conflict } S$ 
  shows  $\text{every-mark-is-a-conflict } S'$ 
  using assms(1,2)
proof (induct rule:  $\text{cdcl}_W\text{-all-induct-lev2}$ )
  case  $(\text{propagate } C L T)$  note  $\text{undef} = \text{this(3)}$  and  $T = \text{this(5)}$ 
  show ?case
  proof (intro allI impI)
    fix  $L'$  mark  $a$   $b$ 
    assume  $a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } T$ 
    then have  $(a = [] \wedge L = L' \wedge \text{mark} = C + \{\#L\# \} \wedge b = \text{trail } S)$ 
       $\vee \text{tl } a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } S$ 
    using  $T \text{ undef}$  by  $(\text{cases } a) \text{ fastforce+}$ 
    moreover {
      assume  $\text{tl } a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } S$ 
      then have  $b \models_{as} CNot (\text{mark} - \{\#L'\# \}) \wedge L' \in \# \text{ mark}$ 
      using mark-confl by auto
    }
    moreover {
      assume  $a = []$  and  $L = L'$  and  $\text{mark} = C + \{\#L\# \}$  and  $b = \text{trail } S$ 
      then have  $b \models_{as} CNot (\text{mark} - \{\#L\# \}) \wedge L \in \# \text{ mark}$ 
      using  $\langle \text{trail } S \models_{as} CNot C \rangle$  by auto
    }
  }
  ultimately show  $b \models_{as} CNot (\text{mark} - \{\#L'\# \}) \wedge L' \in \# \text{ mark}$  by blast
qed

next
  case  $(\text{decide } L)$  note  $\text{undef[simp]} = \text{this(2)}$  and  $T = \text{this(4)}$ 
  have  $\bigwedge a L a \text{ mark } b. a @ \text{Propagated } L a \text{ mark } \# b = \text{Marked } L (\text{backtrack-lvl } S + 1) \# \text{trail } S$ 
     $\implies \text{tl } a @ \text{Propagated } L a \text{ mark } \# b = \text{trail } S$  by  $(\text{case-tac } a, \text{auto})$ 
  then show ?case using mark-confl T unfolding decide.hyps(1) by fastforce
next
  case  $(\text{skip } L C' M D T)$  note  $\text{tr} = \text{this(1)}$  and  $T = \text{this(5)}$ 

```

```

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  $\forall La \text{ mark } a b. a @ \text{Propagated } La \text{ mark } \# b = \text{Propagated } L C' \# M$ 
     $\longrightarrow b \models_{as} CNot (mark - \{\#La\}) \wedge La \in \# mark$ 
    using mark-confl unfolding skip.hyps(1) by simp
  ultimately show  $b \models_{as} CNot (mark - \{\#L'\}) \wedge L' \in \# mark$  by blast
qed
next
case (conflict D)
then show ?case using mark-confl by simp
next
case (resolve L C M D T) note tr-S = this(1) and T = this(4)
show ?case unfolding resolve.hyps(1)
proof (intro allI impI)
  fix L' mark a b
  assume a @ Propagated L' mark # b = trail T
  then have Propagated L ( (C + {\#L\#})) # M
    = (Propagated L ( (C + {\#L\#})) # a) @ Propagated L' mark # b
    using T tr-S by auto
  then show  $b \models_{as} CNot (mark - \{\#L'\}) \wedge L' \in \# mark$ 
    using mark-confl unfolding resolve.hyps(1) by presburger
qed
next
case restart
then show ?case by auto
next
case forget
then show ?case using mark-confl by auto
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and conf = this(3) and undef = this(6)
and
  T = this(7)
have  $\forall l \in \text{set } M2. \neg \text{is-marked } l$ 
  using get-all-marked-decomposition-snd-not-marked backtrack.hyps(1) by blast
obtain M0 where M: trail S = M0 @ M2 @ Marked K (i + 1) # M1
  using backtrack.hyps(1) by auto
have [simp]: trail (reduce-trail-to M1 (add-learned-cls (D + {\#L\#}))
  (update-backtrack-lvl i (update-conflicting C-True S))) = M1
  using decomp lev by (auto simp: cdclW-M-level-inv-decomp)
show ?case
proof (intro allI impI)
  fix La mark a b
  assume a @ Propagated La mark # b = trail T
  then have (a = []  $\wedge$  Propagated La mark = Propagated L (D + {\#L\#}))  $\wedge$  b = M1
     $\vee$  tl a @ Propagated La mark # b = M1
    using M T decomp undef by (cases a) (auto)
  moreover {
    assume A: a = [] and
      P: Propagated La mark = Propagated L ( (D + {\#L\#})) and
      b: b = M1
    have trail S  $\models_{as} CNot D$  using conf confl by auto
  }

```

```

then have vars-of-D: atms-of D  $\subseteq$  atm-of ‘ lits-of (trail S)
  unfolding atms-of-def
  by (meson image-subsetI mem-set-mset-iff true-annots-CNot-all-atms-defined)
have vars-of-D: atms-of D  $\subseteq$  atm-of ‘ lits-of M1
  using backtrack-atms-of-D-in-M1[of S M1 L D i K M2 T] T backtrack lev confl by auto
have no-dup (trail S) using lev by (auto simp: cdclW-M-level-inv-decomp)
then have vars-in-M1:  $\forall x \in \text{atms-of } D. x \notin$ 
  atm-of ‘ lits-of (M0 @ M2 @ Marked K (i + 1) # [])
  using vars-of-D distinct-atms-of-incl-not-in-other[of M0 @ M2 @ Marked K (i + 1) # [] M1]
  unfolding M by auto
have M1  $\models_{as}$  CNot D
  using vars-in-M1 true-annots-remove-if-notin-vars[of M0 @ M2 @ Marked K (i + 1) # [] M1
    CNot D] (trail S  $\models_{as}$  CNot D) unfolding M lits-of-def by simp
then have b  $\models_{as}$  CNot (mark - {#La#})  $\wedge$  La  $\in \#$  mark
  using P b by auto
}
moreover {
  assume tl a @ Propagated La mark # b = M1
  then obtain c' where c' @ Propagated La mark # b = trail S unfolding M by auto
  then have b  $\models_{as}$  CNot (mark - {#La#})  $\wedge$  La  $\in \#$  mark
    using mark-confl by blast
}
ultimately show b  $\models_{as}$  CNot (mark - {#La#})  $\wedge$  La  $\in \#$  mark by fast
qed
qed

```

lemma cdcl_W-conflicting-is-false:

```

assumes
  cdclW S S' and
  M-lev: cdclW-M-level-inv S and
  confl-inv:  $\forall T. \text{conflicting } S = C\text{-Clause } T \longrightarrow \text{trail } S \models_{as} CNot \ T$  and
  marked-confl:  $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S) \longrightarrow (b \models_{as} CNot \ (\text{mark} - \{ \#L \# \}) \wedge L \in \# \ \text{mark})$  and
  dist: distinct-cdclW-state S
shows  $\forall T. \text{conflicting } S' = C\text{-Clause } T \longrightarrow \text{trail } S' \models_{as} CNot \ T$ 
using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
case (skip L C' M D) note tr-S = this(1) and T = this(5)
then have Propagated L C' # M  $\models_{as}$  CNot D using assms skip by auto
moreover
  have L  $\notin \#$  D
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    then have - L  $\in$  lits-of M
    using in-CNot-implies-uminus(2)[of D L Propagated L C' # M]
    (Propagated L C' # M  $\models_{as}$  CNot D) by simp
    then show False
    by (metis M-lev cdclW-M-level-inv-decomp(1) consistent-interp-def insert-iff
      lits-of-cons marked-lit.sel(2) skip.hyps(1))
  qed
ultimately show ?case
using skip.hyps(1-3) true-annots-CNot-lit-of-notin-skip T unfolding cdclW-M-level-inv-def
by fastforce
next
case (resolve L C M D T) note tr = this(1) and confl = this(2) and T = this(4)

```

```

show ?case
proof (intro allI impI)
  fix T'
  have tl (trail S)  $\models_{as}$  CNot C using tr assms(4) by fastforce
  moreover
    have distinct-mset (D + {#- L#}) using confl dist
      unfolding distinct-cdclW-state-def by auto
    then have -L  $\notin$  # D unfolding distinct-mset-def by auto
    have M  $\models_{as}$  CNot D
    proof -
      have Propagated L ( (C + {#L#})) # M  $\models_{as}$  CNot D  $\cup$  CNot {#- L#}
        using confl tr confl-inv by force
      then show ?thesis
        using M-lev  $\langle - L \notin$  # D  $\rangle$  tr true-annots-lit-of-notin-skip
        unfolding cdclW-M-level-inv-def by force
    qed
  moreover assume conflicting T = C-Clause T'
  ultimately
    show trail T  $\models_{as}$  CNot T'
    using tr T by auto
  qed
qed (auto simp: assms(2) cdclW-M-level-inv-decomp)

```

lemma *cdcl_W-conflicting-decomp*:

```

assumes cdclW-conflicting S
shows  $\forall T. \text{conflicting } S = \text{C-Clause } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$ 
and  $\forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$ 
 $\longrightarrow (b \models_{as} \text{CNot } (\text{mark} - \{ \#L\# \}) \wedge L \in \# \text{ mark})$ 
using assms unfolding cdclW-conflicting-def by blast+

```

lemma *cdcl_W-conflicting-decomp2*:

```

assumes cdclW-conflicting S and conflicting S = C-Clause T
shows trail S  $\models_{as}$  CNot T
using assms unfolding cdclW-conflicting-def by blast+

```

lemma *cdcl_W-conflicting-decomp2'*:

```

assumes
  cdclW-conflicting S and
  conflicting S = C-Clause D
shows trail S  $\models_{as}$  CNot D
using assms unfolding cdclW-conflicting-def by auto

```

lemma *cdcl_W-conflicting-S0-cdcl_W[simp]*:

```

cdclW-conflicting (init-state N)
unfolding cdclW-conflicting-def by auto

```

17.4.9 Putting all the invariants together

lemma *cdcl_W-all-inv*:

```

assumes cdclW: cdclW S S' and
1: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
2: cdclW-learned-clause S and
4: cdclW-M-level-inv S and
5: no-strange-atm S and
7: distinct-cdclW-state S and
8: cdclW-conflicting S

```

```

shows all-decomposition-implies-m (init-clss  $S'$ ) (get-all-marked-decomposition (trail  $S'$ ))
and cdclW-learned-clause  $S'$ 
and cdclW-M-level-inv  $S'$ 
and no-strange-atm  $S'$ 
and distinct-cdclW-state  $S'$ 
and cdclW-conflicting  $S'$ 
proof –
show  $S1$ : all-decomposition-implies-m (init-clss  $S'$ ) (get-all-marked-decomposition (trail  $S'$ ))
  using cdclW-propagate-is-conclusion[OF cdclW 4 1 2 - 5] 8 unfolding cdclW-conflicting-def
  by blast
show  $S2$ : cdclW-learned-clause  $S'$  using cdclW-learned-clss[OF cdclW 2 4] .
show  $S4$ : cdclW-M-level-inv  $S'$  using cdclW-consistent-inv[OF cdclW 4] .
show  $S5$ : no-strange-atm  $S'$  using cdclW-no-strange-atm-inv[OF cdclW 5 4] .
show  $S7$ : distinct-cdclW-state  $S'$  using distinct-cdclW-state-inv[OF cdclW 4 7] .
show  $S8$ : cdclW-conflicting  $S'$ 
  using cdclW-conflicting-is-false[OF cdclW 4 - - 7] 8 cdclW-propagate-is-false[OF cdclW 4 2 1 -
    5]
  unfolding cdclW-conflicting-def by fast
qed

```

lemma *rtranclp-cdcl_W-all-inv*:

```

assumes
  cdclW: rtranclp cdclW S S' and
  1: all-decomposition-implies-m (init-clss  $S$ ) (get-all-marked-decomposition (trail  $S$ )) and
  2: cdclW-learned-clause  $S$  and
  4: cdclW-M-level-inv  $S$  and
  5: no-strange-atm  $S$  and
  7: distinct-cdclW-state  $S$  and
  8: cdclW-conflicting  $S$ 
shows
  all-decomposition-implies-m (init-clss  $S'$ ) (get-all-marked-decomposition (trail  $S'$ )) and
  cdclW-learned-clause  $S'$  and
  cdclW-M-level-inv  $S'$  and
  no-strange-atm  $S'$  and
  distinct-cdclW-state  $S'$  and
  cdclW-conflicting  $S'$ 
using assms
proof (induct rule: rtranclp-induct)
case base
  case 1 then show ?case by blast
  case 2 then show ?case by blast
  case 3 then show ?case by blast
  case 4 then show ?case by blast
  case 5 then show ?case by blast
  case 6 then show ?case by blast
next
case (step  $S' S''$ ) note  $H = \text{this}$ 
  case 1 with  $H(3-7)$ [OF this(1-6)] show ?case using cdclW-all-inv[OF  $H(2)$ ]
     $H$  by presburger
  case 2 with  $H(3-7)$ [OF this(1-6)] show ?case using cdclW-all-inv[OF  $H(2)$ ]
     $H$  by presburger
  case 3 with  $H(3-7)$ [OF this(1-6)] show ?case using cdclW-all-inv[OF  $H(2)$ ]
     $H$  by presburger
  case 4 with  $H(3-7)$ [OF this(1-6)] show ?case using cdclW-all-inv[OF  $H(2)$ ]
     $H$  by presburger

```

case 5 with $H(3-7)[OF \text{ this}(1-6)]$ **show** $?case$ **using** $cdcl_W\text{-all-inv}[OF H(2)]$
 H **by** *presburger*
case 6 with $H(3-7)[OF \text{ this}(1-6)]$ **show** $?case$ **using** $cdcl_W\text{-all-inv}[OF H(2)]$
 H **by** *presburger*
qed

lemma *all-invariant-S0-cdcl_W*:
assumes *distinct-mset-mset* N
shows *all-decomposition-implies-m* (*init-clss* (*init-state* N))
 $(\text{get-all-marked-decomposition } (\text{trail } (\text{init-state } N)))$
and *cdcl_W-learned-clause* (*init-state* N)
and $\forall T. \text{conflicting } (\text{init-state } N) = C\text{-Clause } T \longrightarrow (\text{trail } (\text{init-state } N)) \models_{as} C\text{Not } T$
and *no-strange-atm* (*init-state* N)
and *consistent-interp* (*lits-of* (*trail* (*init-state* N)))
and $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = \text{trail } (\text{init-state } N) \longrightarrow$
 $(b \models_{as} C\text{Not } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$
and *distinct-cdcl_W-state* (*init-state* N)
using *assms* **by** *auto*

lemma *cdcl_W-only-propagated-vars-unsat*:
assumes
 $\text{marked}: \forall x \in \text{set } M. \neg \text{is-marked } x$ **and**
 $DN: D \in \# \text{ clauses } S$ **and**
 $D: M \models_{as} C\text{Not } D$ **and**
 $\text{inv}: \text{all-decomposition-implies-m } N (\text{get-all-marked-decomposition } M)$ **and**
 $\text{state}: \text{state } S = (M, N, U, k, C)$ **and**
 $\text{learned-cl}: \text{cdcl}_W\text{-learned-clause } S$ **and**
 $\text{atm-incl}: \text{no-strange-atm } S$
shows *unsatisfiable* (*set-mset* N)
proof (*rule ccontr*)
assume $\neg \text{unsatisfiable } (\text{set-mset } N)$
then obtain I **where**
 $I: I \models_s \text{set-mset } N$ **and**
 $\text{cons}: \text{consistent-interp } I$ **and**
 $\text{tot}: \text{total-over-m } I (\text{set-mset } N)$
unfolding *satisfiable-def* **by** *auto*
have $\text{atms-of-msu } N \cup \text{atms-of-msu } U = \text{atms-of-msu } N$
using $\text{atm-incl state unfolding total-over-m-def no-strange-atm-def}$
by (*auto simp add: clauses-def*)
then have $\text{total-over-m } I (\text{set-mset } N)$ **using** $\text{tot unfolding total-over-m-def}$ **by** *auto*
moreover have $N \models_{psm} U$ **using** $\text{learned-cl state unfolding cdcl}_W\text{-learned-clause-def}$ **by** *auto*
ultimately have $I \models D: I \models D$
using $I DN \text{ cons state unfolding true-clss-clss-def true-clss-def Ball-def}$
by (*metis Un-iff $\langle \text{atms-of-msu } N \cup \text{atms-of-msu } U = \text{atms-of-msu } N \rangle \text{atms-of-ms-union clauses-def mem-set-mset-iff prod.inject set-mset-union total-over-m-def}$*)

have $l0: \{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} = \{\}$ **using** marked **by** *auto*
have $\text{atms-of-ms } (\text{set-mset } N \cup (\lambda a. \{\#lit\text{-of } a\# \})) \text{ 'set } M = \text{atms-of-msu } N$
using $\text{atm-incl state unfolding no-strange-atm-def}$ **by** *auto*
then have $\text{total-over-m } I (\text{set-mset } N \cup (\lambda a. \{\#lit\text{-of } a\# \})) \text{ ' (set } M)$
using $\text{tot unfolding total-over-m-def}$ **by** *auto*
then have $I \models_s (\lambda a. \{\#lit\text{-of } a\# \}) \text{ ' (set } M)$
using $\text{all-decomposition-implies-propagated-lits-are-implied}[OF \text{ inv}] \text{ cons } I$
unfolding $\text{true-clss-clss-def } l0$ **by** *auto*

```

then have IM:  $I \models_s (\lambda a. \{\#lit\text{-of } a\# \})$  ‘ set M by auto
{
  fix K
  assume  $K \in \# D$ 
  then have  $-K \in lits\text{-of } M$ 
    using D unfolding true-annots-def Ball-def CNot-def true-annot-def true-clss-def true-lit-def
    Bex-mset-def by (metis (mono-tags, lifting) count-single less-not-refl mem-Collect-eq)
  then have  $-K \in I$  using IM true-clss-singleton-lit-of-implies-incl lits-of-def by fastforce
}
then have  $\neg I \models D$  using cons unfolding true-clss-def true-lit-def consistent-interp-def by auto
then show False using I-D by blast
qed

```

We have actually a much stronger theorem, namely *all-decomposition-implies ?N (get-all-marked-decomposition ?M) \implies ?N $\cup \{\{\#lit\text{-of } L\# \} \mid L. is\text{-marked } L \wedge L \in set \text{ ?M} \}$ $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$ ‘ set ?M*, that show that the only choices we made are marked in the formula

```

lemma
  assumes all-decomposition-implies-m N (get-all-marked-decomposition M)
  and  $\forall m \in set M. \neg is\text{-marked } m$ 
  shows set-mset N  $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \})$  ‘ set M
proof -
  have T:  $\{\{\#lit\text{-of } L\# \} \mid L. is\text{-marked } L \wedge L \in set M\} = \{\}$  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
  cdclW: cdclW S S' and
  lev: cdclW-M-level-inv S and
  [simp]: conflicting S' = C-Clause {#} and
  learned: cdclW-learned-clause S
shows unsatisfiable (set-mset (init-clss S))
using assms
proof -
  have cdclW-learned-clause S' using cdclW-learned-clss cdclW learned lev by auto
  then have init-clss S'  $\models_{pm} \{\# \}$  using assms(3) unfolding cdclW-learned-clause-def by auto
  then have init-clss S  $\models_{pm} \{\# \}$ 
    using cdclW-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 cdclW S S'
and conflicting S = C-Clause {#}
shows False
using assms by (induct rule: cdclW-all-induct) auto

```

17.4.10 No tautology is learned

This is a simple consequence of all we have shown previously. It is not strictly necessary, but helps finding a better bound on the number of learned clauses.

lemma *learned-clss-are-not-tautologies:*

assumes
cdcl_W *S S'* **and**
lev: *cdcl_W-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*) **note** *confl = this(3)*
have *consistent-interp* (*lits-of* (*trail S*)) **using** *lev* **by** (*auto simp*: *cdcl_W-M-level-inv-decomp*)
moreover
have *trail S* \models_{as} *CNot* (*D* + {*#L#*})
using *conflicting confl unfolding cdcl_W-conflicting-def* **by** *auto*
then have *lits-of* (*trail S*) \models_s *CNot* (*D* + {*#L#*}) **using** *true-annots-true-cls* **by** *blast*
ultimately have $\neg \text{tautology } (D + \{\#L\# \})$ **using** *consistent-CNot-not-tautology* **by** *blast*
then show ?*case* **using** *backtrack no-tauto*
by (*auto simp*: *cdcl_W-M-level-inv-decomp split: split-if-asm*)
next
case *restart*
then show ?*case* **using** *learned-clss-restart-state state-eq-learned-clss no-tauto*
by (*metis* (*no-types, lifting*) *ball-msetE ball-msetI mem-set-mset-iff set-mset-mono subsetCE*)
qed *auto*

definition *final-cdcl_W-state* (*S*:: 'st)
 \longleftrightarrow (*trail S* \models_{asm} *init-clss S*
 $\vee ((\forall L \in \text{set } (\text{trail } S). \neg \text{is-marked } L) \wedge$
 $(\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} \text{CNot } C)))$

definition *termination-cdcl_W-state* (*S*:: 'st)
 \longleftrightarrow (*trail S* \models_{asm} *init-clss S*
 $\vee ((\forall L \in \text{atms-of-msu } (\text{init-clss } S). L \in \text{atm-of 'lits-of } (\text{trail } S))$
 $\wedge (\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} \text{CNot } C)))$

17.5 CDCL Strong Completeness

fun *mapi* :: ('a \Rightarrow nat \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a list \Rightarrow 'b list **where**
mapi - - [] = [] |
mapi *f* *n* (*x* # *xs*) = *f* *x* *n* # *mapi* *f* (*n* - 1) *xs*

lemma *mark-not-in-set-mapi*[*simp*]: $L \notin \text{set } M \Longrightarrow \text{Marked } L \ k \notin \text{set } (\text{mapi } \text{Marked } i \ M)$
by (*induct* *M arbitrary: i*) *auto*

lemma *propagated-not-in-set-mapi*[*simp*]: $L \notin \text{set } M \Longrightarrow \text{Propagated } L \ k \notin \text{set } (\text{mapi } \text{Marked } i \ M)$
by (*induct* *M arbitrary: i*) *auto*

lemma *image-set-mapi*:
 $f \text{ 'set } (\text{mapi } g \ i \ M) = \text{set } (\text{mapi } (\lambda x \ i. f \ (g \ x \ i)) \ i \ M)$
by (*induction* *M arbitrary: i*) *auto*

lemma *mapi-map-convert*:
 $\forall x \ i \ j. f \ x \ i = f \ x \ j \Longrightarrow \text{mapi } f \ i \ M = \text{map } (\lambda x. f \ x \ 0) \ M$
by (*induction* *M arbitrary: i*) *auto*

lemma *defined-lit-mapi*: *defined-lit* (*mapi* *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**
atm-of ‘ (set *M*) \subseteq *atms-of-msu* *N*
shows $\exists S. \text{rtrancpl } \text{cdcl}_W \text{ (init-state } N) S$
 $\wedge \text{state } S = (\text{mapi } \text{Marked } (\text{length } M) M, N, \{\#\}, \text{length } M, C\text{-True})$
using *assms*
proof (*induct* *M*)
case *Nil*
then show ?*case* **by** *auto*
next
case (*Cons* *L* *M*) **note** *IH* = *this*(1)
have *consistent-interp* (set *M*) **and** *distinct* *M* **and** *atm-of* ‘ set *M* \subseteq *atms-of-msu* *N*
using *Cons.prem*s(1–3) **unfolding** *consistent-interp-def* **by** *auto*
then obtain *S* **where**
st: *cdcl_W*^{**} (init-state *N*) *S* **and**
S: state *S* = (*mapi* *Marked* (length *M*) *M*, *N*, {#}, length *M*, *C-True*)
using *IH* **by** *auto*
let ?*S*₀ = *incr-lvl* (*cons-trail* (*Marked* *L* (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* = *N*
using *S* **by** *blast*
moreover have *atm-of* *L* \in *atms-of-msu* *N* **using** *Cons.prem*s(3) **by** *auto*
moreover have *undef*: *undefined-lit* (*trail* *S*) *L*
using *S* <*distinct* (*L*#*M*)> *calculation*(1) **by** (*auto simp*: *defined-lit-mapi* *defined-lit-map*)
ultimately have *cdcl_W* *S* ?*S*₀
using *cdcl_W.other*[*OF* *cdcl_W-o.decide*[*OF* *decide-rule*[*OF* *S*,
of *L* ?*S*₀]]] *S* **by** (*auto simp*: *state-eq-def* *simp* *del*: *state-simp*)
then show ?*case*
using *st* *S* *undef* **by** (*auto intro*!: *exI*[*of* - ?*S*₀])
qed

lemma *cdcl_W-strong-completeness*:
assumes
set *M* \models_s *set-mset* *N* **and**
consistent-interp (set *M*) **and**
distinct *M* **and**
atm-of ‘ (set *M*) \subseteq *atms-of-msu* *N*
obtains *S* **where**
state *S* = (*mapi* *Marked* (length *M*) *M*, *N*, {#}, length *M*, *C-True*) **and**
rtrancpl *cdcl_W* (init-state *N*) *S* **and**
final-cdcl_W-state *S*
proof –
obtain *S* **where**
st: *rtrancpl* *cdcl_W* (init-state *N*) *S* **and**
S: state *S* = (*mapi* *Marked* (length *M*) *M*, *N*, {#}, length *M*, *C-True*)
using *cdcl_W-can-do-step*[*OF* *assms*(2–4)] **by** *auto*
have *lits-of* (*mapi* *Marked* (length *M*) *M*) = *set* *M*
by (*induct* *M*, *auto*)
then have *mapi* *Marked* (length *M*) *M* \models_{asm} *N* **using** *assms*(1) *true-annots-true-cl*s **by** *metis*
then have *final-cdcl_W-state* *S*
using *S* **unfolding** *final-cdcl_W-state-def* **by** *auto*
then show ?*thesis* **using** *that* *st* *S* **by** *blast*

qed

17.6 Higher level strategy

The rules described previously do not lead to a conclusive state. We have to add a strategy.

17.6.1 Definition

lemma *tranclp-conflict-iff*[iff]:
 $full1 \text{ conflict } S \ S' \longleftrightarrow \text{conflict } S \ S'$
proof –
 have $tranclp \text{ conflict } S \ S' \implies \text{conflict } S \ S'$
 unfolding *full1-def* **by** (induct rule: *tranclp.induct*) *force+*
 then have $tranclp \text{ conflict } S \ S' \implies \text{conflict } S \ S'$ **by** (*meson rtranclpD*)
 then show ?thesis unfolding *full1-def* **by** (*metis conflictE conflicting-clause.simps(3)*
conflicting-update-conflicting state-eq-conflicting tranclp.intros(1))
 qed

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

lemma *rtranclp-cdcl_W-cp-rtranclp-cdcl_W*:
 $\text{cdcl}_W\text{-cp}^{**} \ S \ T \implies \text{cdcl}_W^{**} \ S \ T$
by (induction rule: *rtranclp-induct*) (auto simp: *cdcl_W-cp.simps* dest: *cdcl_W.intros*)

lemma *cdcl_W-cp-state-eq-compatible*:
assumes
 $\text{cdcl}_W\text{-cp } S \ T$ **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows $\text{cdcl}_W\text{-cp } S' \ T'$
using *assms*
apply (*induction*)
 using *conflict-state-eq-compatible* **apply** *auto*[1]
 using *propagate'* *propagate-state-eq-compatible* **by** *auto*

lemma *tranclp-cdcl_W-cp-state-eq-compatible*:
assumes
 $\text{cdcl}_W\text{-cp}^{++} \ S \ T$ **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows $\text{cdcl}_W\text{-cp}^{++} \ S' \ T'$
using *assms*
proof *induction*
 case *base*
 then show ?case
 using *cdcl_W-cp-state-eq-compatible* **by** *blast*
next
 case (*step U V*)
 obtain *ss* :: 'st **where**
 $\text{cdcl}_W\text{-cp } S \ ss \wedge \text{cdcl}_W\text{-cp}^{**} \ ss \ U$
by (*metis* (*no-types*) *step(1)* *tranclpD*)
 then show ?case
by (*meson* *cdcl_W-cp-state-eq-compatible* *rtranclp.rtrancl-into-rtrancl* *rtranclp-into-tranclp2*)

state-eq-ref step(2) step(4) step(5))
qed

lemma *conflicting-clause-full-cdcl_W-cp:*

conflicting S ≠ C-True ⇒ full cdcl_W-cp S S

unfolding *full-def rtrancpl-unfold trancpl-unfold by (auto simp add: cdcl_W-cp.simps)*

lemma *skip-unique:*

skip S T ⇒ skip S T' ⇒ T ∼ T'

by *(fastforce simp: state-eq-def simp del: state-simp)*

lemma *resolve-unique:*

resolve S T ⇒ resolve S T' ⇒ T ∼ T'

by *(fastforce simp: state-eq-def simp del: state-simp)*

lemma *cdcl_W-cp-no-more-clauses:*

assumes *cdcl_W-cp S S'*

shows *clauses S = clauses S'*

using *assms by (induct rule: cdcl_W-cp.induct) (auto elim!: conflictE propagateE)*

lemma *trancpl-cdcl_W-cp-no-more-clauses:*

assumes *cdcl_W-cp⁺⁺ S S'*

shows *clauses S = clauses S'*

using *assms by (induct rule: trancpl.induct) (auto dest: cdcl_W-cp-no-more-clauses)*

lemma *rtrancpl-cdcl_W-cp-no-more-clauses:*

assumes *cdcl_W-cp^{**} S S'*

shows *clauses S = clauses S'*

using *assms by (induct rule: rtrancpl-induct) (fastforce dest: cdcl_W-cp-no-more-clauses)+*

lemma *no-conflict-after-conflict:*

conflict S T ⇒ ¬conflict T U

by *fastforce*

lemma *no-propagate-after-conflict:*

conflict S T ⇒ ¬propagate T U

by *fastforce*

lemma *trancpl-cdcl_W-cp-propagate-with-conflict-or-not:*

assumes *cdcl_W-cp⁺⁺ S U*

shows *(propagate⁺⁺ S U ∧ conflicting U = C-True)*

*∨ (∃ T D. propagate^{**} S T ∧ conflict T U ∧ conflicting U = C-Clause D)*

proof –

have *propagate⁺⁺ S U ∨ (∃ T. propagate^{**} S T ∧ conflict T U)*

using *assms by induction*

*(force simp: cdcl_W-cp.simps trancpl-into-rtrancpl dest: no-conflict-after-conflict
no-propagate-after-conflict)+*

moreover

have *propagate⁺⁺ S U ⇒ conflicting U = C-True*

unfolding *trancpl-unfold-end by auto*

moreover

have *∧ T. conflict T U ⇒ ∃ D. conflicting U = C-Clause D*

by *auto*

ultimately show *?thesis by meson*

qed

lemma *cdcl_W-cp-conflicting-not-empty[simp]*: *conflicting S = C-Clause D* $\implies \neg \text{cdcl}_W\text{-cp } S S'$
proof
 assume *cdcl_W-cp S S'* **and** *conflicting S = C-Clause D*
 then show *False* **by** (*induct rule: cdcl_W-cp.induct*) *auto*
qed

lemma *no-step-cdcl_W-cp-no-conflict-no-propagate*:
 assumes *no-step cdcl_W-cp S*
 shows *no-step conflict S* **and** *no-step propagate S*
 using *assms conflict'* **apply** *blast*
by (*meson assms conflict' propagate'*)

CDCL with the reasonable strategy: we fully propagate the conflict and propagate, then we apply any other possible rule *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''*

17.6.2 Invariants

These are the same invariants as before, but lifted

lemma *cdcl_W-cp-learned-clause-inv*:
 assumes *cdcl_W-cp S S'*
 shows *learned-clss S = learned-clss S'*
 using *assms* **by** (*induct rule: cdcl_W-cp.induct*) *fastforce+*

lemma *rtrancpl-cdcl_W-cp-learned-clause-inv*:
 assumes *cdcl_W-cp** S S'*
 shows *learned-clss S = learned-clss S'*
 using *assms* **by** (*induct rule: rtrancpl-induct*) (*fastforce dest: cdcl_W-cp-learned-clause-inv*)**+**

lemma *trancpl-cdcl_W-cp-learned-clause-inv*:
 assumes *cdcl_W-cp++ S S'*
 shows *learned-clss S = learned-clss S'*
 using *assms* **by** (*simp add: rtrancpl-cdcl_W-cp-learned-clause-inv trancpl-into-rtrancpl*)

lemma *cdcl_W-cp-backtrack-lvl*:
 assumes *cdcl_W-cp S S'*
 shows *backtrack-lvl S = backtrack-lvl S'*
 using *assms* **by** (*induct rule: cdcl_W-cp.induct*) *fastforce+*

lemma *rtrancpl-cdcl_W-cp-backtrack-lvl*:
 assumes *cdcl_W-cp** S S'*
 shows *backtrack-lvl S = backtrack-lvl S'*
 using *assms* **by** (*induct rule: rtrancpl-induct*) (*fastforce dest: cdcl_W-cp-backtrack-lvl*)**+**

lemma *cdcl_W-cp-consistent-inv*:
 assumes *cdcl_W-cp S S'*
 and *cdcl_W-M-level-inv S*
 shows *cdcl_W-M-level-inv S'*
 using *assms*
proof (*induct rule: cdcl_W-cp.induct*)
 case (*conflict'*)

then show ?case using $cdcl_W$ -consistent-inv $cdcl_W.conflict$ by blast
 next
 case (propagate' $S S'$)
 have $cdcl_W S S'$
 using propagate'.hyps(1) propagate by blast
 then show $cdcl_W$ -M-level-inv S'
 using propagate'.prems(1) $cdcl_W$ -consistent-inv propagate by blast
 qed

lemma full1- $cdcl_W$ -cp-consistent-inv:

assumes full1 $cdcl_W$ -cp $S S'$
 and $cdcl_W$ -M-level-inv S
 shows $cdcl_W$ -M-level-inv S'
 using assms unfolding full1-def

proof –

have $cdcl_W$ -cp⁺⁺ $S S'$ and $cdcl_W$ -M-level-inv S using assms unfolding full1-def by auto
 then show ?thesis by (induct rule: tranclp.induct) (blast intro: $cdcl_W$ -cp-consistent-inv)+
 qed

lemma rtranclp- $cdcl_W$ -cp-consistent-inv:

assumes rtranclp $cdcl_W$ -cp $S S'$
 and $cdcl_W$ -M-level-inv S
 shows $cdcl_W$ -M-level-inv S'
 using assms unfolding full1-def
 by (induction rule: rtranclp.induct) (blast intro: $cdcl_W$ -cp-consistent-inv)+

lemma $cdcl_W$ -stgy-consistent-inv:

assumes $cdcl_W$ -stgy $S S'$
 and $cdcl_W$ -M-level-inv S
 shows $cdcl_W$ -M-level-inv S'
 using assms apply (induct rule: $cdcl_W$ -stgy.induct)
 unfolding full-unfold by (blast intro: $cdcl_W$ -consistent-inv full1- $cdcl_W$ -cp-consistent-inv
 $cdcl_W.other$)+

lemma rtranclp- $cdcl_W$ -stgy-consistent-inv:

assumes $cdcl_W$ -stgy^{**} $S S'$
 and $cdcl_W$ -M-level-inv S
 shows $cdcl_W$ -M-level-inv S'
 using assms by induction (auto dest!: $cdcl_W$ -stgy-consistent-inv)

lemma $cdcl_W$ -cp-no-more-init-clss:

assumes $cdcl_W$ -cp $S S'$
 shows init-clss $S = init-clss S'$
 using assms by (induct rule: $cdcl_W$ -cp.induct) auto

lemma tranclp- $cdcl_W$ -cp-no-more-init-clss:

assumes $cdcl_W$ -cp⁺⁺ $S S'$
 shows init-clss $S = init-clss S'$
 using assms by (induct rule: tranclp.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: tranclp-cdcl_W-cp-no-more-init-clss*
tranclp-cdcl_W-o-no-more-init-clss)
by (*metis cdcl_W-o-no-more-init-clss rtranclp-unfold tranclp-cdcl_W-cp-no-more-init-clss*)

lemma *rtranclp-cdcl_W-stgy-no-more-init-clss*:
assumes *cdcl_W-stgy** S S'* **and** *cdcl_W-M-level-inv S*
shows *init-clss S = init-clss S'*
using *assms*
apply (*induct rule: rtranclp-induct, simp*)
using *cdcl_W-stgy-no-more-init-clss* **by** (*simp add: rtranclp-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** $(\forall l \in \text{set } M. \neg \text{is-marked } l)$
using *assms* **by** *induction fastforce+*

lemma *rtranclp-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** $\forall l \in \text{set } M. \neg \text{is-marked } l$
using *assms* **by** *induction (fastforce dest!: cdcl_W-cp-dropWhile-trail')+*

lemma *cdcl_W-cp-dropWhile-trail*:
assumes *cdcl_W-cp S S'*
shows $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$
using *assms* **by** *induction fastforce+*

lemma *rtranclp-cdcl_W-cp-dropWhile-trail*:
assumes *cdcl_W-cp** S S'*
shows $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$
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-msu (init-clss S))
shows $\text{length } (\text{trail } S) \leq \text{card } (\text{atms-of-msu } (\text{init-clss } S))$

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 (trail S))*
using *assms(1,3) unfolding S* **by** (*auto simp add: finite-subset*)
have $\text{length } (\text{trail } S) = \text{card } (\text{atm-of ' lits-of } (\text{trail } S))$
using *no-dup-length-eq-card-atm-of-lits-of no-d* **by** *blast*
then show *?thesis* **using** *assms(1) unfolding no-strange-atm-def*
by (*auto simp add: assms(3) card-mono*)
qed

lemma *cdcl_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 $(\lambda S. \text{card } (\text{atms-of-msu } (\text{init-clss } S)) - \text{length } (\text{trail } S))$

```

    + (if conflicting  $S = C\text{-True}$  then 1 else 0))  $S$ 
  > ( $\lambda S. \text{card} (\text{atms-of-msu} (\text{init-clss } S)) - \text{length} (\text{trail } S)$ 
    + (if conflicting  $S = C\text{-True}$  then 1 else 0))  $T$ 
using assms
proof -
  have  $\text{length} (\text{trail } T) \leq \text{card} (\text{atms-of-msu} (\text{init-clss } T))$ 
  apply (rule length-model-le-vars)
    using cdclW-no-strange-atm-inv alien M-lev apply (meson cdclW cdclW.simps cdclW-cp.cases)
    using M-lev cdclW cdclW-cp-consistent-inv cdclW-M-level-inv-def apply blast
    using cdclW by (auto simp: cdclW-cp.simps)
  with assms
  show ?thesis by induction (auto split: split-if-asm)+
qed

```

```

lemma cdclW-cp-wf:  $\text{wf} \{(b,a). (\text{cdcl}_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a) \wedge \text{cdcl}_W\text{-cp } a \ b\}$ 
apply (rule wf-wf-if-measure'[of less-than - -
  ( $\lambda S. \text{card} (\text{atms-of-msu} (\text{init-clss } S)) - \text{length} (\text{trail } S)$ 
    + (if conflicting  $S = C\text{-True}$  then 1 else 0))])
apply simp
using cdclW-cp-decreasing-measure unfolding less-than-iff by blast

```

```

lemma rtranclp-cdclW-all-struct-inv-cdclW-cp-iff-rtranclp-cdclW-cp:
assumes
  lev: cdclW-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  $\text{cdcl}_W^{**} S \ T$ 
      by (metis rtranclp-unfold cdclW-cp-conflicting-not-empty cp st
        rtranclp-propagate-is-rtranclp-cdclW tranclp-cdclW-cp-propagate-with-conflict-or-not)
    then have
      cdclW-M-level-inv T and
      no-strange-atm T
      using  $\langle \text{cdcl}_W^{**} S \ T \rangle$  apply (simp add: assms(1) rtranclp-cdclW-consistent-inv)
      using  $\langle \text{cdcl}_W^{**} S \ T \rangle$  alien rtranclp-cdclW-no-strange-atm-inv lev by blast
    then have ( $\lambda a \ b. (\text{cdcl}_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a) \wedge \text{cdcl}_W\text{-cp } a \ b)^{**} T \ U$ 
      using cp by auto
    then show ?case using IH by auto
qed

```


qed

lemma *cdcl_W-cp-normalized-element*:

assumes

lev: *cdcl_W-M-level-inv S* **and**

no-strange-atm S

obtains *T* **where** *full cdcl_W-cp S T*

proof –

let *?inv* = $\lambda a. (cdcl_W\text{-}M\text{-level}\text{-}inv\ a \wedge no\text{-}strange\text{-}atm\ a)$

obtain *T* **where** *T*: *full* ($\lambda a\ b. ?inv\ a \wedge cdcl_W\text{-}cp\ a\ b$) *S T*

using *cdcl_W-cp-wf wf-exists-normal-form*[*of* $\lambda a\ b. ?inv\ a \wedge cdcl_W\text{-}cp\ a\ b$]

unfolding *full-def* **by** *blast*

then have *cdcl_W-cp** S T*

using *rtrancp-cdcl_W-all-struct-inv-cdcl_W-cp-iff-rtrancp-cdcl_W-cp assms* **unfolding** *full-def*
by *blast*

moreover

then have *cdcl_W** S T*

using *rtrancp-cdcl_W-cp-rtrancp-cdcl_W* **by** *blast*

then have

cdcl_W-M-level-inv T **and**

no-strange-atm T

using $\langle cdcl_W^{**}\ S\ T \rangle$ **apply** (*simp add: assms(1) rtrancp-cdcl_W-consistent-inv*)

using $\langle cdcl_W^{**}\ S\ T \rangle$ *assms(2) rtrancp-cdcl_W-no-strange-atm-inv lev* **by** *blast*

then have *no-step cdcl_W-cp T*

using *T* **unfolding** *full-def* **by** *auto*

ultimately show *thesis* **using** *that* **unfolding** *full-def* **by** *blast*

qed

lemma *in-atms-of-implies-atm-of-on-atms-of-ms*:

$C + \{\#L\# \} \in \# A \implies x \in \text{atms-of } C \implies x \in \text{atms-of-msu } A$

by (*metis add.commute atm-iff-pos-or-neg-lit atms-of-atms-of-ms-mono contra-subsetD*
mem-set-mset-iff multi-member-skip)

lemma *propagate-no-strange-atm*:

assumes

propagate S S' **and**

no-strange-atm S

shows *no-strange-atm S'*

using *assms* **by** *induction*

(*auto simp add: no-strange-atm-def clauses-def in-plus-implies-atm-of-on-atms-of-ms*

in-atms-of-implies-atm-of-on-atms-of-ms)

lemma *always-exists-full-cdcl_W-cp-step*:

assumes *no-strange-atm S*

shows $\exists S''. \text{full } cdcl_W\text{-}cp\ S\ S''$

using *assms*

proof (*induct card (atms-of-msu (init-clss S) – atm-of 'lits-of (trail S)) arbitrary: S*)

case 0 **note** *card = this(1)* **and** *alien = this(2)*

then have *atm: atms-of-msu (init-clss S) = atm-of ' lits-of (trail S)*

unfolding *no-strange-atm-def* **by** *auto*

{ **assume** *a: $\exists S'. \text{conflict } S\ S'$*

then obtain *S'* **where** *S': conflict S S'* **by** *metis*

then have $\forall S''. \neg cdcl_W\text{-}cp\ S'\ S''$ **by** *auto*

then have *?case* **using** *a S' cdcl_W-cp.conflict'* **unfolding** *full-def* **by** *blast*

}

```

moreover {
  assume  $a: \exists S'. \text{propagate } S S'$ 
  then obtain  $S'$  where  $\text{propagate } S S'$  by blast
  then obtain  $M N U k C L$  where  $S: \text{state } S = (M, N, U, k, C\text{-True})$ 
  and  $S': \text{state } S' = (\text{Propagated } L ( (C + \{\#L\#\})) \# M, N, U, k, C\text{-True})$ 
  and  $C + \{\#L\#\} \in \# \text{ clauses } S$ 
  and  $M \models_{as} C\text{Not } C$ 
  and  $\text{undefined-lit } M L$ 
  using  $\text{propagate}$  by auto
  have  $\text{atms-of-msu } U \subseteq \text{atms-of-msu } N$  using  $\text{alien } S$  unfolding  $\text{no-strange-atm-def}$  by auto
  then have  $\text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$ 
    using  $\langle C + \{\#L\#\} \in \# \text{ clauses } S \rangle S$  unfolding  $\text{atms-of-ms-def clauses-def}$  by force+
  then have False using  $\langle \text{undefined-lit } M L \rangle S$  unfolding  $\text{atm unfolding lits-of-def}$ 
    by  $(\text{auto simp add: defined-lit-map})$ 
}
ultimately show  $?case$  by  $(\text{metis } \text{cdcl}_W\text{-cp.cases full-def rtranclp.rtrancl-refl})$ 
next
case  $(\text{Suc } n)$  note  $IH = \text{this}(1)$  and  $\text{card} = \text{this}(2)$  and  $\text{alien} = \text{this}(3)$ 
{ assume  $a: \exists S'. \text{conflict } S S'$ 
  then obtain  $S'$  where  $S': \text{conflict } S S'$  by metis
  then have  $\forall S''. \neg \text{cdcl}_W\text{-cp } S' S''$  by auto
  then have  $?case$  unfolding  $\text{full-def Ex-def}$  using  $S' \text{cdcl}_W\text{-cp.conflict'}$  by blast
}
moreover {
  assume  $a: \exists S'. \text{propagate } S S'$ 
  then obtain  $S'$  where  $\text{propagate: propagate } S S'$  by blast
  then obtain  $M N U k C L$  where
     $S: \text{state } S = (M, N, U, k, C\text{-True})$  and
     $S': \text{state } S' = (\text{Propagated } L ( (C + \{\#L\#\})) \# M, N, U, k, C\text{-True})$  and
     $C + \{\#L\#\} \in \# \text{ clauses } S$  and
     $M \models_{as} C\text{Not } C$  and
     $\text{undefined-lit } M L$ 
  by fastforce
  then have  $\text{atm-of } L \notin \text{atm-of ' lits-of } M$ 
    unfolding  $\text{lits-of-def}$  by  $(\text{auto simp add: defined-lit-map})$ 
  moreover
    have  $\text{no-strange-atm } S'$  using  $\text{alien propagate propagate-no-strange-atm}$  by blast
    then have  $\text{atm-of } L \in \text{atms-of-msu } N$  using  $S'$  unfolding  $\text{no-strange-atm-def}$  by auto
    then have  $\bigwedge A. \{\text{atm-of } L\} \subseteq \text{atms-of-msu } N - A \vee \text{atm-of } L \in A$  by force
  moreover have  $\text{Suc } n - \text{card } \{\text{atm-of } L\} = n$  by simp
  moreover have  $\text{card } (\text{atms-of-msu } N - \text{atm-of ' lits-of } M) = \text{Suc } n$ 
    using  $\text{card } S S'$  by simp
  ultimately
    have  $\text{card } (\text{atms-of-msu } N - \text{atm-of ' insert } L (\text{lits-of } M)) = n$ 
      by  $(\text{metis } (\text{no-types}) \text{Diff-insert card-Diff-subset finite.emptyI finite.insertI image-insert})$ 
    then have  $n = \text{card } (\text{atms-of-msu } (\text{init-clss } S') - \text{atm-of ' lits-of } (\text{trail } S'))$ 
      using  $\text{card } S S'$  by simp
  then have  $a1: \text{Ex } (\text{full } \text{cdcl}_W\text{-cp } S')$  using  $IH \langle \text{no-strange-atm } S' \rangle$  by blast
  have  $?case$ 
    proof –
      obtain  $S'' :: 'st$  where
         $\text{ff1: } \text{cdcl}_W\text{-cp}^{**} S' S'' \wedge \text{no-step } \text{cdcl}_W\text{-cp } S''$ 
        using  $a1$  unfolding  $\text{full-def}$  by blast
      have  $\text{cdcl}_W\text{-cp}^{**} S S''$ 
        using  $\text{ff1 } \text{cdcl}_W\text{-cp.intros}(2)[\text{OF } \text{propagate}]$ 
    
```

```

      by (metis (no-types) converse-rtrancl-into-rtrancl)
    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 cdclW-cp.cases rtrancl.rtrancl-refl)
qed

```

17.6.3 Literal of highest level in conflicting clauses

One important property of the cdcl_W with strategy is that, whenever a conflict takes place, there is at least a literal of level k involved (except if we have derived the false clause). The reason is that we apply conflicts before a decision is taken.

abbreviation *no-clause-is-false* :: 'st \Rightarrow bool **where**

no-clause-is-false \equiv

$\lambda S. (\text{conflicting } S = C\text{-True} \longrightarrow (\forall D \in \# \text{ clauses } S. \neg \text{trail } S \models_{as} C\text{Not } D))$

abbreviation *conflict-is-false-with-level* :: 'st \Rightarrow bool **where**

conflict-is-false-with-level $S' \equiv \forall D. \text{conflicting } S' = C\text{-Clause } D \longrightarrow D \neq \{\#\}$

$\longrightarrow (\exists L \in \# D. \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S')$

lemma *not-conflict-not-any-negated-init-clss*:

assumes $\forall S'. \neg \text{conflict } S S'$

shows *no-clause-is-false* S

using *assms state-eq-ref* **by** blast

lemma *full-cdcl_W-cp-not-any-negated-init-clss*:

assumes *full* $\text{cdcl}_W\text{-cp} S S'$

shows *no-clause-is-false* S'

using *assms not-conflict-not-any-negated-init-clss* **unfolding** *full-def* **by** blast

lemma *full1-cdcl_W-cp-not-any-negated-init-clss*:

assumes *full1* $\text{cdcl}_W\text{-cp} S S'$

shows *no-clause-is-false* S'

using *assms not-conflict-not-any-negated-init-clss* **unfolding** *full1-def* **by** blast

lemma *cdcl_W-stgy-not-non-negated-init-clss*:

assumes $\text{cdcl}_W\text{-stgy} S S'$

shows *no-clause-is-false* S'

using *assms apply* (induct rule: $\text{cdcl}_W\text{-stgy.induct}$)

using *full1-cdcl_W-cp-not-any-negated-init-clss* *full-cdcl_W-cp-not-any-negated-init-clss* **by** metis+

lemma *rtrancl-cdcl_W-stgy-not-non-negated-init-clss*:

assumes $\text{cdcl}_W\text{-stgy}^{**} S S'$ **and** *no-clause-is-false* S

shows *no-clause-is-false* S'

using *assms* **by** (induct rule: rtrancl-induct) (auto simp: $\text{cdcl}_W\text{-stgy-not-non-negated-init-clss}$)

lemma *cdcl_W-stgy-conflict-ex-lit-of-max-level*:

assumes $\text{cdcl}_W\text{-cp} S S'$

and *no-clause-is-false* S

and $\text{cdcl}_W\text{-M-level-inv } S$

shows *conflict-is-false-with-level* S'

using *assms*

```

proof (induct rule: cdclW-cp.induct)
  case conflict'
  then show ?case by auto
next
  case propagate'
  then show ?case by auto
qed

lemma no-chained-conflict:
  assumes conflict  $S S'$ 
  and conflict  $S' S''$ 
  shows False
  using assms by fastforce

lemma rtrancpl-cdclW-cp-propa-or-propa-conf:
  assumes cdclW-cp**  $S U$ 
  shows propagate**  $S U \vee (\exists T. \text{propagate** } S T \wedge \text{conflict } T U)$ 
  using assms
proof induction
  case base
  then show ?case by auto
next
  case (step  $U V$ ) note  $SU = \text{this}(1)$  and  $UV = \text{this}(2)$  and  $IH = \text{this}(3)$ 
  consider (confl)  $T$  where propagate**  $S T$  and conflict  $T U$ 
  | (propa) propagate**  $S U$  using  $IH$  by auto
  then show ?case
  proof cases
  case confl
  then have False using  $UV$  by auto
  then show ?thesis by fast
next
  case propa
  also have conflict  $U V \vee \text{propagate } U V$  using  $UV$  by (auto simp add: cdclW-cp.simps)
  ultimately show ?thesis by force
qed
qed

lemma rtrancpl-cdclW-co-conflict-ex-lit-of-max-level:
  assumes full: full cdclW-cp  $S U$ 
  and cls-f: no-clause-is-false  $S$ 
  and conflict-is-false-with-level  $S$ 
  and lev: cdclW-M-level-inv  $S$ 
  shows conflict-is-false-with-level  $U$ 
proof (intro allI impI)
  fix  $D$ 
  assume confl: conflicting  $U = C\text{-Clause } D$  and
   $D: D \neq \{\#\}$ 
  consider (CT) conflicting  $S = C\text{-True} \mid (SD) D'$  where conflicting  $S = C\text{-Clause } D'$ 
  by (cases conflicting  $S$ ) auto
  then show  $\exists L \in \#D. \text{get-level } L (\text{trail } U) = \text{backtrack-lvl } U$ 
  proof cases
  case SD
  then have  $S = U$ 
  by (metis (no-types) assms(1) cdclW-cp-conflicting-not-empty full-def rtrancplD trancplD)
  then show ?thesis using assms(3) confl  $D$  by blast-

```

```

next
case CT
have init-clss U = init-clss S and learned-clss U = learned-clss S
  using assms(1) unfolding full-def
  apply (metis (no-types) rtrancpD trancp-cdclW-cp-no-more-init-clss)
  by (metis (mono-tags, lifting) assms(1) full-def rtrancp-cdclW-cp-learned-clause-inv)
obtain T where propagate** S T and TU: conflict T U
proof -
  have f5: U ≠ S
  using confl CT by force
  then have cdclW-cp++ S U
  by (metis full full-def rtrancpD)
  have  $\bigwedge p \text{ pa. } \neg \text{propagate } p \text{ pa} \vee \text{conflicting } \text{pa} =$ 
    (C-True::'v literal multiset conflicting-clause)
  by auto
  then show ?thesis
  using f5 that trancp-cdclW-cp-propagate-with-conflict-or-not[OF  $\langle \text{cdcl}_W\text{-cp}^{++} S U \rangle$ ]
  full confl CT unfolding full-def by auto
qed
have init-clss T = init-clss S and learned-clss T = learned-clss S
  using TU  $\langle \text{init-clss } U = \text{init-clss } S \rangle \langle \text{learned-clss } U = \text{learned-clss } S \rangle$  by auto
then have D ∈# clauses S
  using TU confl by (fastforce simp: clauses-def)
then have  $\neg \text{trail } S \models_{\text{as}} \text{CNot } D$ 
  using cls-f CT by simp
moreover
  obtain M where tr-U: trail U = M @ trail S and nm:  $\forall m \in \text{set } M. \neg \text{is-marked } m$ 
  by (metis (mono-tags, lifting) assms(1) full-def rtrancp-cdclW-cp-dropWhile-trail)
  have trail U  $\models_{\text{as}} \text{CNot } D$ 
  using TU confl by auto
ultimately obtain L where L ∈# D and  $\neg L \in \text{lits-of } M$ 
  unfolding tr-U CNot-def true-annot-def Ball-def true-annot-def true-cl-def by auto

moreover have inv-U: cdclW-M-level-inv U
  by (metis cdclW-stgy.conflict' cdclW-stgy-consistent-inv full full-unfold lev)
moreover
  have backtrack-lvl U = backtrack-lvl S
  using full unfolding full-def by (auto dest: rtrancp-cdclW-cp-backtrack-lvl)

moreover
  have no-dup (trail U)
  using inv-U unfolding cdclW-M-level-inv-def by auto
{ fix x :: ('v, nat, 'v literal multiset) marked-lit and
  xb :: ('v, nat, 'v literal multiset) marked-lit
  assume a1: atm-of L = atm-of (lit-of xb)
  moreover assume a2:  $\neg L = \text{lit-of } x$ 
  moreover assume a3:  $(\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } M$ 
     $\cap (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } (\text{trail } S) = \{\}$ 
  moreover assume a4:  $x \in \text{set } M$ 
  moreover assume a5:  $xb \in \text{set } (\text{trail } S)$ 
  moreover have atm-of ( $\neg L$ ) = atm-of L
  by auto
  ultimately have False
  by auto
}

```

then have $LS: atm\text{-}of\ L \notin atm\text{-}of\ ' lits\text{-}of\ (trail\ S)$
using $\langle -L \in lits\text{-}of\ M \rangle \langle no\text{-}dup\ (trail\ U) \rangle$ **unfolding** $tr\text{-}U\ lits\text{-}of\text{-}def$ **by** *auto*
ultimately have $get\text{-}level\ L\ (trail\ U) = backtrack\text{-}lvl\ U$
proof (*cases get-all-levels-of-marked (trail S) $\neq []$, goal-cases*)
case 2 note $LD = this(1)$ **and** $LM = this(2)$ **and** $inv\text{-}U = this(3)$ **and** $US = this(4)$ **and**
 $LS = this(5)$ **and** $ne = this(6)$
have $backtrack\text{-}lvl\ S = 0$
using $lev\ ne$ **unfolding** $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** *auto*
moreover have $get\text{-}rev\text{-}level\ L\ 0\ (rev\ M) = 0$
using nm **by** *auto*
ultimately show *?thesis* **using** $LS\ ne\ US$ **unfolding** $tr\text{-}U$
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** $inv\text{-}U = this(3)$ **and** $US = this(4)$ **and**
 $LS = this(5)$ **and** $ne = this(6)$

have $hd\ (get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (trail\ S)) = backtrack\text{-}lvl\ S$
using $ne\ lev$ **unfolding** $cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$
by (*cases get-all-levels-of-marked (trail S) auto*)
moreover have $atm\text{-}of\ L \in atm\text{-}of\ ' lits\text{-}of\ M$
using $\langle -L \in lits\text{-}of\ M \rangle$ **by** (*simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def*)
ultimately show *?thesis*
using $nm\ ne$ **unfolding** $tr\text{-}U$
using $get\text{-}level\text{-}skip\text{-}beginning\text{-}hd\text{-}get\text{-}all\text{-}levels\text{-}of\text{-}marked[OF\ LS,\ of\ M]$
 $get\text{-}level\text{-}skip\text{-}in\text{-}all\text{-}not\text{-}marked[of\ rev\ M\ L\ backtrack\text{-}lvl\ S]$
unfolding $lits\text{-}of\text{-}def\ US$
by *auto*
qed
then show $\exists L \in \#D. get\text{-}level\ L\ (trail\ U) = backtrack\text{-}lvl\ U$
using $\langle L \in \#D \rangle$ **by** *blast*
qed
qed

17.6.4 Literal of highest level in marked literals

definition $mark\text{-}is\text{-}false\text{-}with\text{-}level :: 'st \Rightarrow bool$ **where**

$mark\text{-}is\text{-}false\text{-}with\text{-}level\ S' \equiv$

$\forall D\ M1\ M2\ L. M1\ @\ Propagated\ L\ D\ \# \ M2 = trail\ S' \longrightarrow D - \{\#L\} \neq \{\#\}$
 $\longrightarrow (\exists L. L \in \#D \wedge get\text{-}level\ L\ (trail\ S') = get\text{-}maximum\text{-}possible\text{-}level\ M1)$

definition $no\text{-}more\text{-}propagation\text{-}to\text{-}do :: 'st \Rightarrow bool$ **where**

$no\text{-}more\text{-}propagation\text{-}to\text{-}do\ S \equiv$

$\forall D\ M\ M'\ L. D + \{\#L\} \in \# \text{ clauses } S \longrightarrow trail\ S = M' @ M \longrightarrow M \models_{as} CNot\ D$
 $\longrightarrow undefined\text{-}lit\ M\ L \longrightarrow get\text{-}maximum\text{-}possible\text{-}level\ M < backtrack\text{-}lvl\ S$
 $\longrightarrow (\exists L. L \in \#D \wedge get\text{-}level\ L\ (trail\ S) = get\text{-}maximum\text{-}possible\text{-}level\ M)$

lemma $propagate\text{-}no\text{-}more\text{-}propagation\text{-}to\text{-}do$:

assumes $propagate: propagate\ S\ S'$

and $H: no\text{-}more\text{-}propagation\text{-}to\text{-}do\ S$

and $M: cdcl_W\text{-}M\text{-}level\text{-}inv\ S$

shows $no\text{-}more\text{-}propagation\text{-}to\text{-}do\ S'$

using *assms*

proof –

obtain $M\ N\ U\ k\ C\ L$ **where**

$S: state\ S = (M, N, U, k, C\text{-}True)$ **and**

S' : state $S' = (\text{Propagated } L \ ((C + \{\#L\#\})) \# M, N, U, k, C\text{-True})$ **and**
 $C + \{\#L\#\} \in \# \text{ clauses } S$ **and**
 $M \models_{as} C \text{Not } C$ **and**
undefined-lit $M \ L$
using *propagate* **by** *auto*
let $?M' = \text{Propagated } L \ ((C + \{\#L\#\})) \# M$
show *?thesis unfolding no-more-propagation-to-do-def*
proof (*intro allI impI*)
fix $D \ M1 \ M2 \ L'$
assume $D\text{-}L: D + \{\#L'\#\} \in \# \text{ clauses } S'$
and *trail* $S' = M2 \ @ \ M1$
and *get-max*: *get-maximum-possible-level* $M1 < \text{backtrack-lvl } S'$
and $M1 \models_{as} C \text{Not } D$
and *undef*: *undefined-lit* $M1 \ L'$
have $tl \ M2 \ @ \ M1 = \text{trail } S \vee (M2 = [] \wedge M1 = \text{Propagated } L \ ((C + \{\#L\#\})) \# M)$
using $\langle \text{trail } S' = M2 \ @ \ M1 \rangle S' \ S$ **by** (*cases* $M2$) *auto*
moreover {
assume $tl \ M2 \ @ \ M1 = \text{trail } S$
moreover **have** $D + \{\#L'\#\} \in \# \text{ clauses } S$ **using** $D\text{-}L \ S \ S'$ **unfolding** *clauses-def* **by** *auto*
moreover **have** *get-maximum-possible-level* $M1 < \text{backtrack-lvl } S$
using *get-max* $S \ S'$ **by** *auto*
ultimately obtain L' **where** $L' \in \# D$ **and**
get-level $L' \ (\text{trail } S) = \text{get-maximum-possible-level } M1$
using $H \ \langle M1 \models_{as} C \text{Not } D \rangle \text{undef}$ **unfolding** *no-more-propagation-to-do-def* **by** *metis*
moreover
{ **have** *cdcl_W-M-level-inv* S'
using *cdcl_W-consistent-inv*[$OF - M$] *cdcl_W.propagate*[*OF propagate*] **by** *blast*
then have *no-dup* $?M'$ **using** S' **unfolding** *cdcl_W-M-level-inv-def* **by** *auto*
moreover
have *atm-of* $L' \in \text{atm-of } (lits\text{-of } M1)$
using $\langle L' \in \# D \rangle \langle M1 \models_{as} C \text{Not } D \rangle$ **by** (*metis atm-of-uminus image-eqI*
in-CNot-implies-uminus(2))
then have *atm-of* $L' \in \text{atm-of } (lits\text{-of } M)$
using $\langle tl \ M2 \ @ \ M1 = \text{trail } S \rangle S$ **by** *auto*
ultimately have *atm-of* $L \neq \text{atm-of } L'$ **unfolding** *lits-of-def* **by** *auto*
}
ultimately have $\exists L' \in \# D. \text{get-level } L' \ (\text{trail } S') = \text{get-maximum-possible-level } M1$
using $S \ S'$ **by** *auto*
}
moreover {
assume $M2 = []$ **and** $M1: M1 = \text{Propagated } L \ ((C + \{\#L\#\})) \# M$
have *cdcl_W-M-level-inv* S'
using *cdcl_W-consistent-inv*[$OF - M$] *cdcl_W.propagate*[*OF propagate*] **by** *blast*
then have *get-all-levels-of-marked* $(\text{trail } S') = \text{rev } ([\text{Suc } 0..<(\text{Suc } 0+k)])$
using S' **unfolding** *cdcl_W-M-level-inv-def* **by** *auto*
then have *get-maximum-possible-level* $M1 = \text{backtrack-lvl } S'$
using *get-maximum-possible-level-max-get-all-levels-of-marked*[*of* $M1$] $S' \ M1$
by (*auto intro: Max-eqI*)
then have *False* **using** *get-max* **by** *auto*
}
ultimately show $\exists L. L \in \# D \wedge \text{get-level } L \ (\text{trail } S') = \text{get-maximum-possible-level } M1$ **by** *fast*
qed
qed

lemma *conflict-no-more-propagation-to-do*:

```

assumes conflict: conflict  $S$   $S'$ 
and  $H$ : no-more-propagation-to-do  $S$ 
and  $M$ : cdclW-M-level-inv  $S$ 
shows no-more-propagation-to-do  $S'$ 
using assms unfolding no-more-propagation-to-do-def conflict.simps by force

lemma cdclW-cp-no-more-propagation-to-do:
assumes conflict: cdclW-cp  $S$   $S'$ 
and  $H$ : no-more-propagation-to-do  $S$ 
and  $M$ : cdclW-M-level-inv  $S$ 
shows no-more-propagation-to-do  $S'$ 
using assms
proof (induct rule: cdclW-cp.induct)
case (conflict'  $S$   $S'$ )
then show ?case using conflict-no-more-propagation-to-do[of  $S$   $S'$ ] by blast
next
case (propagate'  $S$   $S'$ ) note  $S = \text{this}$ 
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  $\exists S'$ . cdclW-stgy  $S$   $S'$ 
proof –
obtain  $S''$  where full cdclW-cp  $S'$   $S''$ 
using always-exists-full-cdclW-cp-step  $alien$  cdclW-no-strange-atm-inv cdclW-o-no-more-init-clss
   $o$  other lev by (meson cdclW-consistent-inv)
then show ?thesis
using assms by (metis always-exists-full-cdclW-cp-step cdclW-stgy.conflict' full-unfold other')
qed

lemma backtrack-no-decomp:
assumes  $S$ : state  $S = (M, N, U, k, C\text{-}Clause (D + \{\#L\# \}))$ 
and  $L$ : get-level  $L$   $M = k$ 
and  $D$ : get-maximum-level  $D$   $M < k$ 
and  $M\text{-}L$ : cdclW-M-level-inv  $S$ 
shows  $\exists S'$ . cdclW-o  $S$   $S'$ 
proof –
have  $L\text{-}D$ : get-level  $L$   $M = \text{get-maximum-level } (D + \{\#L\# \})$   $M$ 
using  $L$   $D$  by (simp add: get-maximum-level-plus)
let ? $i = \text{get-maximum-level } D$   $M$ 
obtain  $K$   $M1$   $M2$  where  $K$ : (Marked  $K$  (? $i + 1$ )  $\#$   $M1, M2$ )  $\in$  set (get-all-marked-decomposition
 $M)$ 
using backtrack-ex-decomp[OF  $M\text{-}L$ , of ? $i$ ]  $D$   $S$  by auto
show ?thesis using backtrack-rule[OF  $S$   $K$   $L$   $L\text{-}D$ ] by (meson bj cdclW-bj.simps state-eq-ref)
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 confl: cdclW-conflicting S
and confl-k: conflict-is-false-with-level S
shows (conflicting S = C-Clause {#} ∧ unsatisfiable (set-mset (init-clss S)))
      ∨ (conflicting S = C-True ∧ trail S ⊨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
  have conflicting S = C-Clause {#}
    ∨ conflicting S = C-True
    ∨ (∃ D L. conflicting S = C-Clause (D + {#L#}))
  apply (case-tac conflicting S, auto)
  by (case-tac x2, auto)
moreover {
  assume conflicting S = C-Clause {#}
  then have unsatisfiable (set-mset (init-clss S))
    using assms(3) unfolding cdclW-learned-clause-def true-clss-clss-def
    by (metis (no-types, lifting) Un-insert-right atms-of-empty satisfiable-def
      sup-bot.right-neutral total-over-m-insert total-over-set-empty true-clss-empty)
}
moreover {
  assume conflicting S = C-True
  { assume ¬?M ⊨asm ?N
    have atm-of ' (lits-of ?M) = atms-of-msu ?N (is ?A = ?B)
    proof
      show ?A ⊆ ?B using alien unfolding no-strange-atm-def by auto
      show ?B ⊆ ?A
        proof (rule ccontr)
          assume ¬?B ⊆ ?A
          then obtain l where l ∈ ?B and l ∉ ?A by auto
          then have undefined-lit ?M (Pos l)
            using ⟨l ∉ ?A⟩ unfolding lits-of-def by (auto simp add: defined-lit-map)
          then have ∃ S'. cdclW-o S S'
            using cdclW-o.decide decide.intros ⟨l ∈ ?B⟩ no-strange-atm-def
            by (metis ⟨conflicting S = C-True⟩ literal.sel(1) state-eq-def)
          then show False
            using termi cdclW-then-exists-cdclW-stgy-step[OF - alien] level-inv by blast
        qed
      qed
    obtain D where ¬ ?M ⊨a D and D ∈# ?N
      using ⟨¬?M ⊨asm ?N⟩ unfolding lits-of-def true-annots-def Ball-def by auto
    have atms-of D ⊆ atm-of ' (lits-of ?M)
      using ⟨D ∈# ?N⟩ unfolding ⟨atm-of ' (lits-of ?M) = atms-of-msu ?N⟩ atms-of-ms-def
      by (auto simp add: atms-of-def)
    then have a1: atm-of ' set-mset D ⊆ atm-of ' lits-of (trail S)
      by (auto simp add: atms-of-def lits-of-def)
    have total-over-m (lits-of ?M) {D}
      using ⟨atms-of D ⊆ atm-of ' (lits-of ?M)⟩ atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      by (fastforce simp: total-over-set-def)
    then have ?M ⊨as CNot D
      using total-not-true-clss-true-clss-CNot ⟨¬ trail S ⊨a D⟩ true-annot-def

```

```

    true-annots-true-clb by fastforce
  then have False
  proof -
    obtain S' where
      f2: full cdclW-cp S S'
      by (meson alien always-exists-full-cdclW-cp-step level-inv)
    then have S' = S
      using cdclW-stgy.conflict'[of S] by (metis (no-types) full-unfold termi)
    then show ?thesis
      using f2 ⟨D ∈ # init-clss S⟩ ⟨conflicting S = C-True⟩ ⟨trail S ⊨as CNot D⟩
      clauses-def full-cdclW-cp-not-any-negated-init-clss by auto
  qed
}
then have ?M ⊨asm ?N by blast
}
moreover {
  assume ∃ D L. conflicting S = C-Clause (D + {#L#})
  obtain D L where LD: conflicting S = C-Clause (D + {#L#}) and get-level L ?M = ?k
  proof -
    obtain mm :: 'v literal multiset and ll :: 'v literal where
      f2: conflicting S = C-Clause (mm + {#ll#})
      using ⟨∃ D L. conflicting S = C-Clause (D + {#L#})⟩ by force
    have ∀ m. (conflicting S ≠ C-Clause m ∨ m = {#})
      ∨ (∃ l. l ∈ # m ∧ get-level l (trail S) = backtrack-lvl S)
      using confl-k by blast
    then show ?thesis
      using f2 that by (metis (no-types) multi-member-split single-not-empty union-eq-empty)
  qed
  let ?D = D + {#L#}
  have ?D ≠ {#} by auto
  have ?M ⊨as CNot ?D using confl LD unfolding cdclW-conflicting-def by auto
  then have ?M ≠ [] unfolding true-annots-def Ball-def true-annot-def true-clb-def by force
  { have M: ?M = hd ?M # tl ?M using ⟨?M ≠ []⟩ list.collapse by fastforce
    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 (case-tac hd ?M) auto
    have get-all-levels-of-marked (hd (trail S) # tl (trail S))
      = rev [1..<1 + length (get-all-levels-of-marked ?M)]
      using level-inv ⟨get-level L ?M = ?k⟩ M unfolding cdclW-M-level-inv-def M[symmetric]
      by blast
    then have l'-tl: l' # get-all-levels-of-marked (tl ?M)
      = rev [1..<1 + length (get-all-levels-of-marked ?M)] unfolding L' by simp
    moreover have ... = length (get-all-levels-of-marked ?M)
      # rev [1..W-M-level-inv-def by auto
    have *: ∧list. no-dup list ⇒
      - L ∈ lits-of list ⇒ atm-of L ∈ atm-of ' lits-of list
      by (metis atm-of-uminus imageI)
  }
}

```

```

have L' = -L
proof (rule ccontr)
  assume ¬ ?thesis
  moreover have -L ∈ lits-of ?M using confl LD unfolding cdclW-conflicting-def by auto
  ultimately have get-level L (hd (trail S) # tl (trail S)) = get-level L (tl ?M)
    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-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 L (tl ?M) < ?k
      using get-maximum-possible-level-ge-get-level[of L tl ?M]
      by (metis One-nat-def add.right-neutral add-Suc-right diff-add-inverse2
        get-maximum-possible-level-max-get-all-levels-of-marked k' le-imp-less-Suc
        list.simps(15))
    finally show False using ⟨get-level L ?M = ?k⟩ M by auto
qed
have L: hd ?M = Marked (-L) ?k using ⟨l' = ?k⟩ ⟨L' = -L⟩ L' by auto

have g-a-l: get-all-levels-of-marked ?M = rev [1..<1 + ?k]
  using level-inv ⟨get-level L ?M = ?k⟩ M unfolding cdclW-M-level-inv-def by auto
have g-k: get-maximum-level D (trail S) ≤ ?k
  using get-maximum-possible-level-ge-get-maximum-level[of D ?M]
  get-maximum-possible-level-max-get-all-levels-of-marked[of ?M]
  by (auto simp add: Max-n-upt g-a-l)
have get-maximum-level D (trail S) < ?k
proof (rule ccontr)
  assume ¬ ?thesis
  then have get-maximum-level D (trail S) = ?k using M g-k unfolding L by auto
  then obtain L' where L' ∈ # D and L-k: get-level L' ?M = ?k
    using get-maximum-level-exists-lit[of ?k D ?M] unfolding k'[symmetric] by auto
  have L ≠ L' using no-dup ⟨L' ∈ # D⟩
    unfolding distinct-cdclW-state-def LD by (metis add commute add-eq-self-zero
      count-single count-union less-not-refl3 distinct-mset-def union-single-eq-member)
  have L' = -L
  proof (rule ccontr)
    assume ¬ ?thesis
    then have get-level L' ?M = get-level L' (tl ?M)
      using M ⟨L ≠ L'⟩ get-level-skip-beginning[of L' hd ?M tl ?M] unfolding L
      by (auto simp add: atm-of-eq-atm-of)
    moreover have ... < ?k
      using level-inv g-r get-rev-level-less-max-get-all-levels-of-marked[of L' 0
        rev (tl ?M)] L-k l'-tl calculation g-a-l
      by (auto simp add: Max-n-upt cdclW-M-level-inv-def)
    finally show False using L-k by simp
  qed
  then have taut: tautology (D + {#L#})
    using ⟨L' ∈ # D⟩ by (metis add commute mset-leD mset-le-add-left multi-member-this
      tautology-minus)
  have consistent-interp (lits-of ?M)
    using level-inv unfolding cdclW-M-level-inv-def by auto

```

```

    then have  $\neg ?M \models_{as} CNot ?D$ 
      using taut by (metis (no-types)  $\langle L' = - L \rangle \langle L' \in \# D \rangle$  add.commute consistent-interp-def
        in-CNot-implies-uminus(2) mset-leD mset-le-add-left 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
then have False
  using backtrack-no-decomp[OF -  $\langle get\text{-}level\ L\ (trail\ S) = backtrack\text{-}lvl\ S \rangle$  - level-inv]
  LD alien termi by (metis cdclW-then-exists-cdclW-stgy-step level-inv)
}
moreover {
  assume  $\neg is\text{-}marked\ (hd\ ?M)$ 
  then obtain  $L'\ C$  where  $L'C: hd\ ?M = Propagated\ L'\ C$  by (case-tac  $hd\ ?M$ , auto)
  then have  $M: ?M = Propagated\ L'\ C \# tl\ ?M$  using  $\langle ?M \neq [] \rangle$  list.collapse by fastforce
  then obtain  $C'$  where  $C': C = C' + \{\#L'\#\}$ 
    using confl unfolding cdclW-conflicting-def by (metis append-Nil diff-single-eq-union)
  { assume  $-L' \notin \# ?D$ 
    then have False
      using bj[OF cdclW-bj.skip[OF skip-rule[OF -  $\langle -L' \notin \# ?D \rangle \langle ?D \neq \{\#\} \rangle$ , of  $S\ C\ tl\ (trail\ S)$  -
        ]]]
      termi M by (metis LD alien cdclW-then-exists-cdclW-stgy-step state-eq-def level-inv)
    }
  }
moreover {
  assume  $-L' \in \# ?D$ 
  then obtain  $D'$  where  $D': ?D = D' + \{\#-L'\#\}$  by (metis insert-DiffM2)
  have  $g\text{-}r: get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (Propagated\ L'\ C \# tl\ (trail\ S))$ 
    =  $rev\ [Suc\ 0..<Suc\ (length\ (get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (trail\ S)))]$ 
    using level-inv M unfolding cdclW-M-level-inv-def by auto
  have  $Max\ (insert\ 0\ (set\ (get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (Propagated\ L'\ C \# tl\ (trail\ S)))) = ?k$ 
    using level-inv M unfolding g-r cdclW-M-level-inv-def set-rev
    by (auto simp add: Max-n-upt)
  then have  $get\text{-}maximum\text{-}level\ D'\ (Propagated\ L'\ C \# tl\ ?M) \leq ?k$ 
    using get-maximum-possible-level-ge-get-maximum-level[of  $D'\ Propagated\ L'\ C \# tl\ ?M$ ]
    unfolding get-maximum-possible-level-max-get-all-levels-of-marked by auto
  then have  $get\text{-}maximum\text{-}level\ D'\ (Propagated\ L'\ C \# tl\ ?M) = ?k$ 
     $\vee get\text{-}maximum\text{-}level\ D'\ (Propagated\ L'\ C \# tl\ ?M) < ?k$ 
    using le-neq-implies-less by blast
  moreover {
    assume  $g\text{-}D'\text{-}k: get\text{-}maximum\text{-}level\ D'\ (Propagated\ L'\ C \# tl\ ?M) = ?k$ 
    have False
    proof -
      have  $f1: get\text{-}maximum\text{-}level\ D'\ (trail\ S) = backtrack\text{-}lvl\ S$ 
        using M g-D'-k by auto
      have  $(trail\ S, init\text{-}clss\ S, learned\text{-}clss\ S, backtrack\text{-}lvl\ S, C\text{-}Clause\ (D + \{\#L'\#\}))$ 
        = state S
        by (metis (no-types) LD)
      then have cdclW-o S (update-conflicting (C-Clause ( $D' \# \cup C'$ )) (tl-trail S))
        using f1 bj[OF cdclW-bj.resolve[OF resolve-rule[of  $S\ L'\ C'\ tl\ ?M\ ?N\ ?U\ ?k\ D$ ]]]
         $C'\ D'\ M$  by (metis state-eq-def)
      then show ?thesis
        by (meson alien cdclW-then-exists-cdclW-stgy-step termi level-inv)
    qed
  }
}
moreover {

```

```

assume get-maximum-level  $D'$  (Propagated  $L' C \# tl ?M$ ) < ? $k$ 
then have False
proof –
  assume  $a1$ : get-maximum-level  $D'$  (Propagated  $L' C \# tl (trail S)$ ) < backtrack-lvl  $S$ 
  obtain  $mm :: 'v$  literal multiset and  $ll :: 'v$  literal where
     $f2$ : conflicting  $S = C$ -Clause ( $mm + \{\#ll\# \}$ )
    get-level  $ll (trail S) = backtrack-lvl S$ 
    using  $LD \langle get-level L (trail S) = backtrack-lvl S \rangle$  by blast
  then have  $f3$ : get-maximum-level  $D'$  ( $trail S$ )  $\leq$  get-level  $ll (trail S)$ 
    using  $M a1$  by force
  have get-level  $ll (trail S) \neq$  get-maximum-level  $D'$  ( $trail S$ )
    using  $f2 M$  calculation(2) by presburger
  have  $f1$ :  $trail S =$  Propagated  $L' C \# tl (trail S)$ 
    conflicting  $S = C$ -Clause ( $D' + \{\#- L'\# \}$ )
    using  $D' LD M$  by force+
  have  $f2$ : conflicting  $S = C$ -Clause ( $mm + \{\#ll\# \}$ )
    get-level  $ll (trail S) = backtrack-lvl S$ 
    using  $f2$  by force+
  have  $ll = - L'$ 
    by (metis (no-types)  $D' LD \langle get-level ll (trail S) \neq$  get-maximum-level  $D'$  ( $trail S$ ) $\rangle$ 
      conflicting-clause.inject  $f2 f3$  get-maximum-level-ge-get-level insert-noteq-member
      le-antisym)
  then show ?thesis
    using  $f2 f1 M$  backtrack-no-decomp[of  $S$ ]
    by (metis (no-types)  $a1$  alien cdclW-then-exists-cdclW-stgy-step level-inv termi)
  qed
}
ultimately have False by blast
}
ultimately have False by blast
}
ultimately have False by blast
}
ultimately show ?thesis by blast
qed

```

```

lemma cdclW-cp-tranclp-cdclW:
  cdclW-cp  $S S' \implies cdcl_W^{++} S S'$ 
  apply (induct rule: cdclW-cp.induct)
  by (meson cdclW.conflict cdclW.propagate tranclp.r-into-trancl tranclp.trancl-into-trancl) $+$ 

```

```

lemma tranclp-cdclW-cp-tranclp-cdclW:
  cdclW-cp $^{++} S S' \implies cdcl_W^{++} S S'$ 
  apply (induct rule: tranclp.induct)
  apply (simp add: cdclW-cp-tranclp-cdclW)
  by (meson cdclW-cp-tranclp-cdclW tranclp-trans)

```

```

lemma cdclW-stgy-tranclp-cdclW:
  cdclW-stgy  $S S' \implies cdcl_W^{++} S S'$ 
proof (induct rule: cdclW-stgy.induct)
  case conflict'
  then show ?case
    unfolding full1-def by (simp add: tranclp-cdclW-cp-tranclp-cdclW)
next
  case (other' S' S'')

```

then have $S' = S'' \vee \text{cdcl}_W\text{-cp}^{++} S' S''$
by (*simp add: rtracpl-unfold full-def*)
then show ?case
using *other'* **by** (*meson cdcl_W-ops.other cdcl_W-ops-axioms tracpl.r-into-tracl*
tracpl-cdcl_W-cp-tracpl-cdcl_W tracpl-trans)
qed

lemma *tracpl-cdcl_W-stgy-tracpl-cdcl_W:*
cdcl_W-stgy⁺⁺ S S' \implies cdcl_W⁺⁺ S S'
apply (*induct rule: tracpl.induct*)
using *cdcl_W-stgy-tracpl-cdcl_W* **apply** *blast*
by (*meson cdcl_W-stgy-tracpl-cdcl_W tracpl-trans*)

lemma *rtracpl-cdcl_W-stgy-rtracpl-cdcl_W:*
*cdcl_W-stgy^{**} S S' \implies cdcl_W^{**} S S'*
using *rtracpl-unfold[of cdcl_W-stgy S S'] tracpl-cdcl_W-stgy-tracpl-cdcl_W[of S S']* **by** *auto*

lemma *cdcl_W-o-conflict-is-false-with-level-inv:*
assumes
cdcl_W-o S S' and
lev: cdcl_W-M-level-inv S and
confl-inv: conflict-is-false-with-level S and
n-d: distinct-cdcl_W-state S and
conflicting: cdcl_W-conflicting S
shows *conflict-is-false-with-level S'*
using *assms(1,2)*

proof (*induct rule: cdcl_W-o-induct-lev2*)
case (*resolve L C M D T*) **note** *tr-S = this(1)* **and** *confl = this(2)* **and** *T = this(4)*
have $-L \notin\# D$ **using** *n-d confl unfolding distinct-cdcl_W-state-def distinct-mset-def* **by** *auto*
moreover have $L \notin\# D$
proof (*rule ccontr*)
assume $\neg ?thesis$
moreover have *Propagated L (C + {#L#}) # M \models_{as} CNot D*
using *conflicting confl tr-S unfolding cdcl_W-conflicting-def* **by** *auto*
ultimately have $-L \in \text{lits-of } (\text{Propagated L } (C + \{\#L\# \})) \# M$
using *in-CNot-implies-uminus(2)* **by** *blast*
moreover have *no-dup (Propagated L (C + {#L#})) # M*
using *lev tr-S unfolding cdcl_W-M-level-inv-def* **by** *auto*
ultimately show *False* **unfolding** *lits-of-def* **by** (*metis consistent-interp-def image-eqI*
list.set-intros(1) lits-of-def marked-lit.sel(2) distinctconsistent-interp)
qed

ultimately
have *g-D: get-maximum-level D (Propagated L (C + {#L#})) # M*
 $= \text{get-maximum-level } D \ M$
proof –
have $\forall a f L. ((a::'v) \in f \text{ ' } L) = (\exists l. (l::'v \text{ literal}) \in L \wedge a = f l)$
by *blast*
then show ?thesis
using *get-maximum-level-skip-first[of L D (C + {#L#}) M]* **unfolding** *atms-of-def*
by (*metis (no-types) $\langle - L \notin\# D \rangle \langle L \notin\# D \rangle \text{ atm-of-eq-atm-of mem-set-mset-iff}$*)
qed

{ assume
get-maximum-level D (Propagated L (C + {#L#})) # M = backtrack-lvl S and
backtrack-lvl S > 0

```

then have D: get-maximum-level D M = backtrack-lvl S unfolding g-D by blast
then have ?case
  using tr-S ⟨backtrack-lvl S > 0⟩ get-maximum-level-exists-lit[of backtrack-lvl S D M] T
  by auto
}
moreover {
  assume [simp]: backtrack-lvl S = 0
  have  $\bigwedge L. \text{get-level } L \ M = 0$ 
  proof -
    fix L
    have atm-of L  $\notin$  atm-of ‘ (lits-of M)  $\implies$  get-level L M = 0 by auto
    moreover {
      assume atm-of L  $\in$  atm-of ‘ (lits-of M)
      have g-r: get-all-levels-of-marked M = rev [Suc 0.. $\text{Suc } (\text{backtrack-lvl } S)$ ]
      using lev tr-S unfolding cdclW-M-level-inv-def by auto
      have Max (insert 0 (set (get-all-levels-of-marked M))) = (backtrack-lvl S)
      unfolding g-r by (simp add: Max-n-upt)
      then have get-level L M = 0
      using get-maximum-possible-level-ge-get-level[of L M]
      unfolding get-maximum-possible-level-max-get-all-levels-of-marked by auto
    }
    ultimately show get-level L M = 0 by blast
  qed
  then have ?case using get-maximum-level-exists-lit-of-max-level[of D# $\cup$ C M] tr-S T
  by (auto simp: Bex-mset-def)
}
ultimately show ?case using resolve.hyps(3) by blast
next
case (skip L C' M D T) note tr-S = this(1) and D = this(2) and T = this(5)
then obtain La where La  $\in$ # D and get-level La (Propagated L C' # M) = backtrack-lvl S
  using skip confl-inv by auto
moreover
  have atm-of La  $\neq$  atm-of L
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    then have La: La = L using ⟨La  $\in$ # D⟩  $\langle$   $\neg$  L  $\notin$ # D  $\rangle$  by (auto simp add: atm-of-eq-atm-of)
    have Propagated L C' # M  $\models_{as}$  CNot D
    using conflicting tr-S D unfolding cdclW-conflicting-def by auto
    then have  $\neg$ L  $\in$  lits-of M
    using ⟨La  $\in$ # D⟩ in-CNot-implies-uminus(2)[of D L Propagated L C' # M] unfolding La
    by auto
    then show False using lev tr-S unfolding cdclW-M-level-inv-def consistent-interp-def by auto
  qed
  then have get-level La (Propagated L C' # M) = get-level La M by auto
  ultimately show ?case using D tr-S T by auto
qed (auto split: split-if-asm simp: cdclW-M-level-inv-decomp)

```

17.6.5 Strong completeness

lemma *cdcl_W-cp-propagate-confl*:

assumes *cdcl_W-cp* S T

shows *propagate*** S T \vee ($\exists S'. \text{propagate** } S \ S' \wedge \text{conflict } S' \ T$)

using *assms* by induction blast+

lemma *rtranchp-cdcl_W-cp-propagate-confl*:

assumes *cdcl_W-cp*** S T

shows $\text{propagate}^{**} S T \vee (\exists S'. \text{propagate}^{**} S S' \wedge \text{conflict } S' T)$
by (*simp add: assms rtrancpl-cdcl_W-cp-propa-or-propa-confl*)

lemma *cdcl_W-cp-propagate-completeness:*

assumes $MN: \text{set } M \models_s \text{set-mset } N$ **and**
cons: consistent-interp (set M) **and**
tot: total-over-m (set M) (set-mset N) **and**
lits-of (trail S) \subseteq set M **and**
init-clss S = N **and**
*propagate^{**} S S'* **and**
learned-clss S = {#}
shows $\text{length (trail S)} \leq \text{length (trail S')} \wedge \text{lits-of (trail S')} \subseteq \text{set M}$
using *assms(6,4,5,7)*

proof (*induction rule: rtrancpl-induct*)

case *base*

then show *?case* **by** *auto*

next

case (*step Y Z*)

note $st = \text{this}(1)$ **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 $\text{len: length (trail S)} \leq \text{length (trail Y)}$ **and** $LM: \text{lits-of (trail Y)} \subseteq \text{set M}$
by *blast+*

obtain $M' N' U k C L$ **where**

$Y: \text{state } Y = (M', N', U, k, C\text{-True})$ **and**

$Z: \text{state } Z = (\text{Propagated } L (C + \{\#L\}) \# M', N', U, k, C\text{-True})$ **and**

$C: C + \{\#L\} \in \# \text{ clauses } Y$ **and**

$M'-C: M' \models_{as} C\text{Not } C$ **and**

undefined-lit (trail Y) L

using *propa* **by** *auto*

have $\text{init-clss } S = \text{init-clss } Y$

using *st* **by** *induction auto*

then have [*simp*]: $N' = N$ **using** *NS Y Z* **by** *simp*

have $\text{learned-clss } Y = \{\#\}$

using *st learned* **by** *induction auto*

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 $MN C \text{learned } Y$ **unfolding** *true-clss-def clauses-def*

by (*metis NS $\langle \text{init-clss } S = \text{init-clss } Y \rangle \langle \text{learned-clss } Y = \{\#\} \rangle \text{add.right-neutral}$*
mem-set-mset-iff)

ultimately have $L \in \text{set } M$ **by** (*simp add: cons consistent-CNot-not*)

then show *?case* **using** $LM \text{len } Y Z$ **by** *auto*

qed

lemma *completeness-is-a-full1-propagation:*

fixes $S :: 'st$ **and** $M :: 'v \text{ literal list}$

assumes $MN: \text{set } M \models_s \text{set-mset } N$

and *cons: consistent-interp (set M)*

and *tot: total-over-m (set M) (set-mset N)*

and *alien: no-strange-atm S*

and *learned: learned-clss S = {#}*


```

and clsS[simp]: init-clss S = N
and lits: lits-of (trail S)  $\subseteq$  set M
shows  $\exists S'. \text{propagate}^{**} S S' \wedge \text{full cdcl}_W\text{-cp } S S'$ 
proof -
  obtain S' where full: full cdclW-cp S S'
  using always-exists-full-cdclW-cp-step alien by blast
  then consider (propa) propagate** S S'
  | (confl)  $\exists X. \text{propagate}^{**} S X \wedge \text{conflict } X S'$ 
  using rtrancpl-cdclW-cp-propagate-confl unfolding full-def by blast
  then show ?thesis
  proof cases
    case propa then show ?thesis using full by blast
  next
    case confl
    then obtain X where
      X: propagate** S X and
      Xconf: conflict X S'
    by blast
    have clsX: init-clss X = init-clss S
    using X by induction auto
    have learnedX: learned-clss X = {#} using X learned by induction auto
    obtain E where
      E: E  $\in$  # init-clss X + learned-clss X and
      Not-E: trail X  $\models_{as}$  CNot E
    using Xconf by (auto simp add: conflict.simps clauses-def)
    have lits-of (trail X)  $\subseteq$  set M
    using cdclW-cp-propagate-completeness[OF assms(1-3) lits - X learned] learned by auto
    then have MNE: set M  $\models_s$  CNot E
    using Not-E
    by (fastforce simp add: true-annots-def true-annot-def true-clss-def true-cl-def)
    have  $\neg$  set M  $\models_s$  set-mset N
    using E consistent-CNot-not[OF cons MNE]
    unfolding learnedX true-clss-def unfolding clsX clsS by auto
    then show ?thesis using MN by blast
  qed
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 *rtrancpl-propagate-is-trail-append:*

*propagate^{**} S T $\implies \exists c. \text{trail } T = c @ \text{trail } S$*
by (induction rule: rtrancpl-induct) auto

lemma *rtrancpl-propagate-is-update-trail:*

*propagate^{**} S T $\implies \text{cdcl}_W\text{-M-level-inv } S \implies T \sim \text{delete-trail-and-rebuild } (\text{trail } T) S$*

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 $\langle \text{propagate}^{**} S T \rangle \langle \text{propagate } T U \rangle$

rtrancpl-mono[of propagate cdcl_W] cdcl_W-cp-consistent-inv propagate'

rtrancpl-propagate-is-rtrancpl-cdcl_W step.prem by blast

then have no-dup (trail U) unfolding cdcl_W-M-level-inv-def by auto

ultimately show ?case using $\langle \text{propagate } T U \rangle$ unfolding state-eq-def by fastforce

qed

lemma *cdcl_W-stgy-strong-completeness-n*:

assumes

MN: *set M* \models_s *set-mset N* **and**
cons: *consistent-interp* (*set M*) **and**
tot: *total-over-m* (*set M*) (*set-mset N*) **and**
atm-incl: *atm-of* ' (*set M*) \subseteq *atms-of-msu N* **and**
distM: *distinct M* **and**
length: $n \leq \text{length } M$

shows

$\exists M' k S. \text{length } M' \geq n \wedge$
lits-of $M' \subseteq \text{set } M \wedge$
no-dup $M' \wedge$
 $S \sim \text{update-backtrack-lvl } k (\text{append-trail } (\text{rev } M') (\text{init-state } N)) \wedge$
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$

using *length*

proof (*induction n*)

case 0

have $\text{update-backtrack-lvl } 0 (\text{append-trail } (\text{rev } []) (\text{init-state } N)) \sim \text{init-state } N$
by (*auto simp: state-eq-def simp del: state-simp*)

moreover have

$0 \leq \text{length } []$ **and**
lits-of $[] \subseteq \text{set } M$ **and**
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) (\text{init-state } N)$
and *no-dup* $[]$
by (*auto simp: state-eq-def simp del: state-simp*)

ultimately show ?*case* **using** *state-eq-sym* **by** *blast*

next

case (*Suc n*) **note** *IH* = *this*(1) **and** *n* = *this*(2)

then obtain $M' k S$ **where**

l-M': $\text{length } M' \geq n$ **and**
M': *lits-of* $M' \subseteq \text{set } M$ **and**
n-d[simp]: *no-dup* M' **and**
 $S: S \sim \text{update-backtrack-lvl } k (\text{append-trail } (\text{rev } M') (\text{init-state } N))$ **and**
 $st: \text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$
by *auto*

have

M: *cdcl_W-M-level-inv* S **and**
alien: *no-strange-atm* S
using *rtranclp-cdcl_W-consistent-inv*[*OF rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*[*OF st*]]
rtranclp-cdcl_W-no-strange-atm-inv[*OF rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*[*OF st*]]
 S **unfolding** *state-eq-def cdcl_W-M-level-inv-def no-strange-atm-def* **by** *auto*

{ **assume** *no-step*: $\neg \text{no-step propagate } S$

obtain S' **where** S' : *propagate*^{**} $S S'$ **and** *full*: *full cdcl_W-cp* $S S'$

using *completeness-is-a-full1-propagation*[*OF assms*(1–3), *of S*] *alien* $M' S$ **by** *auto*

have *lev*: *cdcl_W-M-level-inv* S'

using $M S' rtranclp\text{-cdcl}_W\text{-consistent-inv } rtranclp\text{-propagate-is-rtranclp-cdcl}_W$ **by** *blast*

then have *n-d'[simp]*: *no-dup* (*trail* S')

unfolding *cdcl_W-M-level-inv-def* **by** *auto*

have $\text{length } (\text{trail } S) \leq \text{length } (\text{trail } S') \wedge \text{lits-of } (\text{trail } S') \subseteq \text{set } M$

using S' *full cdcl_W-cp-propagate-completeness*[*OF assms*(1–3), *of S*] $M' S$ **by** *auto*

moreover

have *full*: *full1 cdcl_W-cp* $S S'$

```

    using full no-step no-step-cdclW-cp-no-conflict-no-propagate(2) unfolding full1-def full-def
    rtrancpl-unfold by blast
  then have cdclW-stgy  $S S'$  by (simp add: cdclW-stgy.conflict')
moreover
  have propa: propagate++  $S S'$  using  $S'$  full unfolding full1-def by (metis rtrancplD trancplD)
  have trail  $S = M'$  using  $S$  by auto
  with propa have length (trail  $S'$ ) >  $n$ 
    using l- $M'$  propa by (induction rule: trancpl.induct) auto
moreover
  have st $S'$ : cdclW-stgy** (init-state  $N$ )  $S'$ 
    using st cdclW-stgy.conflict'[OF full] by auto
  then have init-clss  $S' = N$  using st $S'$  rtrancpl-cdclW-stgy-no-more-init-clss by fastforce
moreover
  have
    [simp]: learned-clss  $S' = \{\#\}$  and
    [simp]: init-clss  $S' = \text{init-clss } S$  and
    [simp]: conflicting  $S' = C\text{-True}$ 
    using trancpl-into-rtrancpl[OF ⟨propagate++  $S S'$ ⟩]  $S$ 
    rtrancpl-propagate-is-update-trail[of  $S S'$ ]  $S M$  unfolding state-eq-def by simp-all
  have  $S\text{-}S'$ :  $S' \sim \text{update-backtrack-lvl } (\text{backtrack-lvl } S')$ 
    (append-trail (rev (trail  $S'$ )) (init-state  $N$ )) using  $S$ 
    by (auto simp: state-eq-def simp del: state-simp)
  have cdclW-stgy** (init-state (init-clss  $S'$ ))  $S'$ 
    apply (rule rtrancpl.rtrancpl-into-rtrancpl)
    using st unfolding ⟨init-clss  $S' = N$ ⟩ 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  $S\text{-}S'$  by (auto simp: state-eq-def simp del: state-simp)
}
moreover {
  assume no-step: no-step propagate  $S$ 
  have ?case
    proof (cases length  $M' \geq \text{Suc } n$ )
    case True
      then show ?thesis using l- $M'$   $M'$  st  $M$  alien  $S$  by fastforce
    next
    case False
      then have  $n'$ : length  $M' = n$  using l- $M'$  by auto
      have no-conf: no-step conflict  $S$ 
        proof -
          { fix  $D$ 
            assume  $D \in \# N$  and  $M' \models_{as} C\text{Not } D$ 
            then have set  $M \models D$  using MN unfolding true-clss-def by auto
            moreover have set  $M \models_s C\text{Not } D$ 
              using ⟨ $M' \models_{as} C\text{Not } D$ ⟩  $M'$ 
              by (metis le-iff-sup true-annots-true-clss true-clss-union-increase)
            ultimately have False using cons consistent-CNot-not by blast
          }
        then show ?thesis using  $S$  by (auto simp add: conflict.simps true-clss-def)
      qed
    have len $M$ : length  $M = \text{card } (\text{set } M)$  using dist $M$  by (induction  $M$ ) auto
    have no-dup  $M'$  using  $S M$  unfolding cdclW- $M$ -level-inv-def by auto
    then have card (lits-of  $M'$ ) = length  $M'$ 

```

```

    by (induction M') (auto simp add: lits-of-def card-insert-if)
  then have lits-of M'  $\subseteq$  set M
    using n M' n' lenM by auto
  then obtain m where m: m  $\in$  set M and undef-m: m  $\notin$  lits-of M' by auto
  moreover have undef: undefined-lit M' m
    using M' Marked-Propagated-in-iff-in-lits-of calculation(1,2) cons
    consistent-interp-def by blast
  moreover have atm-of m  $\in$  atms-of-msu (init-clss S)
    using atm-incl calculation S by auto
  ultimately
    have dec: decide S (cons-trail (Marked m (k+1)) (incr-lvl S))
      using decide.intros[of S rev M' N - k m
        cons-trail (Marked m (k + 1)) (incr-lvl S)] S
      by auto
    let ?S' = cons-trail (Marked m (k+1)) (incr-lvl S)
    have lits-of (trail ?S')  $\subseteq$  set M using m M' S undef by auto
    moreover have no-strange-atm ?S'
      using alien dec M by (meson 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')  $\leq$  length (trail S'')  $\wedge$  lits-of (trail S'')  $\subseteq$  set M
      using S'' full cdclW-cp-propagate-completeness[OF assms(1-3), of ?S' S''] m M' S undef
      by simp
    then have Suc n  $\leq$  length (trail S'')  $\wedge$  lits-of (trail S'')  $\subseteq$  set M
      using l-M' S undef by auto
    moreover
      have cdclW-M-level-inv (cons-trail (Marked m (Suc (backtrack-lvl S))))
        (update-backtrack-lvl (Suc (backtrack-lvl S)) S))
        using S (cdclW-M-level-inv (cons-trail (Marked m (k + 1)) (incr-lvl S))) by auto
      then have S'': S''  $\sim$  update-backtrack-lvl (backtrack-lvl S'')
        (append-trail (rev (trail S'')) (init-state N))
        using rtranclp-propagate-is-update-trail[OF S''] S undef n-d'' lev''
        by (auto simp del: state-simp simp: state-eq-def )
      then have cdclW-stgy** (init-state N) S''
        using cdclW-stgy.intros(2)[OF decide[OF dec] - full] no-step no-conf st
        by (auto simp: cdclW-cp.simps)
      ultimately show ?thesis using S'' n-d'' by blast
    qed
  }
  ultimately show ?case by blast
qed

```

lemma *cdcl_W-stgy-strong-completeness*:
 assumes MN: set M \models_s set-mset N
 and cons: consistent-interp (set M)
 and tot: total-over-m (set M) (set-mset N)
 and atm-incl: atm-of ' (set M) \subseteq atms-of-msu N
 and distM: distinct M
 shows

$\exists M' k S.$
 $\text{ lits-of } M' = \text{ set } M \wedge$
 $S \sim \text{ update-backtrack-lvl } k (\text{ append-trail } (\text{ rev } M') (\text{ init-state } N)) \wedge$
 $\text{ cdcl}_W\text{-stgy}^{**} (\text{ init-state } N) S \wedge$
 $\text{ final-cdcl}_W\text{-state } S$
proof –
from $\text{ cdcl}_W\text{-stgy-strong-completeness-n}[OF \text{ assms, of length } M]$
obtain $M' k T$ **where**
 $l: \text{ length } M \leq \text{ length } M'$ **and**
 $M'-M: \text{ lits-of } M' \subseteq \text{ set } M$ **and**
 $\text{ no-dup: no-dup } M'$ **and**
 $T: T \sim \text{ update-backtrack-lvl } k (\text{ append-trail } (\text{ rev } M') (\text{ init-state } N))$ **and**
 $\text{ st: cdcl}_W\text{-stgy}^{**} (\text{ init-state } N) T$
by auto
have $\text{ card } (\text{ set } M) = \text{ length } M$ **using** distM **by** $(\text{ simp add: distinct-card })$
moreover
have $\text{ cdcl}_W\text{-M-level-inv } T$
using $\text{ rtrancpl-cdcl}_W\text{-stgy-consistent-inv}[OF \text{ st}] T$ **by auto**
then have $\text{ card } (\text{ set } ((\text{ map } (\lambda l. \text{ atm-of } (\text{ lit-of } l)) M'))) = \text{ length } M'$
using $\text{ distinct-card no-dup}$ **by fastforce**
moreover have $\text{ card } (\text{ lits-of } M') = \text{ card } (\text{ set } ((\text{ map } (\lambda l. \text{ atm-of } (\text{ lit-of } l)) M')))$
using $\text{ no-dup unfolding lits-of-def apply (induction M') by (auto simp add: card-insert-if)}$
ultimately have $\text{ card } (\text{ set } M) \leq \text{ card } (\text{ lits-of } M')$ **using** l **unfolding lits-of-def by auto**
then have $\text{ set } M = \text{ lits-of } M'$
using $M'-M \text{ card-seteq}$ **by blast**
moreover
then have $M' \models_{asm} N$
using $MN \text{ unfolding true-annots-def Ball-def true-annot-def true-clss-def}$ **by auto**
then have $\text{ final-cdcl}_W\text{-state } T$
using $T \text{ no-dup unfolding final-cdcl}_W\text{-state-def}$ **by auto**
ultimately show $?thesis$ **using** $\text{ st } T$ **by blast**
qed

17.6.6 No conflict with only variables of level less than backtrack level

This invariant is stronger than the previous argument in the sense that it is a property about all possible conflicts.

definition $\text{ no-smaller-conflict } (S :: 'st) \equiv$
 $(\forall M K i M' D. M' @ \text{ Marked } K i \# M = \text{ trail } S \longrightarrow D \in \# \text{ clauses } S$
 $\longrightarrow \neg M \models_{as} CNot D)$

lemma $\text{ no-smaller-conflict-init-sate}[simp]:$
 $\text{ no-smaller-conflict } (\text{ init-state } N)$ **unfolding** $\text{ no-smaller-conflict-def}$ **by auto**

lemma $\text{ cdcl}_W\text{-o-no-smaller-conflict-inv}:$
fixes $S S' :: 'st$
assumes
 $\text{ cdcl}_W\text{-o } S S'$ **and**
 $\text{ lev: cdcl}_W\text{-M-level-inv } S$ **and**
 $\text{ max-lev: conflict-is-false-with-level } S$ **and**
 $\text{ smaller: no-smaller-conflict } S$ **and**
 $\text{ no-f: no-clause-is-false } S$
shows $\text{ no-smaller-conflict } S'$
using $\text{ assms}(1,2)$ **unfolding** $\text{ no-smaller-conflict-def}$
proof $(\text{ induct rule: cdcl}_W\text{-o-induct-lev2})$

```

case (decide L T) note confl = this(1) and undef = this(2) and T = this(4)
have [simp]: clauses T = clauses S
  using T undef by auto
show ?case
proof (intro allI impI)
  fix M'' K i M' Da
  assume M'' @ Marked K i # M' = trail T
  and D: Da ∈ # local.clauses T
  then have tl M'' @ Marked K i # M' = trail S
    ∨ (M'' = [] ∧ Marked K i # M' = Marked L (backtrack-lvl S + 1) # trail S)
    using T undef by (cases M'') auto
  moreover {
    assume tl M'' @ Marked K i # M' = trail S
    then have ¬M' ⊨as CNot Da
      using D T undef no-f confl smaller unfolding no-smaller-confl-def smaller by fastforce
  }
  moreover {
    assume Marked K i # M' = Marked L (backtrack-lvl S + 1) # trail S
    then have ¬M' ⊨as CNot Da using no-f D confl T by auto
  }
  ultimately show ¬M' ⊨as CNot Da by fast
qed
next
case resolve
  then show ?case using smaller no-f max-lev unfolding no-smaller-confl-def by auto
next
case skip
  then show ?case using smaller no-f max-lev unfolding no-smaller-confl-def by auto
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and confl = this(3) and undef = this(6)
  and T = this(7)
obtain c where M: trail S = c @ M2 @ Marked K (i+1) # M1
  using decomp by auto

show ?case
proof (intro allI impI)
  fix M ia K' M' Da
  assume M' @ Marked K' ia # M = trail T
  then have tl M' @ Marked K' ia # M = M1
    using T decomp undef lev by (cases M') (auto simp: cdclW-M-level-inv-decomp)
  assume D: Da ∈ # clauses T
  moreover {
    assume Da ∈ # clauses S
    then have ¬M ⊨as CNot Da using (tl M' @ Marked K' ia # M = M1) M confl undef smaller
      unfolding no-smaller-confl-def by auto
  }
  moreover {
    assume Da: Da = D + {#L#}
    have ¬M ⊨as CNot Da
      proof (rule ccontr)
        assume ¬thesis
        then have -L ∈ lits-of M unfolding Da by auto
        then have -L ∈ lits-of (Propagated L ((D + {#L#}))) # M1
          using UnI2 (tl M' @ Marked K' ia # M = M1)
          by auto
      }
  }

```

```

moreover
  have backtrack S
    (cons-trail (Propagated L (D + {#L#}))
      (reduce-trail-to M1 (add-learned-cls (D + {#L#})
        (update-backtrack-lvl i (update-conflicting C-True S))))))
    using backtrack.intros[of S] backtrack.hyps
    by (force simp: state-eq-def simp del: state-simp)
  then have cdclW-M-level-inv
    (cons-trail (Propagated L (D + {#L#}))
      (reduce-trail-to M1 (add-learned-cls (D + {#L#})
        (update-backtrack-lvl i (update-conflicting C-True S))))))
    using cdclW-consistent-inv[OF - lev] other[OF bj] by auto
  then have no-dup (Propagated L (D + {#L#}) # M1)
    using decomp undef lev unfolding cdclW-M-level-inv-def by auto
  ultimately show False by (metis consistent-interp-def distinctconsistent-interp
    insertCI lits-of-cons marked-lit.sel(2))
qed
}
ultimately show  $\neg M \models_{as} CNot\ Da$ 
  using T undef  $\langle Da = D + \{ \#L\# \} \implies \neg M \models_{as} CNot\ Da \rangle$  decomp lev
  unfolding cdclW-M-level-inv-def by fastforce
qed
qed

```

lemma *conflict-no-smaller-conflict-inv*:

```

assumes conflict S S'
and no-smaller-conflict S
shows no-smaller-conflict S'
using assms unfolding no-smaller-conflict-def by fastforce

```

lemma *propagate-no-smaller-conflict-inv*:

```

assumes propagate: propagate S S'
and n-l: no-smaller-conflict S
shows no-smaller-conflict S'
unfolding no-smaller-conflict-def
proof (intro allI impI)
  fix M' K i M'' D
  assume M': M'' @ 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, C-True) and
    S': state S' = (Propagated L ( (C + {#L#} )) # M, N, U, k, C-True) and
    C + {#L#} ∈ # clauses S and
    M  $\models_{as}$  CNot C and
    undefined-lit M L
  using propagate by auto
  have tl M'' @ Marked K i # M' = trail S using M' S S'
  by (metis Pair-inject list.inject list.sel(3) marked-lit.distinct(1) self-append-conv2
    tl-append2)
  then have  $\neg M' \models_{as} CNot\ D$ 
    using  $\langle D \in \# \text{clauses } S' \rangle$  n-l S S' clauses-def unfolding no-smaller-conflict-def by auto
  then show  $\neg M' \models_{as} CNot\ D$  by auto
qed

```

lemma *cdcl_W-cp-no-smaller-conflict-inv*:

```

  assumes propagate:  $cdcl_W\text{-}cp\ S\ S'$ 
  and  $n\text{-}l$ :  $no\text{-}smaller\text{-}confl\ S$ 
  shows  $no\text{-}smaller\text{-}confl\ S'$ 
  using assms
proof (induct rule:  $cdcl_W\text{-}cp.induct$ )
  case ( $conflict'\ S\ S'$ )
  then show ?case using  $conflict\text{-}no\text{-}smaller\text{-}confl\text{-}inv[of\ S\ S']$  by blast
next
  case ( $propagate'\ S\ S'$ )
  then show ?case using  $propagate\text{-}no\text{-}smaller\text{-}confl\text{-}inv[of\ S\ S']$  by fastforce
qed

```

```

lemma  $rtrancl\text{-}cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv$ :
  assumes propagate:  $cdcl_W\text{-}cp^{**}\ S\ S'$ 
  and  $n\text{-}l$ :  $no\text{-}smaller\text{-}confl\ S$ 
  shows  $no\text{-}smaller\text{-}confl\ S'$ 
  using assms
proof (induct rule:  $rtranclp.induct$ )
  case base
  then show ?case by simp
next
  case ( $step\ S'\ S''$ )
  then show ?case using  $cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv[of\ S'\ S'']$  by fast
qed

```

```

lemma  $trancl\text{-}cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv$ :
  assumes propagate:  $cdcl_W\text{-}cp^{++}\ S\ S'$ 
  and  $n\text{-}l$ :  $no\text{-}smaller\text{-}confl\ S$ 
  shows  $no\text{-}smaller\text{-}confl\ S'$ 
  using assms
proof (induct rule:  $tranclp.induct$ )
  case ( $r\text{-}into\text{-}trancl\ S\ S'$ )
  then show ?case using  $cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv[of\ S\ S']$  by blast
next
  case ( $trancl\text{-}into\text{-}trancl\ S\ S'\ S''$ )
  then show ?case using  $cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv[of\ S'\ S'']$  by fast
qed

```

```

lemma  $full\text{-}cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv$ :
  assumes  $full\ cdcl_W\text{-}cp\ S\ S'$ 
  and  $n\text{-}l$ :  $no\text{-}smaller\text{-}confl\ S$ 
  shows  $no\text{-}smaller\text{-}confl\ S'$ 
  using assms unfolding  $full\text{-}def$ 
  using  $rtrancl\text{-}cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv[of\ S\ S']$  by blast

```

```

lemma  $full1\text{-}cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv$ :
  assumes  $full1\ cdcl_W\text{-}cp\ S\ S'$ 
  and  $n\text{-}l$ :  $no\text{-}smaller\text{-}confl\ S$ 
  shows  $no\text{-}smaller\text{-}confl\ S'$ 
  using assms unfolding  $full1\text{-}def$ 
  using  $trancl\text{-}cdcl_W\text{-}cp\text{-}no\text{-}smaller\text{-}confl\text{-}inv[of\ S\ S']$  by blast

```

```

lemma  $cdcl_W\text{-}stgy\text{-}no\text{-}smaller\text{-}confl\text{-}inv$ :
  assumes  $cdcl_W\text{-}stgy\ S\ S'$ 
  and  $n\text{-}l$ :  $no\text{-}smaller\text{-}confl\ S$ 

```



```

and conflict-is-false-with-level S
and cdclW-M-level-inv S
shows no-smaller-conf S'
using assms
proof (induct rule: cdclW-stgy.induct)
case (conflict' S')
then show ?case using full1-cdclW-cp-no-smaller-conf-inv[of S S'] by blast
next
case (other' S' S'')
have no-smaller-conf S'
  using cdclW-o-no-smaller-conf-inv[OF other'.hyps(1) other'.prems(3,2,1)]
  not-conflict-not-any-negated-init-clss other'.hyps(2) by blast
then show ?case using full-cdclW-cp-no-smaller-conf-inv[of S' S''] other'.hyps by blast
qed

```

lemma *conflict-conflict-is-no-clause-is-false-test:*

```

assumes conflict S S'
and (∀ D ∈# init-clss S + learned-clss S. trail S ⊨as CNot D
  → (∃ L. L ∈# D ∧ get-level L (trail S) = backtrack-lvl S))
shows ∀ D ∈# init-clss S' + learned-clss S'. trail S' ⊨as CNot D
  → (∃ L. L ∈# D ∧ get-level L (trail S') = backtrack-lvl S')
using assms by auto

```

lemma *is-conflicting-exists-conflict:*

```

assumes ¬(∀ D ∈# init-clss S' + learned-clss S'. ¬ trail S' ⊨as CNot D)
and conflicting S' = C-True
shows ∃ S''. conflict S' S''
using assms clauses-def not-conflict-not-any-negated-init-clss by fastforce

```

lemma *cdcl_W-o-conflict-is-no-clause-is-false:*

```

fixes S S' :: 'st
assumes
  cdclW-o S S' and
  lev: cdclW-M-level-inv S and
  max-lev: conflict-is-false-with-level S and
  no-f: no-clause-is-false S and
  no-l: no-smaller-conf S
shows no-clause-is-false S'
  ∨ (conflicting S' = C-True
    → (∀ D ∈# clauses S'. trail S' ⊨as CNot D
      → (∃ L. L ∈# D ∧ get-level L (trail S') = backtrack-lvl S')))
using assms(1,2)
proof (induct rule: cdclW-o-induct-lev2)
case (decide L T) note S = this(1) and undef = this(2) and T = this(4)
show ?case
proof (rule HOL.disjI2, clarify)
fix D
assume D: D ∈# clauses T and M-D: trail T ⊨as CNot D
let ?M = trail S
let ?M' = trail T
let ?k = backtrack-lvl S
have ¬?M ⊨as CNot D
  using no-f D S T undef by auto
have -L ∈# D

```

```

proof (rule ccontr)
  assume  $\neg ?thesis$ 
  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  $lits-of\ (Marked\ L\ (?k + 1)\ \# \ ?M) \models_l L''$ 
    using  $M-D\ x\ T\ undef$  unfolding true-annots-def Ball-def true-annot-def CNot-def
    true-cls-def Bex-mset-def by auto
    show  $\exists L \in\# x. lits-of\ ?M \models_l L$  unfolding Bex-mset-def
    by (metis  $\langle -\ L \notin\# D \rangle \langle L'' \in\# x \rangle L' \langle lits-of\ (Marked\ L\ (?k + 1)\ \# \ ?M) \models_l L'' \rangle$ 
    count-single insertE less-numeral-extra(3) lits-of-cons marked-lit.sel(1)
    true-lit-def uminus-of-uminus-id)
  qed
  then show  $False$  using  $\langle \neg ?M \models_{as} CNot\ D \rangle$  by auto
qed
have  $atm-of\ L \notin atm-of\ ' (lits-of\ ?M)$ 
  using undef defined-lit-map unfolding lits-of-def by fastforce
then have  $get-level\ (-L)\ (Marked\ L\ (?k + 1)\ \# \ ?M) = ?k + 1$  by simp
then show  $\exists La. La \in\# D \wedge get-level\ La\ ?M'$ 
   $= backtrack-lvl\ T$ 
  using  $\langle -L \in\# D \rangle\ T\ undef$  by auto
qed
next
  case resolve
  then show  $?case$  by auto
next
  case skip
  then show  $?case$  by auto
next
case (backtrack  $K\ i\ M1\ M2\ L\ D\ T$ ) note  $decomp = this(1)$  and  $undef = this(6)$  and  $T = this(7)$ 
show  $?case$ 
  proof (rule HOL.disjI2, clarify)
    fix  $Da$ 
    assume  $Da: Da \in\# clauses\ T$ 
    and  $M-D: trail\ T \models_{as} CNot\ Da$ 
    obtain  $c$  where  $M: trail\ S = c @ M2 @ Marked\ K\ (i + 1)\ \# M1$ 
    using  $decomp$  by auto
    have  $tr-T: trail\ T = Propagated\ L\ (D + \{\#L\#\})\ \# M1$ 
    using  $T\ decomp\ undef\ lev$  by (auto simp:  $cdcl_W-M-level-inv-decomp$ )
    have  $backtrack\ S\ T$ 
    using backtrack.intros backtrack.hyps  $T$  by (force simp del: state-simp simp: state-eq-def)
    then have  $lev': cdcl_W-M-level-inv\ T$ 
    using  $cdcl_W-consistent-inv\ lev\ other$  by blast
    then have  $-L \notin lits-of\ M1$ 
    unfolding  $cdcl_W-M-level-inv-def\ lits-of-def$ 
    proof -
      have  $consistent-interp\ (lits-of\ (trail\ S)) \wedge no-dup\ (trail\ S)$ 
       $\wedge backtrack-lvl\ S = length\ (get-all-levels-of-marked\ (trail\ S))$ 
       $\wedge get-all-levels-of-marked\ (trail\ S)$ 
       $= rev\ [1..<1 + length\ (get-all-levels-of-marked\ (trail\ S))]$ 
      using  $lev\ cdcl_W-M-level-inv-def$  by blast
    qed
  qed

```

```

    then show  $\neg L \notin \text{lit-of } \text{'set } M1$ 
    by (metis (no-types) One-nat-def add.right-neutral add.Suc-right
        atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set backtrack.hyps(2)
        cdclW-ops.backtrack-lit-skipped cdclW-ops.axioms decomp lits-of-def)
  qed
{ assume  $Da \in \# \text{ clauses } S$ 
  then have  $\neg M1 \models_{as} CNot\ Da$  using no-l M unfolding no-smaller-conflict-def by auto
}
moreover {
  assume  $Da: Da = D + \{\#L\# \}$ 
  have  $\neg M1 \models_{as} CNot\ Da$  using  $\langle \neg L \notin \text{lits-of } M1 \rangle$  unfolding Da by simp
}
ultimately have  $\neg M1 \models_{as} CNot\ Da$ 
  using Da T undef decomp lev by (fastforce simp: cdclW-M-level-inv-decomp)
then have  $\neg L \in \# Da$ 
  using M-D  $\langle \neg L \notin \text{lits-of } M1 \rangle$  in-CNot-implies-uminus(2)
  true-annots-CNot-lit-of-notin-skip T unfolding tr-T
  by (smt insert-iff lits-of-cons marked-lit.sel(2))
have g-M1: get-all-levels-of-marked M1 = rev [1..i+1]
  using lev lev' T decomp undef unfolding cdclW-M-level-inv-def by auto
have no-dup (Propagated L (D + {#L#}) # M1)
  using lev lev' T decomp undef unfolding cdclW-M-level-inv-def by auto
then have L: atm-of L  $\notin$  atm-of 'lits-of M1 unfolding lits-of-def by auto
have get-level ( $\neg L$ ) (Propagated L ((D + {#L#})) # M1) = i
  using get-level-get-rev-level-get-all-levels-of-marked[OF L,
    of [Propagated L ((D + {#L#}))]]
  by (simp add: g-M1 split: if-splits)
then show  $\exists La. La \in \# Da \wedge \text{get-level } La (\text{trail } T) = \text{backtrack-lvl } T$ 
  using  $\langle \neg L \in \# Da \rangle$  T decomp undef lev by (auto simp: cdclW-M-level-inv-def)
qed
qed

```

lemma full1-cdcl_W-cp-exists-conflict-decompose:

```

  assumes confl:  $\exists D \in \# \text{ clauses } S. \text{trail } S \models_{as} CNot\ D$ 
  and full: full cdclW-cp S U
  and no-conflict: conflicting S = C-True
  shows  $\exists T. \text{propagate}^{**} S\ T \wedge \text{conflict } T\ U$ 
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-conflict)
then show ?thesis
proof cases
  case confl
  then show ?thesis by blast
next
  case propa
  then have conflicting U = C-True
    using no-conflict by induction auto
  moreover have [simp]: learned-clss U = learned-clss S and
    [simp]: init-clss U = init-clss S
    using propa by induction auto
  moreover
  obtain D where D:  $D \in \# \text{ clauses } U$  and
    trS: trail S  $\models_{as} CNot\ D$ 

```

```

    using confl clauses-def by auto
  obtain M where M: trail U = M @ trail S
    using full rtrancp-cdclW-cp-dropWhile-trail unfolding full-def by meson
  have tr-U: trail U  $\models_{as}$  CNot D
    apply (rule true-annots-mono)
    using trS unfolding M by simp-all
  have  $\exists V. \text{conflict } U V$ 
    using (conflicting U = C-True) D clauses-def not-conflict-not-any-negated-init-clss tr-U
    by blast
  then have False using full cdclW-cp.conflict' unfolding full-def by blast
  then show ?thesis by fast
qed
qed

```

lemma *full1-cdcl_W-cp-exists-conflict-full1-decompose*:

```

  assumes confl:  $\exists D \in \# \text{clauses } S. \text{trail } S \models_{as} \text{CNot } D$ 
  and full: full cdclW-cp S U
  and no-confl: conflicting S = C-True
  shows  $\exists T D. \text{propagate}^{**} S T \wedge \text{conflict } T U$ 
     $\wedge \text{trail } T \models_{as} \text{CNot } D \wedge \text{conflicting } U = \text{C-Clause } D \wedge D \in \# \text{clauses } S$ 

```

proof –

```

  obtain T where propa: propagate** S T and confl: conflict T U
    using full1-cdclW-cp-exists-conflict-decompose[OF assms] by blast
  have p: learned-clss T = learned-clss S init-clss T = init-clss S
    using propa by induction auto
  have c: learned-clss U = learned-clss T init-clss U = init-clss T
    using confl by induction auto
  obtain D where trail T  $\models_{as}$  CNot D  $\wedge$  conflicting U = C-Clause D  $\wedge$  D  $\in \#$  clauses S
    using confl p c by (fastforce simp: clauses-def)
  then show ?thesis
    using propa confl by blast
qed

```

lemma *cdcl_W-stgy-no-smaller-conf*:

```

  assumes cdclW-stgy S S'
  and n-l: no-smaller-conf S
  and conflict-is-false-with-level S
  and cdclW-M-level-inv S
  and no-clause-is-false S
  and distinct-cdclW-state S
  and cdclW-conflicting S
  shows no-smaller-conf S'
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict' S')
  show no-smaller-conf S'
    using conflict'.hyps conflict'.prems(1) full1-cdclW-cp-no-smaller-conf-inv by blast
next
  case (other' S' S'')
  have lev': cdclW-M-level-inv S'
    using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
  show no-smaller-conf S''
    using cdclW-stgy-no-smaller-conf-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 blast
  then show ?case by blast
next
  case (other'  $S'$   $S''$ )
  have lev': cdclW-M-level-inv  $S'$ 
    using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
  moreover
    have no-clause-is-false  $S'$ 
       $\vee$  (conflicting  $S' = C\text{-True} \longrightarrow (\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{as} C\text{Not } D$ 
         $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S'))$ )
      using cdclW-o-conflict-is-no-clause-is-false[of  $S$   $S'$ ] other'.hyps(1) other'.prems(1-4) by fast
  moreover {
    assume no-clause-is-false  $S'$ 
    {
      assume conflicting  $S' = C\text{-True}$ 
      then have conflict-is-false-with-level  $S'$  by auto
      moreover have full cdclW-cp  $S'$   $S''$ 
        by (metis (no-types) other'.hyps(3))
      ultimately have conflict-is-false-with-level  $S''$ 
        using rtrancpl-cdclW-co-conflict-ex-lit-of-max-level[of  $S'$   $S''$ ] lev' (no-clause-is-false  $S'$ )
        by blast
    }
  }
  moreover
    {
      assume c: conflicting  $S' \neq C\text{-True}$ 
      have conflicting  $S \neq C\text{-True}$  using other'.hyps(1) c
        by (induct rule: cdclW-o-induct) auto
      then have conflict-is-false-with-level  $S'$ 
        using cdclW-o-conflict-is-false-with-level-inv[OF other'.hyps(1)]
        other'.prems(3,5,6,2) by blast
      moreover have cdclW-cp**  $S'$   $S''$  using other'.hyps(3) unfolding full-def by auto
      then have  $S' = S''$  using c
        by (induct rule: rtrancpl-induct)
        (fastforce intro: conflicting-clause.exhaust)+
      ultimately have conflict-is-false-with-level  $S''$  by auto
    }
  }
  ultimately have conflict-is-false-with-level  $S''$  by blast

```

```

}
moreover {
  assume conf: conflicting S' = C-True
  and D-L:  $\forall D \in \# \text{ clauses } S'. \text{ trail } S' \models_{as} CNot D$ 
     $\rightarrow (\exists L. L \in \# D \wedge \text{get-level } L (\text{trail } S') = \text{backtrack-lvl } S')$ 
  { assume  $\forall D \in \# \text{ clauses } S'. \neg \text{ trail } S' \models_{as} CNot D$ 
    then have no-clause-is-false S' using  $\langle \text{conflicting } S' = C-True \rangle$  by simp
    then have conflict-is-false-with-level S'' using calculation(3) by blast
  }
}
moreover {
  assume  $\neg(\forall D \in \# \text{ clauses } S'. \neg \text{ trail } S' \models_{as} CNot D)$ 
  then obtain T D where
    propagate** S' T and
    conflict T S'' and
    D:  $D \in \# \text{ clauses } S'$  and
    trail S''  $\models_{as} CNot D$  and
    conflicting S'' = C-Clause D
    using full1-cdclW-cp-exists-conflict-full1-decompose[OF - -  $\langle \text{conflicting } S' = C-True \rangle$ ]
    other'(3) by (metis (mono-tags, lifting) ball-msetI bex-msetI conflictE state-eq-trail
      trail-update-conflicting)
  obtain M where M: trail S'' = M @ trail S' and nm:  $\forall m \in \text{set } M. \neg \text{is-marked } m$ 
    using rtrancpl-cdclW-cp-dropWhile-trail other'(3) unfolding full-def by meson
  have btS: backtrack-lvl S'' = backtrack-lvl S'
    using other'.hypos(3) unfolding full-def by (metis rtrancpl-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'.hypos(3))
  then have nd: no-dup (trail S'')
    by (metis (no-types) cdclW-M-level-inv-decomp(2))
  have conflict-is-false-with-level S''
  proof cases
    assume trail S'  $\models_{as} CNot D$ 
    moreover then obtain L where  $L \in \# D$  and get-level L (trail S') = backtrack-lvl S'
      using D-L D by blast
    moreover
      have LS':  $-L \in \text{lits-of } (\text{trail } S')$ 
        using  $\langle \text{trail } S' \models_{as} CNot D \rangle \langle L \in \# D \rangle$  in-CNot-implies-uminus(2) by blast
      { fix x ::  $(v, \text{nat}, v \text{ literal multiset}) \text{ marked-lit}$  and
        xb ::  $(v, \text{nat}, v \text{ literal multiset}) \text{ marked-lit}$ 
        assume a1:  $x \in \text{set } (\text{trail } S')$  and
          a2:  $xb \in \text{set } M$  and
          a3:  $(\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } M \cap (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } (\text{trail } S')$ 
             $= \{\}$  and
          a4:  $-L = \text{lit-of } x$  and
          a5:  $\text{atm-of } L = \text{atm-of } (\text{lit-of } xb)$ 
          moreover have  $\text{atm-of } (\text{lit-of } x) = \text{atm-of } L$ 
            using a4 by (metis (no-types) atm-of-uminus)
          ultimately have False
            using a5 a3 a2 a1 by auto
        }
      then have  $\text{atm-of } L \notin \text{atm-of ' lits-of } M$ 
        using nd LS' unfolding M by (auto simp add: lits-of-def)
      then have get-level L (trail S'') = get-level L (trail S')
        unfolding M by (simp add: lits-of-def)
      ultimately show ?thesis using btS  $\langle \text{conflicting } S'' = C-Clause D \rangle$  by auto
    }
  }

```

```

next
  assume  $\neg \text{trail } S' \models_{as} CNot\ D$ 
  then obtain  $L$  where  $L \in \# D$  and  $LM: -L \in \text{lits-of } M$ 
  using  $\langle \text{trail } S'' \models_{as} CNot\ D \rangle$ 
    by (auto simp add: CNot-def true-cls-def M true-annot-def true-annot-def
      split: split-if-asm)
  { fix  $x :: ('v, nat, 'v \text{ literal multiset}) \text{ marked-lit}$  and
     $xb :: ('v, nat, 'v \text{ literal multiset}) \text{ marked-lit}$ 
    assume  $a1: xb \in \text{set } (\text{trail } S')$  and
       $a2: x \in \text{set } M$  and
       $a3: \text{atm-of } L = \text{atm-of } (\text{lit-of } xb)$  and
       $a4: -L = \text{lit-of } x$  and
       $a5: (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } M \cap (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } (\text{trail } S') = \{\}$ 
    moreover have  $\text{atm-of } (\text{lit-of } xb) = \text{atm-of } (-L)$ 
      using  $a3$  by simp
    ultimately have False
      by auto }
  then have  $LS': \text{atm-of } L \notin \text{atm-of ' lits-of } (\text{trail } S')$ 
    using  $nd \langle L \in \# D \rangle LM$  unfolding M by (auto simp add: lits-of-def)
  show ?thesis
    proof cases
      assume  $ne: \text{get-all-levels-of-marked } (\text{trail } S') = []$ 
      have  $\text{backtrack-lvl } S'' = 0$ 
        using  $inv\ ne\ nm$  unfolding  $cdcl_W\text{-}M\text{-level-inv-def } M$ 
        by (simp add: get-all-levels-of-marked-nil-iff-not-is-marked)
      moreover
        have  $a1: \text{get-rev-level } L\ 0\ (\text{rev } M) = 0$ 
          using  $nm$  by auto
        then have  $\text{get-level } L\ (M @ \text{trail } S') = 0$ 
          by (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  $\langle \text{conflicting } S'' = C\text{-Clause } D \rangle \langle L \in \# D \rangle$  unfolding M
          by auto
    next
      assume  $ne: \text{get-all-levels-of-marked } (\text{trail } S') \neq []$ 
      have  $hd (\text{get-all-levels-of-marked } (\text{trail } S')) = \text{backtrack-lvl } S'$ 
        using  $ne\ lev' M\ nm$  unfolding  $cdcl_W\text{-}M\text{-level-inv-def}$ 
        by (cases  $\text{get-all-levels-of-marked } (\text{trail } S')$ )
        (simp-all add: get-all-levels-of-marked-nil-iff-not-is-marked[symmetric])
      moreover have  $\text{atm-of } L \in \text{atm-of ' lits-of } M$ 
        using  $\langle -L \in \text{lits-of } M \rangle$ 
        by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def)
      ultimately show ?thesis
        using  $nm\ ne \langle L \in \# D \rangle \langle \text{conflicting } S'' = C\text{-Clause } D \rangle$ 
        get-level-skip-beginning-hd-get-all-levels-of-marked[OF  $LS'$ , of M]
        get-level-skip-in-all-not-marked[of  $\text{rev } M\ L\ \text{backtrack-lvl } S'$ ]
        unfolding lits-of-def  $btS\ M$ 
        by auto
    qed
  qed
}
ultimately have  $\text{conflict-is-false-with-level } S''$  by blast
}
moreover

```

```

{
  assume conflicting  $S' \neq C\text{-True}$ 
  have no-clause-is-false  $S'$  using  $\langle \text{conflicting } S' \neq C\text{-True} \rangle$  by auto
  then have conflict-is-false-with-level  $S''$  using calculation(3) by blast
}
ultimately show ?case by fast
qed

```

lemma *rtranclp-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  $S' \wedge$  conflict-is-false-with-level  $S'$ 
using assms(1)
proof (induct rule: rtranclp-induct)
  case base
  then show ?case using n-l cls-false by auto
next
  case (step  $S' S''$ ) note st = this(1) and cdcl = this(2) and IH = this(3)
  have no-smaller-confl  $S'$  and conflict-is-false-with-level  $S'$ 
    using IH by blast+
  moreover have cdclW-M-level-inv  $S'$ 
    using st lev rtranclp-cdclW-stgy-rtranclp-cdclW
    by (blast intro: rtranclp-cdclW-consistent-inv)+
  moreover have no-clause-is-false  $S'$ 
    using st no-f rtranclp-cdclW-stgy-not-non-negated-init-clss by blast
  moreover have distinct-cdclW-state  $S'$ 
    using rtranclp-distinct-cdclW-state-inv[of  $S S'$ ] lev rtranclp-cdclW-stgy-rtranclp-cdclW[OF st]
    dist by auto
  moreover have cdclW-conflicting  $S'$ 
    using rtranclp-cdclW-all-inv(6)[of  $S S'$ ] st alien conflicting decomp dist learned lev
    rtranclp-cdclW-stgy-rtranclp-cdclW by blast
  ultimately show ?case
    using cdclW-stgy-no-smaller-confl[OF cdcl] cdclW-stgy-ex-lit-of-max-level[OF cdcl] by fast
qed

```

17.6.7 Final States are Conclusive

lemma *full-cdcl_W-stgy-final-state-conclusive-non-false*:

```

fixes  $S' :: 'st$ 
assumes full: full cdclW-stgy (init-state  $N$ )  $S'$ 
and no-d: distinct-mset-mset  $N$ 
and no-empty:  $\forall D \in \#N. D \neq \{\#\}$ 
shows (conflicting  $S' = C\text{-Clause } \{\#\} \wedge$  unsatisfiable (set-mset (init-clss  $S'$ )))
   $\vee$  (conflicting  $S' = C\text{-True} \wedge$  trail  $S' \models_{asm}$  init-clss  $S'$ )
proof -
  let ?S = init-state  $N$ 
  have

```


termi: $\forall S''. \neg \text{cdcl}_W\text{-stgy } S' S''$ and
step: $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S'$ using *full unfolding full-def* by *auto*
moreover have
learned: $\text{cdcl}_W\text{-learned-clause } S'$ and
level-inv: $\text{cdcl}_W\text{-M-level-inv } S'$ and
alien: *no-strange-atm* S' and
no-dup: *distinct-cdcl_W-state* S' and
conf: $\text{cdcl}_W\text{-conflicting } S'$ and
decomp: *all-decomposition-implies-m* (*init-clss* S') (*get-all-marked-decomposition* (*trail* S'))
using *no-d* *trancpl-cdcl_W-stgy-trancpl-cdcl_W*[*of* $?S S'$] *step* *rtrancpl-cdcl_W-all-inv*(1-6)[*of* $?S S'$]
unfolding *rtrancpl-unfold* by *auto*
moreover
have $\forall D \in \#N. \neg [] \models_{\text{as}} C\text{Not } D$ using *no-empty* by *auto*
then have *conf-k*: *conflict-is-false-with-level* S'
using *rtrancpl-cdcl_W-stgy-no-smaller-conf-inv*[*OF step*] *no-d* by *auto*
show *?thesis*
using *cdcl_W-stgy-final-state-conclusive*[*OF termi decomp learned level-inv alien no-dup conf*
conf-k].
qed

lemma *conflict-is-full1-cdcl_W-cp*:
assumes *cp*: *conflict* $S S'$
shows *full1 cdcl_W-cp* $S S'$
proof –
have $\text{cdcl}_W\text{-cp } S S'$ and *conflicting* $S' \neq C\text{-True}$ using *cp cdcl_W-cp.intros* by *auto*
then have $\text{cdcl}_W\text{-cp}^{++} S S'$ by *blast*
moreover have *no-step cdcl_W-cp* S'
using (*conflicting* $S' \neq C\text{-True}$) by (*metis cdcl_W-cp-conflicting-not-empty*
conflicting-clause.exhaust)
ultimately show *full1 cdcl_W-cp* $S S'$ **unfolding** *full1-def* by *blast+*
qed

lemma *cdcl_W-cp-fst-empty-conflicting-false*:
assumes $\text{cdcl}_W\text{-cp } S S'$
and *trail* $S = []$
and *conflicting* $S \neq C\text{-True}$
shows *False*
using *assms* by (*induct rule: cdcl_W-cp.induct*) *auto*

lemma *cdcl_W-o-fst-empty-conflicting-false*:
assumes $\text{cdcl}_W\text{-o } S S'$
and *trail* $S = []$
and *conflicting* $S \neq C\text{-True}$
shows *False*
using *assms* by (*induct rule: cdcl_W-o.induct*) *auto*

lemma *cdcl_W-stgy-fst-empty-conflicting-false*:
assumes $\text{cdcl}_W\text{-stgy } S S'$
and *trail* $S = []$
and *conflicting* $S \neq C\text{-True}$
shows *False*
using *assms* **apply** (*induct rule: cdcl_W-stgy.induct*)
using *trancplD cdcl_W-cp-fst-empty-conflicting-false* **unfolding** *full1-def* **apply** *metis*
using *cdcl_W-o-fst-empty-conflicting-false* by *blast*

```

thm cdclW-cp.induct[split-format(complete)]

lemma cdclW-cp-conflicting-is-false:
  cdclW-cp S S'  $\implies$  conflicting S = C-Clause {#}  $\implies$  False
  by (induction rule: cdclW-cp.induct) auto

lemma rtrancpl-cdclW-cp-conflicting-is-false:
  cdclW-cp++ S S'  $\implies$  conflicting S = C-Clause {#}  $\implies$  False
  apply (induction rule: trancpl.induct)
  by (auto dest: cdclW-cp-conflicting-is-false)

lemma cdclW-o-conflicting-is-false:
  cdclW-o S S'  $\implies$  conflicting S = C-Clause {#}  $\implies$  False
  by (induction rule: cdclW-o.induct) auto

lemma cdclW-stgy-conflicting-is-false:
  cdclW-stgy S S'  $\implies$  conflicting S = C-Clause {#}  $\implies$  False
  apply (induction rule: cdclW-stgy.induct)
  unfolding full1-def apply (metis (no-types) cdclW-cp-conflicting-not-empty trancplD)
  unfolding full-def by (metis conflict-with-false-implies-terminated other)

lemma rtrancpl-cdclW-stgy-conflicting-is-false:
  cdclW-stgy** S S'  $\implies$  conflicting S = C-Clause {#}  $\implies$  S' = S
  apply (induction rule: rtrancpl.induct)
  apply simp
  using cdclW-stgy-conflicting-is-false by blast

lemma full-cdclW-init-clss-with-false-normal-form:
  assumes
     $\forall m \in \text{set } M. \neg \text{is-marked } m$  and
    E = C-Clause D and
    state S = (M, N, U, 0, E)
    full cdclW-stgy S S' and
    all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))
    cdclW-learned-clause S
    cdclW-M-level-inv S
    no-strange-atm S
    distinct-cdclW-state S
    cdclW-conflicting S
  shows  $\exists M''. \text{state } S' = (M'', N, U, 0, \text{C-Clause } \{ \# \})$ 
  using assms(10,9,8,7,6,5,4,3,2,1)
proof (induction M arbitrary: E D S)
  case Nil
  then show ?case
    using rtrancpl-cdclW-stgy-conflicting-is-false unfolding full-def cdclW-conflicting-def by auto
next
  case (Cons L M) note IH = this(1) and full = this(8) and E = this(10) and inv = this(2-7) and
    S = this(9) and nm = this(11)
  obtain K p where K: L = Propagated K p
  using nm by (cases L) auto
  have every-mark-is-a-conflict S using inv unfolding cdclW-conflicting-def by auto
  then have MpK: M  $\models_{as}$  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  $K': L = \text{Propagated } K \ ( (p - \{\#K\#\}) + \{\#K\#\} )$ 
using  $K$  by auto

consider  $(D) \ D = \{\#\} \mid (D') \ D \neq \{\#\}$  by blast
then show ?case
proof cases
case  $D$ 
then show ?thesis
using full rtrancplp-cdclW-stgy-conflicting-is-false  $S$  unfolding full-def  $E \ D$  by auto
next
case  $D'$ 
then have no-p: no-step propagate  $S$  and no-c: no-step conflict  $S$ 
using  $S \ E$  by auto
then have no-step cdclW-cp  $S$  by (auto simp: cdclW-cp.simps)
have res-skip:  $\exists T. (\text{resolve } S \ T \wedge \text{no-step skip } S \wedge \text{full cdcl}_W\text{-cp } T \ T)$ 
 $\vee (\text{skip } S \ T \wedge \text{no-step resolve } S \wedge \text{full cdcl}_W\text{-cp } T \ T)$ 
proof cases
assume  $\neg \text{lit-of } L \notin D$ 
then obtain  $T$  where  $sk: \text{skip } S \ T$  and  $res: \text{no-step resolve } S$ 
using  $S$  that  $D' \ K$  unfolding skip.simps  $E$  by fastforce
have full cdclW-cp  $T \ T$ 
using  $sk$  by (auto simp add: conflicting-clause-full-cdclW-cp)
then show ?thesis
using  $sk \ res$  by blast
next
assume  $LD: \neg \neg \text{lit-of } L \notin D$ 
then have  $D: C\text{-Clause } D = C\text{-Clause } ((D - \{\# \neg \text{lit-of } L\#\}) + \{\# \neg \text{lit-of } L\#\})$ 
by (auto simp add: multiset-eq-iff)

have  $\bigwedge L. \text{get-level } L \ M = 0$ 
by (simp add: nm)
then have get-maximum-level  $(D - \{\# \neg K\#\})$ 
 $(\text{Propagated } K \ ( (p - \{\#K\#\}) + \{\#K\#\} ) \# M) = 0$ 
using  $LD$  get-maximum-level-exists-lit-of-max-level
proof -
obtain  $L'$  where  $\text{get-level } L' (L\#M) = \text{get-maximum-level } D (L\#M)$ 
using  $LD$  get-maximum-level-exists-lit-of-max-level[of  $D \ L\#M$ ] by fastforce
then show ?thesis by (metis (mono-tags)  $K' \ \text{bex-msetE} \ \text{get-level-skip-all-not-marked}$ 
 $\text{get-maximum-level-exists-lit } nm \ \text{not-gr0}$ )
qed
then obtain  $T$  where  $sk: \text{resolve } S \ T$  and  $res: \text{no-step skip } S$ 
using resolve-rule[of  $S \ K \ p - \{\#K\#\} \ M \ N \ U \ 0 \ (D - \{\# \neg K\#\})$ 
update-conflicting  $(C\text{-Clause } (\text{remdups-mset } (D - \{\# \neg K\#\}) + (p - \{\#K\#\})))$  (tl-trail  $S$ )]
 $S$  unfolding  $K' \ D \ E$  by fastforce
have full cdclW-cp  $T \ T$ 
using  $sk$  by (auto simp add: conflicting-clause-full-cdclW-cp)
then show ?thesis
using  $sk \ res$  by blast
qed
then have step-s:  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$ 
using (no-step cdclW-cp  $S$ ) other' by (meson bj resolve skip)
have get-all-marked-decomposition  $(L \# M) = [([], L\#M)]$ 
using  $nm$  unfolding  $K$  apply (induction  $M$  rule: marked-lit-list-induct, simp)
by (case-tac  $hd$  (get-all-marked-decomposition  $xs$ ), auto)+

```

then have *no-b*: *no-step backtrack S*
using *nm S* **by** *auto*
have *no-d*: *no-step decide S*
using *S E* **by** *auto*

have *full-S-S*: *full cdcl_W-cp S S*
using *S E* **by** (*auto simp add: conflicting-clause-full-cdcl_W-cp*)
then have *no-f*: *no-step (full1 cdcl_W-cp) S*
unfolding *full-def full1-def rtrancpl-unfold* **by** (*meson rtrancplD*)
obtain *T* **where**
s: *cdcl_W-stgy S T* **and** *st*: *cdcl_W-stgy** T S'*
using *full step-s full unfolding full-def* **by** (*metis rtrancpl-unfold rtrancplD*)
have *resolve S T ∨ skip S T*
using *s no-b no-d res-skip full-S-S unfolding cdcl_W-stgy.simps cdcl_W-o.simps full-unfold full1-def*
by (*auto dest!: rtrancplD simp: cdcl_W-bj.simps*)
then obtain *D'* **where** *T*: *state T = (M, N, U, 0, C-Clause D')*
using *S E* **by** *auto*

have *st-c*: *cdcl_W** S T*
using *E T rtrancpl-cdcl_W-stgy-rtrancpl-cdcl_W s* **by** *blast*
have *cdcl_W-conflicting T*
using *rtrancpl-cdcl_W-all-inv(6)[OF st-c inv(6,5,4,3,2,1)]* .
show *?thesis*
apply (*rule IH[of T]*)
using *rtrancpl-cdcl_W-all-inv(6)[OF st-c inv(6,5,4,3,2,1)]* **apply** *blast*
using *rtrancpl-cdcl_W-all-inv(5)[OF st-c inv(6,5,4,3,2,1)]* **apply** *blast*
using *rtrancpl-cdcl_W-all-inv(4)[OF st-c inv(6,5,4,3,2,1)]* **apply** *blast*
using *rtrancpl-cdcl_W-all-inv(3)[OF st-c inv(6,5,4,3,2,1)]* **apply** *blast*
using *rtrancpl-cdcl_W-all-inv(2)[OF st-c inv(6,5,4,3,2,1)]* **apply** *blast*
using *rtrancpl-cdcl_W-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-cdcl_W-stgy-final-state-conclusive-is-one-false*:
fixes *S' :: 'st*
assumes *full*: *full cdcl_W-stgy (init-state N) S'*
and *no-d*: *distinct-mset-mset N*
and *empty*: $\{\#\} \in \# N$
shows *conflicting S' = C-Clause {#} ∧ unsatisfiable (set-mset (init-clss S'))*

proof –
let *?S = init-state N*
have *cdcl_W-stgy** ?S S'* **and** *no-step cdcl_W-stgy S'* **using** *full unfolding full-def* **by** *auto*
then have *plus-or-eq*: *cdcl_W-stgy++ ?S S' ∨ S' = ?S* **unfolding** *rtrancpl-unfold* **by** *auto*
have $\exists S''$. *conflict ?S S''* **using** *empty not-conflict-not-any-negated-init-clss* **by** *force*

then have *cdcl_W-stgy*: $\exists S'$. *cdcl_W-stgy ?S S'*
using *cdcl_W-cp.conflict'[of ?S] conflict-is-full1-cdcl_W-cp cdcl_W-stgy.intros(1)* **by** *metis*
have *S' ≠ ?S* **using** (*no-step cdcl_W-stgy S'*) *cdcl_W-stgy* **by** *blast*

then obtain *St:: 'st* **where** *St*: *cdcl_W-stgy ?S St* **and** *cdcl_W-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 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 = N
  using fullSt cdclW-stgy-no-more-init-clss[OF St] by auto
have conflicting St ≠ C-True
  proof (rule ccontr)
    assume ¬ ?thesis
    then have ∃ T. conflict St T
      using empty cls-St by (fastforce simp: clauses-def)
    then show False using fullSt unfolding full1-def by blast
  qed

have 1: ∀ 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 rtranclp-cdclW-all-inv(1)[OF st] no-d bt by simp
have 4: cdclW-learned-clause St
  using rtranclp-cdclW-all-inv(2)[OF st] no-d bt bt by simp
have 5: cdclW-M-level-inv St
  using rtranclp-cdclW-all-inv(3)[OF st] no-d bt by simp
have 6: no-strange-atm St
  using rtranclp-cdclW-all-inv(4)[OF st] no-d bt by simp
have 7: distinct-cdclW-state St
  using rtranclp-cdclW-all-inv(5)[OF st] no-d bt by simp
have 8: cdclW-conflicting St
  using rtranclp-cdclW-all-inv(6)[OF st] no-d bt by simp
have init-clss S' = init-clss St and conflicting S' = C-Clause {#}
  using ⟨conflicting St ≠ C-True⟩ full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St]
  2 3 4 5 6 7 8 St apply (metis ⟨cdclW-stgy** St S'⟩ rtranclp-cdclW-stgy-no-more-init-clss)
  using ⟨conflicting St ≠ C-True⟩ full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St - -
    S'] 2 3 4 5 6 7 8 by (metis bt conflicting-clause.exhaust prod.inject)

moreover have init-clss S' = N
  using ⟨cdclW-stgy** (init-state N) S'⟩ rtranclp-cdclW-stgy-no-more-init-clss by fastforce
moreover have unsatisfiable (set-mset N)
  by (meson empty mem-set-mset-iff satisfiable-def true-clss-empty true-clss-def)
ultimately show ?thesis by auto
qed

```

```

lemma full-cdclW-stgy-final-state-conclusive:
  fixes  $S' :: 'st$ 
  assumes full: full cdclW-stgy (init-state N) S' and no-d: distinct-mset-mset N
  shows (conflicting S' = C-Clause {#}  $\wedge$  unsatisfiable (set-mset (init-clss S')))
     $\vee$  (conflicting S' = C-True  $\wedge$  trail S'  $\models_{asm}$  init-clss S')
  using assms full-cdclW-stgy-final-state-conclusive-is-one-false
    full-cdclW-stgy-final-state-conclusive-non-false by blast

lemma full-cdclW-stgy-final-state-conclusive-from-init-state:
  fixes  $S' :: 'st$ 
  assumes full: full cdclW-stgy (init-state N) S'
  and no-d: distinct-mset-mset N
  shows (conflicting S' = C-Clause {#}  $\wedge$  unsatisfiable (set-mset N))
     $\vee$  (conflicting S' = C-True  $\wedge$  trail S'  $\models_{asm}$  N  $\wedge$  satisfiable (set-mset N))
proof –
  have N: init-clss S' = N
    using full unfolding full-def by (auto dest: rtranclp-cdclW-stgy-no-more-init-clss)
  consider
    (confl) conflicting S' = C-Clause {#} and unsatisfiable (set-mset (init-clss S'))
  | (sat) conflicting S' = C-True and trail S'  $\models_{asm}$  init-clss S'
  using full-cdclW-stgy-final-state-conclusive[OF assms] by auto
  then show ?thesis
    proof cases
      case confl
      then show ?thesis by (auto simp: N)
    next
      case sat
      have cdclW-M-level-inv (init-state N) by auto
      then have cdclW-M-level-inv S'
        using full rtranclp-cdclW-stgy-consistent-inv unfolding full-def by blast
      then have consistent-interp (lits-of (trail S')) unfolding cdclW-M-level-inv-def by blast
      moreover have lits-of (trail S')  $\models_s$  set-mset (init-clss S')
        using sat(2) by (auto simp add: true-annots-def true-annot-def true-clss-def)
      ultimately have satisfiable (set-mset (init-clss S')) by simp
      then show ?thesis using sat unfolding N by blast
    qed
  qed
end
end
theory CDCL-W-Termination
imports CDCL-W
begin

context cdclW-ops
begin

```

17.7 Termination

The condition that no learned clause is a tautology is overkill (in the sense that the no-duplicate condition is enough), but we can reuse *build-all-simple-clss*.

The invariant contains all the structural invariants that holds,

definition *cdcl_W-all-struct-inv* **where**

```

cdclW-all-struct-inv S =
  (no-strange-atm S  $\wedge$  cdclW-M-level-inv S
     $\wedge$  ( $\forall s \in \#$  learned-clss S.  $\neg$ tautology s)

```

\wedge *distinct-cdcl_W-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 rtrancpl-cdcl_W-all-struct-inv-inv rtrancpl-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: rtrancpl-induct*) (*auto intro: cdcl_W-stgy-cdcl_W-all-struct-inv*)

17.8 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* $S = C\text{-Clause } D$

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(1)* **and** *undef = this(6)* **and** $T = \text{this}(7)$

and

$D \cdot T = \text{this}(9)$ **and** $D \cdot S = \text{this}(10)$

then have $D = C + \{\#L\# \}$

using *not-gr0 lev* **by** (*auto simp: cdcl_W-M-level-inv-decomp if-0-1-ge-0*)

then show *?case*

using T *backtrack.hyps*(1–5) *backtrack.intros* **by** *auto*
qed *auto*

lemma *cdcl_W-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: $D \in \# \text{ learned-clss } T$ **and**
new: $D \notin \# \text{ learned-clss } S$ **and**
cdcl_W: *cdcl_W-stgy* $S \ T$ **and**
lev: *cdcl_W-M-level-inv* S
shows $\exists S'. \text{ backtrack } S \ S' \wedge \text{ cdcl}_W\text{-stgy}^{**} \ S' \ T \wedge \text{ conflicting } S = \text{C-Clause } D$
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*) *rtranclp-cdcl_W-cp-learned-clause-inv*
trancplp-into-rtranclp)
next
case (*other'* $S' \ S''$)
then have $D \in \# \text{ learned-clss } S'$
unfolding *full-def* **by** (*auto* *dest*: *rtranclp-cdcl_W-cp-learned-clause-inv*)
then show ?*case*
using *cdcl_W-o-new-clause-learned-is-backtrack-step*[*OF* - $\langle D \notin \# \text{ learned-clss } S \rangle \langle \text{cdcl}_W\text{-o } S \ S' \rangle$]
 $\langle \text{full cdcl}_W\text{-cp } S' \ S'' \rangle \text{ lev}$ **by** (*metis* *cdcl_W-stgy.conflict'* *full-unfold r-into-rtranclp*
rtranclp.rtrancl-refl)
qed

lemma *rtranclp-cdcl_W-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: $D \in \# \text{ learned-clss } T$ **and**
new: $D \notin \# \text{ learned-clss } S$ **and**
cdcl_W: *cdcl_W-stgy*^{**} $S \ T$ **and**
lev: *cdcl_W-M-level-inv* S
shows $\exists S' \ S''. \text{ cdcl}_W\text{-stgy}^{**} \ S \ S' \wedge \text{ backtrack } S' \ S'' \wedge \text{ conflicting } S' = \text{C-Clause } D \wedge$
 $\text{ cdcl}_W\text{-stgy}^{**} \ S'' \ T$
using *cdcl_W* *learned* *new*
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show ?*case* **by** *blast*
next
case (*step* $T \ U$) **note** $st = \text{this}(1)$ **and** $o = \text{this}(2)$ **and** $IH = \text{this}(3)$ **and**
 $D \cdot U = \text{this}(4)$ **and** $D \cdot S = \text{this}(5)$
show ?*case*
proof (*cases* $D \in \# \text{ learned-clss } T$)
case *True*
then obtain $S' \ S''$ **where**
 st' : *cdcl_W-stgy*^{**} $S \ S'$ **and**
 bt : *backtrack* $S' \ S''$ **and**
 $conf$: *conflicting* $S' = \text{C-Clause } D$ **and**
 st'' : *cdcl_W-stgy*^{**} $S'' \ T$
using $IH \ D \cdot S$ **by** *metis*
then show ?*thesis* **using** o **by** (*meson rtranclp.simps*)
next
case *False*
have *cdcl_W-M-level-inv* T
using *lev rtranclp-cdcl_W-stgy-consistent-inv st* **by** *blast*
then obtain S' **where**
 bt : *backtrack* $T \ S'$ **and**


```

    st': cdclW-stgy** S' U and
    confl: conflicting T = C-Clause D
    using cdclW-cp-new-clause-learned-has-backtrack-step[OF D-U False o]
    by metis
  then have cdclW-stgy** S T and
    backtrack T S' and
    conflicting T = C-Clause D and
    cdclW-stgy** S' U
    using o st by auto
  then show ?thesis by blast
qed
qed

```

lemma *propagate-no-more-Marked-lit*:
 assumes *propagate S S'*
 shows *Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')*
 using *assms* by auto

lemma *conflict-no-more-Marked-lit*:
 assumes *conflict S S'*
 shows *Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')*
 using *assms* by auto

lemma *cdcl_W-cp-no-more-Marked-lit*:
 assumes *cdcl_W-cp S S'*
 shows *Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')*
 using *assms* apply (induct rule: *cdcl_W-cp.induct*)
 using *conflict-no-more-Marked-lit propagate-no-more-Marked-lit* by auto

lemma *rtrancpl-cdcl_W-cp-no-more-Marked-lit*:
 assumes *cdcl_W-cp** S S'*
 shows *Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')*
 using *assms* apply (induct rule: *rtrancpl-induct*)
 using *cdcl_W-cp-no-more-Marked-lit* by blast+

lemma *cdcl_W-o-no-more-Marked-lit*:
 assumes *cdcl_W-o S S'* and *cdcl_W-M-level-inv S* and $\neg \text{decide } S S'$
 shows *Marked K i ∈ set (trail S') ⟶ Marked K i ∈ set (trail S)*
 using *assms*
proof (induct rule: *cdcl_W-o-induct-lev2*)
 case *backtrack* note *decomp = this(1)* and *undef = this(6)* and *T = this(7)* and *lev = this(8)*
 then show ?case
 by (auto simp: *cdcl_W-M-level-inv-decomp*)
next
 case (*decide L T*)
 then show ?case by blast
qed *auto*

lemma *cdcl_W-new-marked-at-beginning-is-decide*:
 assumes *cdcl_W-stgy S S'* and
lev: cdcl_W-M-level-inv S and
trail S' = M' @ Marked L i # M and
trail S = M
 shows $\exists T. \text{decide } S T \wedge \text{no-step } \text{cdcl}_W\text{-cp } S$
 using *assms*

proof (*induct rule: cdcl_W-stgy.induct*)
case (*conflict' S'*) **note** $st = \text{this}(1)$ **and** $no\text{-}dup = \text{this}(2)$ **and** $S' = \text{this}(3)$ **and** $S = \text{this}(4)$
have $cdcl_W\text{-}M\text{-level-inv } S'$
using $full1\text{-}cdcl_W\text{-}cp\text{-consistent-inv } no\text{-}dup \text{ st by blast}$
then have $\text{Marked } L \ i \in \text{set } (\text{trail } S')$ **and** $\text{Marked } L \ i \notin \text{set } (\text{trail } S)$
using $no\text{-}dup \text{ unfolding } S \ S' \ cdcl_W\text{-}M\text{-level-inv-def by (auto simp add: rev-image-eqI)}$
then have *False*
using $st \ rtrancp\text{-}cdcl_W\text{-}cp\text{-no-more-Marked-lit[of } S \ S']$
unfolding $full1\text{-def } rtrancp\text{-unfold by blast}$
then show *?case* **by fast**
next
case (*other' T U*) **note** $o = \text{this}(1)$ **and** $ns = \text{this}(2)$ **and** $st = \text{this}(3)$ **and** $no\text{-}dup = \text{this}(4)$ **and**
 $S' = \text{this}(5)$ **and** $S = \text{this}(6)$
have $cdcl_W\text{-}M\text{-level-inv } U$
by (*metis (full-types) lev cdcl_W.simps cdcl_W-consistent-inv full-def o*
other'.hyps(3) rtrancp-cdcl_W-cp-consistent-inv)
then have $\text{Marked } L \ i \in \text{set } (\text{trail } U)$ **and** $\text{Marked } L \ i \notin \text{set } (\text{trail } S)$
using $no\text{-}dup \text{ unfolding } S \ S' \ cdcl_W\text{-}M\text{-level-inv-def by (auto simp add: rev-image-eqI)}$
then have $\text{Marked } L \ i \in \text{set } (\text{trail } T)$
using $st \ rtrancp\text{-}cdcl_W\text{-}cp\text{-no-more-Marked-lit unfolding full-def by blast}$
then show *?case*
using $cdcl_W\text{-}o\text{-no-more-Marked-lit[OF } o] \langle \text{Marked } L \ i \notin \text{set } (\text{trail } S) \rangle ns \text{ lev by meson}$
qed

lemma *cdcl_W-o-is-decide:*

assumes $cdcl_W\text{-}o \ S' \ T$ **and** $cdcl_W\text{-}M\text{-level-inv } S'$
 $\text{trail } T = \text{drop } (\text{length } M_0) \ M' @ \text{Marked } L \ i \ \# \ H @ M \text{ and}$
 $\neg (\exists M'. \text{trail } S' = M' @ \text{Marked } L \ i \ \# \ H @ M)$
shows *decide S' T*
using *assms*

proof (*induction rule:cdcl_W-o-induct-lev2*)

case (*backtrack K i M1 M2 L D*)
then obtain c **where** $\text{trail } S' = c @ M2 @ \text{Marked } K \ (\text{Suc } i) \ \# \ M1$
by auto
then show *?case*
using *backtrack by (cases drop (length M₀) M') (auto simp: cdcl_W-M-level-inv-def)*

next

case *decide*
show *?case* **using** *decide-rule[of S'] decide(1-4) by auto*

qed *auto*

lemma *rtrancp-cdcl_W-new-marked-at-beginning-is-decide:*

assumes $cdcl_W\text{-stgy}^{**} \ R \ U$ **and**
 $\text{trail } U = M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**
 $\text{trail } R = M$ **and**
 $cdcl_W\text{-}M\text{-level-inv } R$
shows
 $\exists S \ T \ T'. \ cdcl_W\text{-stgy}^{**} \ R \ S \wedge \text{decide } S \ T \wedge \text{cdcl}_W\text{-stgy}^{**} \ T \ U \wedge \text{cdcl}_W\text{-stgy}^{**} \ S \ U \wedge$
 $no\text{-step } cdcl_W\text{-cp } S \wedge \text{trail } T = \text{Marked } L \ i \ \# \ H @ M \wedge \text{trail } S = H @ M \wedge \text{cdcl}_W\text{-stgy } S \ T' \wedge$
 $cdcl_W\text{-stgy}^{**} \ T' \ U$
using *assms*

proof (*induct arbitrary: M H M' i rule: rtrancp-induct*)

case *base*
then show *?case* **by auto**

next

case (*step* T U) **note** $st = \text{this}(1)$ **and** $IH = \text{this}(3)$ **and** $s = \text{this}(2)$ **and**
 $U = \text{this}(4)$ **and** $S = \text{this}(5)$ **and** $lev = \text{this}(6)$
show ?*case*
proof (*cases* $\exists M'. \text{trail } T = M' @ \text{Marked } L \text{ } i \# H @ M$)
case *False*
with s **show** ?*thesis* **using** $U \text{ } s \text{ } st \text{ } S$
proof *induction*
case (*conflict'* W) **note** $cp = \text{this}(1)$ **and** $nd = \text{this}(2)$ **and** $W = \text{this}(3)$
then obtain M_0 **where** $\text{trail } W = M_0 @ \text{trail } T$ **and** $n\text{marked}: \forall l \in \text{set } M_0. \neg \text{is-marked } l$
using *rtranclp-cdcl_W-cp-dropWhile-trail* **unfolding** *full1-def rtranclp-unfold* **by** *meson*
then have $MV: M' @ \text{Marked } L \text{ } i \# H @ M = M_0 @ \text{trail } T$ **unfolding** W **by** *simp*
then have $V: \text{trail } T = \text{drop } (\text{length } M_0) (M' @ \text{Marked } L \text{ } i \# H @ M)$
by *auto*
have $\text{takeWhile } (\text{Not } o \text{ is-marked}) M' = M_0 @ \text{takeWhile } (\text{Not } o \text{ is-marked}) (\text{trail } T)$
using *arg-cong[OF MV, of takeWhile (Not o is-marked)] nmarked*
by (*simp add: takeWhile-tail*)
from *arg-cong[OF this, of length]* **have** $\text{length } M_0 \leq \text{length } M'$
unfolding *length-append by (metis (no-types, lifting) Nat.le-trans le-add1 length-takeWhile-le)*
then have *False* **using** $nd \text{ } V$ **by** *auto*
then show ?*case* **by** *fast*
next
case (*other'* $T' \text{ } U$) **note** $o = \text{this}(1)$ **and** $ns = \text{this}(2)$ **and** $cp = \text{this}(3)$ **and** $nd = \text{this}(4)$
and $U = \text{this}(5)$ **and** $st = \text{this}(6)$
obtain M_0 **where** $\text{trail } U = M_0 @ \text{trail } T'$ **and** $n\text{marked}: \forall l \in \text{set } M_0. \neg \text{is-marked } l$
using *rtranclp-cdcl_W-cp-dropWhile-trail cp* **unfolding** *full-def* **by** *meson*
then have $MV: M' @ \text{Marked } L \text{ } i \# H @ M = M_0 @ \text{trail } T'$ **unfolding** U **by** *simp*
then have $V: \text{trail } T' = \text{drop } (\text{length } M_0) (M' @ \text{Marked } L \text{ } i \# H @ M)$
by *auto*
have $\text{takeWhile } (\text{Not } o \text{ is-marked}) M' = M_0 @ \text{takeWhile } (\text{Not } o \text{ is-marked}) (\text{trail } T')$
using *arg-cong[OF MV, of takeWhile (Not o is-marked)] nmarked*
by (*simp add: takeWhile-tail*)
from *arg-cong[OF this, of length]* **have** $\text{length } M_0 \leq \text{length } M'$
unfolding *length-append by (metis (no-types, lifting) Nat.le-trans le-add1 length-takeWhile-le)*
then have $tr\text{-}T': \text{trail } T' = \text{drop } (\text{length } M_0) M' @ \text{Marked } L \text{ } i \# H @ M$ **using** V **by** *auto*
then have $LT': \text{Marked } L \text{ } i \in \text{set } (\text{trail } T')$ **by** *auto*
moreover
have *cdcl_W-M-level-inv* T
using *lev rtranclp-cdcl_W-stgy-consistent-inv step.hyps(1)* **by** *blast*
then have *decide* $T \text{ } T'$ **using** $o \text{ } nd \text{ } tr\text{-}T' \text{ } cdcl_W\text{-o-is-decide}$ **by** *metis*
ultimately **have** *decide* $T \text{ } T'$ **using** *cdcl_W-o-no-more-Marked-lit[OF o]* **by** *blast*
then have $1: cdcl_W\text{-stgy}^{**} R \text{ } T$ **and** $2: \text{decide } T \text{ } T'$ **and** $3: cdcl_W\text{-stgy}^{**} T' \text{ } U$
using $st \text{ } other'.prems(4)$
by (*metis cdcl_W-stgy.conflict' cp full-unfold r-into-rtranclp rtranclp.rtrancl-refl*)
have [*simp*]: $\text{drop } (\text{length } M_0) M' = []$
using $\langle \text{decide } T \text{ } T' \rangle \langle \text{Marked } L \text{ } i \in \text{set } (\text{trail } T') \rangle \text{ } nd \text{ } tr\text{-}T'$
by (*auto simp add: Cons-eq-append-conv*)
have $T': \text{drop } (\text{length } M_0) M' @ \text{Marked } L \text{ } i \# H @ M = \text{Marked } L \text{ } i \# \text{trail } T$
using $\langle \text{decide } T \text{ } T' \rangle \langle \text{Marked } L \text{ } i \in \text{set } (\text{trail } T') \rangle \text{ } nd \text{ } tr\text{-}T'$
by *auto*
have $\text{trail } T' = \text{Marked } L \text{ } i \# \text{trail } T$
using $\langle \text{decide } T \text{ } T' \rangle \langle \text{Marked } L \text{ } i \in \text{set } (\text{trail } T') \rangle \text{ } tr\text{-}T'$
by *auto*
then have $5: \text{trail } T' = \text{Marked } L \text{ } i \# H @ M$

```

    using append.simps(1) list.sel(3) local.other'(5) tl-append2 by (simp add: tr-T')
  have 6: trail T = H @ M
  by (metis (no-types) ⟨trail T' = Marked L i # trail T⟩
    ⟨trail T' = drop (length M0) M' @ Marked L i # H @ M⟩ append-Nil list.sel(3) nd
    tl-append2)
  have 7: cdclW-stgy** T U using other'.prems(4) st by auto
  have 8: cdclW-stgy T U cdclW-stgy** U U
    using cdclW-stgy.other'[OF other'(1-3)] by simp-all
  show ?case apply (rule exI[of - T], rule exI[of - T], rule exI[of - U])
    using ns 1 2 3 5 6 7 8 by fast
qed
next
case True
then obtain M' where T: trail T = M' @ Marked L i # H @ M by metis
from IH[OF this S lev] obtain S' S'' S''' where
  1: cdclW-stgy** R S' and
  2: decide S' S'' and
  3: cdclW-stgy** S'' T and
  4: no-step cdclW-cp S' and
  6: trail S'' = Marked L i # H @ M and
  7: trail S' = H @ M and
  8: cdclW-stgy** S' T and
  9: cdclW-stgy S' S''' and
  10: cdclW-stgy** S''' T
  by blast
  have cdclW-stgy** S'' U using s ⟨cdclW-stgy** S'' T⟩ by auto
  moreover have cdclW-stgy** S' U using 8 s by auto
  moreover have cdclW-stgy** S''' U using 10 s by auto
  ultimately show ?thesis apply - apply (rule exI[of - S'], rule exI[of - S'])
    using 1 2 4 6 7 8 9 by blast
qed
qed

lemma rtrancp-cdclW-new-marked-at-beginning-is-decide':
  assumes cdclW-stgy** R U and
  trail U = M' @ Marked L i # H @ M and
  trail R = M and
  cdclW-M-level-inv R
  shows ∃ y y'. cdclW-stgy** R y ∧ cdclW-stgy y y' ∧ ¬ (∃ c. trail y = c @ Marked L i # H @ M)
    ∧ (λ a b. cdclW-stgy a b ∧ (∃ c. trail a = c @ Marked L i # H @ M))** y' U
proof -
  fix T'
  obtain S' T T' where
    st: cdclW-stgy** R S' and
    decide S' T and
    TU: cdclW-stgy** T U and
    no-step cdclW-cp S' and
    trT: trail T = Marked L i # H @ M and
    trS': trail S' = H @ M and
    S'U: cdclW-stgy** S' U and
    S'T': cdclW-stgy S' T' and
    T'U: cdclW-stgy** T' U
    using rtrancp-cdclW-new-marked-at-beginning-is-decide[OF assms] by blast
  have n: ¬ (∃ c. trail S' = c @ Marked L i # H @ M) using trS' by auto
  show ?thesis

```

using *rtranclp-trans*[*OF st*] *rtranclp-exists-last-with-prop*[*of cdcl_W-stgy S' T' -*
 $\lambda a \cdot \neg(\exists c. \text{trail } a = c @ \text{Marked } L \ i \ \# \ H @ M), \text{OF } S'T' T'U \ n]$
by *meson*
qed

lemma *beginning-not-marked-invert*:

assumes *A: M @ A = M' @ Marked K i # H and*

nm: $\forall m \in \text{set } M. \neg \text{is-marked } m$

shows $\exists M. A = M @ \text{Marked } K \ i \ \# \ H$

proof –

have $A = \text{drop } (\text{length } M) (M' @ \text{Marked } K \ i \ \# \ H)$

using *arg-cong*[*OF A, of drop (length M)*] **by** *auto*

moreover have $\text{drop } (\text{length } M) (M' @ \text{Marked } K \ i \ \# \ H) = \text{drop } (\text{length } M) M' @ \text{Marked } K \ i \ \# \ H$

using *nm* **by** (*metis* (*no-types, lifting*) *A drop-Cons' drop-append marked-lit.disc(1) not-gr0*
 $\text{nth-append nth-append-length nth-mem zero-less-diff}$)

finally show *?thesis* **by** *fast*

qed

lemma *cdcl_W-stgy-trail-has-new-marked-is-decide-step*:

assumes *cdcl_W-stgy S T*

$\neg (\exists c. \text{trail } S = c @ \text{Marked } L \ i \ \# \ H @ M)$ **and**

$(\lambda a \ b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L \ i \ \# \ H @ M))^* T \ U$ **and**

$\exists M'. \text{trail } U = M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**

lev: cdcl_W-M-level-inv S

shows $\exists S'. \text{decide } S \ S' \wedge \text{full } \text{cdcl}_W\text{-cp } S' \ T \wedge \text{no-step } \text{cdcl}_W\text{-cp } S$

using *assms(3,1,2,4,5)*

proof *induction*

case (*step T U*)

then show *?case* **by** *fastforce*

next

case *base*

then show *?case*

proof (*induction rule: cdcl_W-stgy.induct*)

case (*conflict' T*) **note** *cp = this(1)* **and** *nd = this(2)* **and** *M' = this(3)* **and** *no-dup = this(3)*

then obtain *M'* **where** *M': trail T = M' @ Marked L i # H @ M* **by** *metis*

obtain *M''* **where** *M'': trail T = M'' @ trail S and nm: $\forall m \in \text{set } M''. \neg \text{is-marked } m$*

using *cp unfolding full1-def*

by (*metis rtranclp-cdcl_W-cp-dropWhile-trail' tranclp-into-rtranclp*)

have *False*

using *beginning-not-marked-invert*[*of M'' trail S M' L i H @ M*] *M' nm nd unfolding M''*

by *fast*

then show *?case* **by** *fast*

next

case (*other' T U'*) **note** *o = this(1)* **and** *ns = this(2)* **and** *cp = this(3)* **and** *nd = this(4)*

and *trU' = this(5)*

have *cdcl_W-cp** T U'* **using** *cp unfolding full-def* **by** *blast*

from *rtranclp-cdcl_W-cp-dropWhile-trail*[*OF this*]

have $\exists M'. \text{trail } T = M' @ \text{Marked } L \ i \ \# \ H @ M$

using *trU' beginning-not-marked-invert*[*of - trail T - L i H @ M*] **by** *metis*

then obtain *M'* **where** *M': trail T = M' @ Marked L i # H @ M*

by *auto*

with *o lev nd cp ns*

show *?case*

proof (*induction rule: cdcl_W-o-induct-lev2*)

case (*decide L*) **note** *dec = this(1)* **and** *cp = this(5)* **and** *ns = this(4)*

```

then have decide  $S$  (cons-trail (Marked  $L$  (backtrack-lvl  $S + 1$ )) (incr-lvl  $S$ ))
  using decide.hyps decide.intros[of  $S$ ] by force
then show ?case using cp decide.premis by (meson decide-state-eq-compatible ns state-eq-ref
  state-eq-sym)
next
  case (backtrack  $K j M1 M2 L' D T$ ) note decomp = this(1) and cp = this(3)
    and undef = this(6) and  $T = \text{this}(7)$  and trT = this(12) and ns = this(4)
  obtain MS3 where MS3: trail  $S = MS3 @ M2 @ \text{Marked } K (\text{Suc } j) \# M1$ 
    using get-all-marked-decomposition-exists-prepend[OF decomp] by metis
  have tl ( $M' @ \text{Marked } L i \# H @ M$ ) = tl  $M' @ \text{Marked } L i \# H @ M$ 
    using lev trT  $T$  lev undef decomp by (cases  $M'$ ) (auto simp: cdclW-M-level-inv-decomp)
  then have  $M''$ :  $M1 = \text{tl } M' @ \text{Marked } L i \# H @ M$ 
    using arg-cong[OF trT[simplified], of tl]  $T$  decomp undef lev
    by (simp add: cdclW-M-level-inv-decomp)
  have False using nd MS3  $T$  undef decomp unfolding  $M''$  by auto
  then show ?case by fast
qed auto
qed
qed

```

lemma rtranclp-cdcl_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: rtranclp-induct) auto

```

lemma cdcl_W-o-cannot-learn:

```

assumes
  cdclW-o  $y z$  and
  lev: cdclW-M-level-inv  $y$  and
  trM: trail  $y = c @ \text{Marked } Kh i \# H$  and
  DL:  $D + \{\#L\# \} \notin \text{learned-clss } y$  and
  DH:  $\text{atms-of } D \subseteq \text{atm-of 'lits-of } H$  and
  LH:  $\text{atm-of } L \notin \text{atm-of 'lits-of } H$  and
  learned:  $\forall T. \text{conflicting } y = C\text{-Clause } T \longrightarrow \text{trail } y \models_{\text{as}} C\text{Not } T$  and
  z: trail  $z = c' @ \text{Marked } Kh i \# H$ 
shows  $D + \{\#L\# \} \notin \text{learned-clss } z$ 
using assms(1-2) trM DL DH LH learned  $z$ 
proof (induction rule: cdclW-o-induct-lev2)
  case (backtrack  $K j M1 M2 L' D' T$ ) note decomp = this(1) and confl = this(3) and levD = this(5)
    and undef = this(6) and  $T = \text{this}(7)$ 
  obtain M3 where M3: trail  $y = M3 @ M2 @ \text{Marked } K (\text{Suc } j) \# M1$ 
    using decomp get-all-marked-decomposition-exists-prepend by metis
  have  $M$ : trail  $y = c @ \text{Marked } Kh i \# H$  using trM by simp
  have  $H$ : get-all-levels-of-marked (trail  $y$ ) = rev [ $1..<1 + \text{backtrack-lvl } y$ ]
    using lev unfolding cdclW-M-level-inv-def by auto
  have  $c' @ \text{Marked } Kh i \# H = \text{Propagated } L' (D' + \{\#L\# \}) \# \text{trail } (\text{reduce-trail-to } M1 y)$ 
    using backtrack.premis(6) decomp undef  $T$  lev by (force simp: cdclW-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  $D + \{\#L\# \} \in \# \text{ learned-clss } T$ 
then have  $DLD': D + \{\#L\# \} = D' + \{\#L'\# \}$ 
  using  $DL \ T \text{ neq0-conv undef decomp lev by (fastforce simp: cdcl}_W\text{-M-level-inv-def)}$ 
have  $L\text{-cKh: atm-of } L \in \text{atm-of 'lits-of } (c @ [\text{Marked } Kh \ i])$ 
  using  $LH \text{ learned } M \ DLD'[\text{symmetric}] \text{ confl by (fastforce simp add: image-iff)}$ 
have  $\text{get-all-levels-of-marked } (M3 @ M2 @ \text{Marked } K \ (j + 1) \# M1)$ 
   $= \text{rev } [1..<1 + \text{backtrack-lvl } y]$ 
  using  $\text{lev unfolding cdcl}_W\text{-M-level-inv-def } M3 \text{ by auto}$ 
from  $\text{arg-cong}[OF \text{ this, of } \lambda a. (\text{Suc } j) \in \text{set } a] \text{ have backtrack-lvl } y \geq j \text{ by auto}$ 

have  $DD'[\text{simp}]: D = D'$ 
proof (rule ccontr)
  assume  $D \neq D'$ 
  then have  $L' \in \# \ D \text{ using } DLD' \text{ by (metis add.left-neutral count-single count-union}$ 
     $\text{diff-union-cancelR neq0-conv union-single-eq-member)}$ 
  then have  $\text{get-level } L' \ (\text{trail } y) \leq \text{get-maximum-level } D \ (\text{trail } y)$ 
    using  $\text{get-maximum-level-ge-get-level by blast}$ 
  moreover {
    have  $\text{get-maximum-level } D \ (\text{trail } y) = \text{get-maximum-level } D \ H$ 
      using  $DH \text{ unfolding } M \text{ by (simp add: get-maximum-level-skip-beginning)}$ 
    moreover
      have  $\text{get-all-levels-of-marked } (\text{trail } y) = \text{rev } [1..<1 + \text{backtrack-lvl } y]$ 
        using  $\text{lev unfolding cdcl}_W\text{-M-level-inv-def by auto}$ 
      then have  $\text{get-all-levels-of-marked } H = \text{rev } [1..< i]$ 
        unfolding  $M \text{ by (auto dest: append-cons-eq-upt-length-i}$ 
           $\text{simp add: rev-swap[symmetric])}$ 
      then have  $\text{get-maximum-possible-level } H < i$ 
        using  $\text{get-maximum-possible-level-max-get-all-levels-of-marked[of } H] \ \langle i > 0 \rangle \text{ by auto}$ 
      ultimately have  $\text{get-maximum-level } D \ (\text{trail } y) < i$ 
        by (metis (full-types) dual-order.strict-trans nat-neq-iff not-le
           $\text{get-maximum-possible-level-ge-get-maximum-level}) \}$ 
    moreover
      have  $L \in \# \ D'$ 
        by (metis  $DLD' \ \langle D \neq D' \rangle \text{ add.left-neutral count-single count-union diff-union-cancelR}$ 
           $\text{neq0-conv union-single-eq-member})$ 
      then have  $\text{get-maximum-level } D' \ (\text{trail } y) \geq \text{get-level } L \ (\text{trail } y)$ 
        using  $\text{get-maximum-level-ge-get-level by blast}$ 
    moreover {
      have  $\text{get-all-levels-of-marked } (c @ [\text{Marked } Kh \ i]) = \text{rev } [i..< \text{backtrack-lvl } y + 1]$ 
        using  $\text{append-cons-eq-upt-length-i-end[of rev (get-all-levels-of-marked } H) \ i}$ 
           $\text{rev (get-all-levels-of-marked } c) \text{ Suc } 0 \text{ Suc (backtrack-lvl } y)] \ H$ 
        unfolding  $M \text{ apply (auto simp add: rev-swap[symmetric])}$ 
        by (metis (no-types, hide-lams) Nil-is-append-conv Suc-le-eq less-Suc-eq list.sel(1)
           $\text{rev.simps(2) rev-rev-ident upt-Suc upt-rec})$ 
      have  $\text{get-level } L \ (\text{trail } y) = \text{get-level } L \ (c @ [\text{Marked } Kh \ i])$ 
        using  $L\text{-cKh } LH \text{ unfolding } M \text{ by simp}$ 
      have  $\text{get-level } L \ (c @ [\text{Marked } Kh \ i]) \geq i$ 
        using  $L\text{-cKh}$ 
         $\langle \text{get-all-levels-of-marked } (c @ [\text{Marked } Kh \ i]) = \text{rev } [i..<\text{backtrack-lvl } y + 1] \rangle$ 
         $\text{backtrack.hyps(2) calculation(1,2) by auto}$ 
      then have  $\text{get-level } L \ (\text{trail } y) \geq i$ 
        using  $M \ \langle \text{get-level } L \ (\text{trail } y) = \text{get-level } L \ (c @ [\text{Marked } Kh \ i]) \rangle \text{ by auto } \}$ 
    moreover have  $\text{get-maximum-level } D' \ (\text{trail } y) < \text{get-level } L' \ (\text{trail } y)$ 
      using  $\langle j \leq \text{backtrack-lvl } y \rangle \text{ backtrack.hyps(2,5) calculation(1-4) by linarith}$ 
  }

```

```

ultimately show False using backtrack.hyps(4) by linarith
qed
then have LL': L = L' using DLD' by auto
have nd: no-dup (trail y) using lev unfolding cdclW-M-level-inv-def by auto

{ assume D: D' = {#}
  then have j: j = 0 using levD by auto
  have  $\forall m \in \text{set } M1. \neg \text{is-marked } m$ 
    using H unfolding M3 j
    by (auto simp add: rev-swap[symmetric] get-all-levels-of-marked-no-marked
        dest!: append-cons-eq-upt-length-i)
  then have False using d by auto
}
moreover {
  assume D[simp]: D'  $\neq$  {#}
  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''  $\in$  #D' and
    L''D': get-level L'' (trail y) = get-maximum-level D' (trail y)
    using get-maximum-level-exists-lit-of-max-level[OF D, of trail y] by auto
  have L''M: atm-of L''  $\in$  atm-of ' lits-of (trail y)
    using get-rev-level-ge-0-atm-of-in[of 0 L'' rev (trail y)]  $\langle j > 0 \rangle$  levD L''D' by auto
  then have L''  $\in$  lits-of (Marked Kh i # d)
  proof -
    {
      assume L''H: atm-of L''  $\in$  atm-of ' lits-of H
      have get-all-levels-of-marked H = rev [1..i]
        using H unfolding M
        by (auto simp add: rev-swap[symmetric] dest!: append-cons-eq-upt-length-i)
      moreover have get-level L'' (trail y) = get-level L'' H
        using L''H unfolding M by simp
      ultimately have False
        using levD  $\langle j > 0 \rangle$  get-rev-level-in-levels-of-marked[of L'' 0 rev H]  $\langle i \leq j \rangle$ 
        unfolding L''D'[symmetric] nd by auto
    }
  then show ?thesis
    using DD' DH  $\langle L'' \in \# D' \rangle$  atm-of-lit-in-atms-of contra-subsetD by metis
  qed
  then have False
    using DH  $\langle L'' \in \# D' \rangle$  nd unfolding M3 d
    by (auto simp add: atms-of-def image-iff image-subset-iff lits-of-def)
}
ultimately show False by blast
qed
qed auto

```

lemma cdcl_W-stgy-with-trail-end-has-not-been-learned:

assumes cdcl_W-stgy y z and
 cdcl_W-M-level-inv y and
 trail y = c @ Marked Kh i # H and

$D + \{\#L\# \} \notin \text{learned-clss } y$ and
 $DH: \text{atms-of } D \subseteq \text{atm-of 'lits-of } H$ and
 $LH: \text{atm-of } L \notin \text{atm-of 'lits-of } H$ and
 $\forall T. \text{conflicting } y = C\text{-Clause } T \longrightarrow \text{trail } y \models_{as} C\text{Not } T$ and
 $\text{trail } z = c' @ \text{Marked } K i \# H$
shows $D + \{\#L\# \} \notin \text{learned-clss } z$
using *assms*
proof *induction*
case *conflict'*
then show *?case*
unfolding *full1-def* **using** *trancpl-cdcl_W-cp-learned-clause-inv* **by** *auto*
next
case (*other'* $T U$) **note** $o = \text{this}(1)$ and $cp = \text{this}(3)$ and $lev = \text{this}(4)$ and $trY = \text{this}(5)$ and
 $notin = \text{this}(6)$ and $DH = \text{this}(7)$ and $LH = \text{this}(8)$ and $confl = \text{this}(9)$ and $trU = \text{this}(10)$
obtain c' **where** $c': \text{trail } T = c' @ \text{Marked } K i \# H$
using *cp beginning-not-marked-invert[of - trail T c' Kh i H]*
 $rtrancpl-cdcl_W\text{-cp-dropWhile-trail[of } T U]$ **unfolding** *trU full-def* **by** *fastforce*
show *?case*
using *cdcl_W-o-cannot-learn[OF o lev trY notin DH LH confl c']*
 $rtrancpl-cdcl_W\text{-cp-learned-clause-inv cp}$ **unfolding** *full-def* **by** *auto*
qed

lemma *rtrancpl-cdcl_W-stgy-with-trail-end-has-not-been-learned:*
assumes $(\lambda a b. cdcl_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Marked } K i \# H @ []))^{**} S z$ and
 $cdcl_W\text{-all-struct-inv } S$ and
 $\text{trail } S = c @ \text{Marked } K i \# H$ and
 $D + \{\#L\# \} \notin \text{learned-clss } S$ and
 $DH: \text{atms-of } D \subseteq \text{atm-of 'lits-of } H$ and
 $LH: \text{atm-of } L \notin \text{atm-of 'lits-of } H$ and
 $\exists c'. \text{trail } z = c' @ \text{Marked } K i \# H$
shows $D + \{\#L\# \} \notin \text{learned-clss } z$
using *assms(1-4,7)*
proof (*induction rule: rtrancpl-induct*)
case *base*
then show *?case* **by** *auto[1]*
next
case (*step* $T U$) **note** $st = \text{this}(1)$ and $s = \text{this}(2)$ and $IH = \text{this}(3)[OF \text{this}(4-6)]$
and $lev = \text{this}(4)$ and $trS = \text{this}(5)$ and $DL\text{-}S = \text{this}(6)$ and $trU = \text{this}(7)$
obtain c **where** $c: \text{trail } T = c @ \text{Marked } K i \# H$ **using** s **by** *auto*
obtain c' **where** $c': \text{trail } U = c' @ \text{Marked } K i \# H$ **using** trU **by** *blast*
have $cdcl_W^{**} S T$
proof –
have $\forall p pa. \exists s sa. \forall sb sc sd se. (\neg p^{**} (sb::'st) sc \vee p s sa \vee pa^{**} sb sc)$
 $\wedge (\neg pa s sa \vee \neg p^{**} sd se \vee pa^{**} sd se)$
by (*metis (no-types) mono-rtrancpl*)
then have $cdcl_W\text{-stgy}^{**} S T$
using st **by** *blast*
then show *?thesis*
using *rtrancpl-cdcl_W-stgy-rtrancpl-cdcl_W* **by** *blast*
qed
then have $lev': cdcl_W\text{-all-struct-inv } T$
using *rtrancpl-cdcl_W-all-struct-inv-inv[of S T]* lev **by** *auto*
then have $confl': \forall Ta. \text{conflicting } T = C\text{-Clause } Ta \longrightarrow \text{trail } T \models_{as} C\text{Not } Ta$
unfolding *cdcl_W-all-struct-inv-def cdcl_W-conflicting-def* **by** *blast*
show *?case*

```

apply (rule cdclW-stgy-with-trail-end-has-not-been-learned[OF - - c - DH LH confl' c])
using s lev' IH c unfolding cdclW-all-struct-inv-def by blast+
qed

lemma cdclW-stgy-new-learned-clause:
assumes cdclW-stgy S T and
  lev: cdclW-M-level-inv S and
  E ∉# learned-clss S and
  E ∈# learned-clss T
shows  $\exists S'. \text{backtrack } S S' \wedge \text{conflicting } S = \text{C-Clause } E \wedge \text{full } \text{cdcl}_W\text{-cp } S' T$ 
using assms
proof induction
case conflict'
then show ?case unfolding full1-def by (auto dest: tranclp-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 ∈# learned-clss T
  using learned cp rtranclp-cdclW-cp-learned-clause-inv unfolding full-def by auto
then have backtrack S T and conflicting S = C-Clause E
  using cdclW-o-new-clause-learned-is-backtrack-step[OF - not-yet o] lev by blast+
then show ?case using cp by blast
qed

lemma cdclW-stgy-no-relearned-clause:
assumes
  invR: cdclW-all-struct-inv R and
  st': cdclW-stgy** R S and
  bt: backtrack S T and
  confl: conflicting S = C-Clause E and
  already-learned: E ∈# clauses S and
  R: trail R = []
shows False
proof -
have M-lev: cdclW-M-level-inv R
  using invR unfolding cdclW-all-struct-inv-def by auto
have cdclW-M-level-inv S
  using M-lev assms(2) rtranclp-cdclW-stgy-consistent-inv by blast
with bt obtain D L M1 M2-loc K i where
  T: T ~ cons-trail (Propagated L ((D + {#L#})))
  (reduce-trail-to M1 (add-learned-cls (D + {#L#})))
  (update-backtrack-lvl (get-maximum-level D (trail S)) (update-conflicting C-True S))
  and
  decomp: (Marked K (Suc (get-maximum-level D (trail S))) # M1, M2-loc) ∈
    set (get-all-marked-decomposition (trail S)) and
  k: get-level L (trail S) = backtrack-lvl S and
  level: get-level L (trail S) = get-maximum-level (D + {#L#}) (trail S) and
  confl-S: conflicting S = C-Clause (D + {#L#}) and
  i: i = get-maximum-level D (trail S) and
  undef: undefined-lit M1 L
by (induction rule: backtrack-induction-lev2) metis
obtain M2 where
  M: trail S = M2 @ Marked K (Suc i) # M1
using get-all-marked-decomposition-exists-prepend[OF decomp] unfolding i by (metis append-assoc)

have invS: cdclW-all-struct-inv S

```

using $invR$ $rtrancp-cdcl_W$ -all-struct-inv-inv $rtrancp-cdcl_W$ -stgy-rtrancp-cdcl_W st' **by** $blast$
then have $conf$: $cdcl_W$ -conflicting S **unfolding** $cdcl_W$ -all-struct-inv-def **by** $blast$
then have $trail\ S \models_{as} CNot\ (D + \{\#L\# \})$ **unfolding** $cdcl_W$ -conflicting-def $conf$ - S **by** $auto$
then have MD : $trail\ S \models_{as} CNot\ D$ **by** $auto$

have lev' : $cdcl_W$ -M-level-inv S **using** $invS$ **unfolding** $cdcl_W$ -all-struct-inv-def **by** $blast$

have $get\ lvs\ M$: $get\ all\ levels\ of\ marked\ (trail\ S) = rev\ [1..<Suc\ (backtrack\ lvs\ S)]$
using lev' **unfolding** $cdcl_W$ -M-level-inv-def **by** $auto$

have lev : $cdcl_W$ -M-level-inv R **using** $invR$ **unfolding** $cdcl_W$ -all-struct-inv-def **by** $blast$
then have $vars\ of\ D$: $atms\ of\ D \subseteq atm\ of\ 'lits\ of\ M1$
using $backtrack\ atms\ of\ D\ in\ M1[OF\ lev'\ undef - decomp - - T]$ $conf\ S\ conf\ T\ decomp\ k\ level$
 $lev'\ i\ undef$ **unfolding** $cdcl_W$ -conflicting-def **by** $(auto\ simp: cdcl_W\ M\ level\ inv\ def)$
have $no\ dup$ $(trail\ S)$ **using** lev' **by** $(auto\ simp: cdcl_W\ M\ level\ inv\ decomp)$
have $vars\ in\ M1$:
 $\forall x \in atms\ of\ D. x \notin atm\ of\ 'lits\ of\ (M2\ @\ [Marked\ K\ (get\ maximum\ level\ D\ (trail\ S) + 1)])$
apply $(rule\ vars\ of\ D\ distinct\ atms\ of\ incl\ not\ in\ other[of$
 $M2\ @\ Marked\ K\ (get\ maximum\ level\ D\ (trail\ S) + 1)\ \# \ []\ M1\ D])$
using $\langle no\ dup\ (trail\ S) \rangle M\ vars\ of\ D$ **by** $simp\ all$
have $M1\ D$: $M1 \models_{as} CNot\ D$
using $vars\ in\ M1\ true\ annots\ remove\ if\ not\ in\ vars[of\ M2\ @\ Marked\ K\ (i + 1)\ \# \ []\ M1\ CNot\ D]$
 $\langle trail\ S \models_{as} CNot\ D \rangle M$ **by** $simp$

have $get\ lvs\ M$: $get\ all\ levels\ of\ marked\ (trail\ S) = rev\ [1..<Suc\ (backtrack\ lvs\ S)]$
using lev' **unfolding** $cdcl_W$ -M-level-inv-def **by** $auto$
then have $backtrack\ lvs\ S > 0$ **unfolding** M **by** $(auto\ split: split\ if\ asm\ simp\ add: upt.simps(2))$

obtain $M1'\ K'\ Ls$ **where**
 M' : $trail\ S = Ls\ @\ Marked\ K'\ (backtrack\ lvs\ S)\ \# \ M1'$ **and**
 Ls : $\forall l \in set\ Ls. \neg is\ marked\ l$ **and**
 $set\ M1 \subseteq set\ M1'$
proof –
let $?Ls = takeWhile\ (Not\ o\ is\ marked)\ (trail\ S)$
have MLs : $trail\ S = ?Ls\ @\ dropWhile\ (Not\ o\ is\ marked)\ (trail\ S)$
by $auto$
have $dropWhile\ (Not\ o\ is\ marked)\ (trail\ S) \neq []$ **unfolding** M **by** $auto$
moreover
from $hd\ dropWhile[OF\ this]$ **have** $is\ marked(hd\ (dropWhile\ (Not\ o\ is\ marked)\ (trail\ S)))$
by $simp$
ultimately
obtain $K'\ K'k$ **where**
 $K'k$: $dropWhile\ (Not\ o\ is\ marked)\ (trail\ S)$
 $= Marked\ K'\ K'k\ \# \ tl\ (dropWhile\ (Not\ o\ is\ marked)\ (trail\ S))$
by $(cases\ dropWhile\ (Not\ o\ is\ marked)\ (trail\ S);$
 $cases\ hd\ (dropWhile\ (Not\ o\ is\ marked)\ (trail\ S)))$
 $simp\ all$
moreover have $\forall l \in set\ ?Ls. \neg is\ marked\ l$ **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 $K'k = backtrack\ lvs\ S$
using $calculation(2)$ **by** $(auto\ split: split\ if\ asm\ simp\ add: get\ lvs\ M\ upt.simps(2))$

```

    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-lvs-M: get-all-levels-of-marked (trail S) = rev [1.. $\text{Suc}$  (backtrack-lvl S)]
  using lev' unfolding cdclW-M-level-inv-def by auto
then have backtrack-lvl S > 0 unfolding M by (auto split: split-if-asm simp add: upt.simps(2) i)

have M1'-D: M1' ⊨as CNot D using M1-D ⟨set M1 ⊆ set M1'⟩ by (auto intro: true-annots-mono)
have -L ∈ lits-of (trail S) using conf confl-S unfolding cdclW-conflicting-def by auto
have lvs-M1': get-all-levels-of-marked M1' = rev [1.. $\text{backtrack-lvl}$  S]
  using get-lvs-M Ls by (auto simp add: get-all-levels-of-marked-no-marked M'
    split: split-if-asm simp add: upt.simps(2))
have L-notin: atm-of L ∈ atm-of 'lits-of Ls ∨ atm-of L = atm-of K'
  proof (rule ccontr)
    assume ¬ ?thesis
    then have atm-of L ∉ atm-of 'lits-of (Marked K' (backtrack-lvl S) # rev Ls) by simp
    then have get-level L (trail S) = get-level L M1'
      unfolding M' by auto
    then show False using get-level-in-levels-of-marked[of L M1' ⟨backtrack-lvl S > 0⟩]
      unfolding k lvs-M1' by auto
  qed
obtain Y Z where
  RY: cdclW-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 rtrancpl-cdclW-new-marked-at-beginning-is-decide'[OF st' - lev, of Ls K'
    backtrack-lvl S M1' []]
  unfolding R M' by auto
have [simp]: cdclW-M-level-inv Y
  using RY lev rtrancpl-cdclW-stgy-consistent-inv by blast
obtain M' where trZ: trail Z = M' @ Marked K' (backtrack-lvl S) # M1'
  using rtrancpl-cdclW-stgy-with-trail-end-has-trail-end[OF Z] M' by auto
have no-dup (trail Y)
  using RY lev rtrancpl-cdclW-stgy-consistent-inv unfolding cdclW-M-level-inv-def by blast
then obtain Y' where
  dec: decide Y Y' and
  Y'Z: full cdclW-cp Y' Z and
  no-step cdclW-cp Y
  using cdclW-stgy-trail-has-new-marked-is-decide-step[OF YZ nt Z] M' by auto
have trY: trail Y = M1'
  proof -
    obtain M' where M: trail Z = M' @ Marked K' (backtrack-lvl S) # M1'
      using rtrancpl-cdclW-stgy-with-trail-end-has-trail-end[OF Z] M' by auto
    obtain M'' where M'': trail Z = M'' @ trail Y' and ∀ m ∈ set M''. ¬ is-marked m
      using Y'Z rtrancpl-cdclW-cp-dropWhile-trail' unfolding full-def by blast
    obtain M''' where trail Y' = M''' @ Marked K' (backtrack-lvl S) # M1'
      using M'' unfolding M
      by (metis (no-types, lifting) ⟨∀ m ∈ set M''. ¬ is-marked m⟩ beginning-not-marked-invert)
    then show ?thesis using dec nt by (induction M''') auto
  qed
have Y-CT: conflicting Y = C-True using ⟨decide Y Y'⟩ by auto

```

```

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: D + {#L#} ∈# clauses Y
  have atm-of L ∉ atm-of ' lits-of M1
    apply (rule backtrack-lit-skipped[of - S])
    using decomp i k lev' unfolding cdclW-M-level-inv-def by auto
  then have LM1: undefined-lit M1 L
    by (metis Marked-Propagated-in-iff-in-lits-of atm-of-uminus image-eqI)
  have L-trY: undefined-lit (trail Y) L
    using L-notin ⟨no-dup (trail S)⟩ unfolding defined-lit-map trY M'
    by (auto simp add: image-iff lits-of-def)
  have ∃ Y'. propagate Y Y'
    using propagate-rule[of Y] DL M1'-D L-trY Y-CT trY DL by (metis state-eq-ref)
  then have False using ⟨no-step cdclW-cp Y⟩ propagate' by blast
}
moreover {
  assume DL: D + {#L#} ∉# 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
  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 D + {#L#} ∉# learned-clss S
    apply (rule rtrancp-cdclW-stgy-with-trail-end-has-not-been-learned[OF Z invZ trZ])
    using DL lY-lZ unfolding clauses-def apply simp
    apply (metis (no-types, lifting) ⟨set M1 ⊆ set M1'⟩ image-mono order-trans
        vars-of-D lits-of-def)
    using L-notin ⟨no-dup (trail S)⟩ unfolding M' by (auto simp add: image-iff lits-of-def)
  then have False
    using already-learned DL confl st' M-lev unfolding M'
    by (simp add: ⟨init-clss Y = init-clss R⟩ clauses-def confl-S
        rtrancp-cdclW-stgy-no-more-init-clss)
}
ultimately show False by blast
qed

```

lemma *rtrancp-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: trancp-cdclW-cp-no-more-clauses)
  end

```

```

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: rtrancpl-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 rtrancpl-cdclW-stgy-cdclW-all-struct-inv st by blast
    then have cdclW-M-level-inv S
      unfolding cdclW-all-struct-inv-def by auto
    ultimately obtain E where
      conflicting S = C-Clause E and
      cls-S': clauses S' = {#E#} + clauses S
    using ⟨cdclW-M-level-inv S⟩
    by (induction rule: backtrack-induction-lev2) (auto simp: cdclW-M-level-inv-decomp)
    then have E ∉ # clauses S
      using cdclW-stgy-no-relearned-clause R invR local.backtrack st by blast
    then show ?thesis using IH by (simp add: distinct-mset-add-single cls-S')
  qed auto
qed
qed

```

```

lemma cdclW-stgy-distinct-mset-clauses:
  assumes
    st: cdclW-stgy** (init-state N) S and
    no-duplicate-clause: distinct-mset N and
    no-duplicate-in-clause: distinct-mset-mset N
  shows distinct-mset (clauses S)
  using rtrancpl-cdclW-stgy-distinct-mset-clauses[OF - st] assms
  by (auto simp: cdclW-all-struct-inv-def distinct-cdclW-state-def)

```

17.9 Decrease of a measure

```

fun cdclW-measure where
  cdclW-measure S =
    [(?::nat) ^ (card (atms-of-msu (init-clss S))) - card (set-mset (learned-clss S)),
     if conflicting S = C-True then 1 else 0,
     if conflicting S = C-True then card (atms-of-msu (init-clss S)) - length (trail S)
     else length (trail S)
    ]

```

```

lemma length-model-le-vars-all-inv:
  assumes cdclW-all-struct-inv S
  shows length (trail S) ≤ card (atms-of-msu (init-clss S))
  using assms length-model-le-vars[of S] unfolding cdclW-all-struct-inv-def
  by (auto simp: cdclW-M-level-inv-decomp)
end

```

```

locale cdclW-termination =
  cdclW-ops trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail
  add-init-cl
  add-learned-cls remove-cls update-backtrack-lvl update-conflicting init-state
  restart-state

```

for

trail :: 'st::equal \Rightarrow ('v::linorder, nat, 'v clause) marked-lits **and**
init-clss :: 'st \Rightarrow 'v clauses **and**
learned-clss :: 'st \Rightarrow 'v clauses **and**
backtrack-lvl :: 'st \Rightarrow nat **and**
conflicting :: 'st \Rightarrow 'v clause conflicting-clause **and**

cons-trail :: ('v, nat, 'v clause) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-trail :: 'st \Rightarrow 'st **and**
add-init-cls :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**
add-learned-cls :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**
remove-cls :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**
update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st **and**
update-conflicting :: 'v clause conflicting-clause \Rightarrow 'st \Rightarrow 'st **and**

init-state :: 'v clauses \Rightarrow 'st **and**
restart-state :: 'st \Rightarrow 'st

begin

lemma *learned-clss-less-upper-bound*:

fixes *S* :: 'st

assumes

distinct-cdcl_W-state S **and**

$\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$

shows $\text{card}(\text{set-mset } (\text{learned-clss } S)) \leq 3 \wedge \text{card } (\text{atms-of-msu } (\text{learned-clss } S))$

proof –

have $\text{set-mset } (\text{learned-clss } S) \subseteq \text{build-all-simple-clss } (\text{atms-of-msu } (\text{learned-clss } S))$

apply (*rule simplified-in-build-all*)

using *assms unfolding distinct-cdcl_W-state-def* **by** *auto*

then have $\text{card}(\text{set-mset } (\text{learned-clss } S))$

$\leq \text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{learned-clss } S)))$

by (*simp add: build-all-simple-clss-finite card-mono*)

then show *?thesis*

by (*meson atms-of-ms-finite build-all-simple-clss-card finite-set-mset order-trans*)

qed

lemma *lern3[intro!, simp]*:

$a < a' \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$

$\implies ([a::\text{nat}, b, c], [a', b', c']) \in \text{lern } \{(x, y). x < y\} \text{ } 3$

apply *auto*

unfolding *lern-conv* **apply** *fastforce*

unfolding *lern-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:

$\neg(\text{learned-clss } S \subseteq \# \text{ learned-clss } S' \wedge [] = \text{trail } S' \wedge \text{conflicting } S' = \text{C-True})$

and

learned-clss S $\subseteq \#$ *learned-clss S'* **and**

no-relearn: $\bigwedge S'. \text{backtrack } S S' \implies \forall T. \text{conflicting } S = \text{C-Clause } T \longrightarrow T \notin \# \text{ learned-clss } S$

and

alien: *no-strange-atm S* **and**
M-level: *cdcl_W-M-level-inv S* **and**
no-taut: $\forall s \in \#$ *learned-clss S*. $\neg \text{tautology } s$ **and**
no-dup: *distinct-cdcl_W-state S* **and**
conf: *cdcl_W-conflicting S*
shows (*cdcl_W-measure S'*, *cdcl_W-measure S*) $\in \text{lexn } \{(a, b). a < b\} \ 3$
using *assms(1) M-level assms(2,3)*
proof (*induct rule: cdcl_W-all-induct-lev2*)
case (*propagate C L*) **note** *undef = this(3)* **and** *T = this(4)* **and** *conf = this(5)*
have *propa*: *propagate S (cons-trail (Propagated L (C + {#L#})) S)*
using *propagate-rule[OF - propagate.hyps(1,2)] propagate.hyps* **by** *auto*
then have *no-dup'*: *no-dup (Propagated L (C + {#L#})) # trail S*
by (*metis M-level cdcl_W-M-level-inv-decomp(2) marked-lit.sel(2) propagate'*
r-into-rtrancpl rtrancpl-cdcl_W-cp-consistent-inv trail-cons-trail undef)

let *?N = init-clss S*
have *no-strange-atm (cons-trail (Propagated L (C + {#L#})) S)*
using *alien cdcl_W.propagate cdcl_W-no-strange-atm-inv propa M-level* **by** *blast*
then have *atm-of ' lits-of (Propagated L (C + {#L#})) # trail S*
 \subseteq *atms-of-msu (init-clss S)*
using *undef unfolding no-strange-atm-def* **by** *auto*
then have *card (atm-of ' lits-of (Propagated L (C + {#L#})) # trail S)*
 \leq *card (atms-of-msu (init-clss S))*
by (*meson atms-of-ms-finite card-mono finite-set-mset*)
then have *length (Propagated L (C + {#L#})) # trail S* \leq *card (atms-of-msu ?N)*
using *no-dup-length-eq-card-atm-of-lits-of no-dup'* **by** *fastforce*
then have *H*: *card (atms-of-msu (init-clss S)) - length (trail S)*
 $=$ *Suc (card (atms-of-msu (init-clss S)) - Suc (length (trail S)))*
by *simp*
show *?case using conf T undef* **by** (*auto simp: H*)
next
case (*decide L*) **note** *conf = this(1)* **and** *undef = this(2)* **and** *T = this(4)*
moreover
have *dec*: *decide S (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))*
using *decide.intros decide.hyps* **by** *force*
then have *cdcl_W:cdcl_W S (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))*
using *cdcl_W.simps* **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))*
 \leq *card (atms-of-msu (init-clss S))*
using *no-dup clauses-def undef*
length-model-le-vars[of cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S)]
by *fastforce*
ultimately show *?case using conf* **by** *auto*
next
case (*skip L C' M D*) **note** *tr = this(1)* **and** *conf = this(2)* **and** *T = this(5)*
show *?case using conf T unfolding clauses-def* **by** (*simp add: tr*)
next
case *conflict*


```

then show ?case by simp
next
case resolve
then show ?case using finite unfolding clauses-def by simp
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and conf = this(3) and undef = this(6)
and
  T = this(7) and lev = this(8)
let ?S' = T
have bt: backtrack S ?S'
  using backtrack.hyps backtrack.intros[of S - - - D L K i] by auto
have D + {#L#}  $\notin$  learned-clss S
  using no-relearn conf bt by auto
then have card-T:
  card (set-mset ({#D + {#L#}#} + learned-clss S)) = Suc (card (set-mset (learned-clss S)))
  by (simp add:)
have distinct-cdclW-state ?S'
  using bt M-level distinct-cdclW-state-inv no-dup other by blast
moreover have  $\forall s \in \# \text{learned-clss } ?S'. \neg \text{tautology } s$ 
  using learned-clss-are-not-tautologies[OF cdclW.other[OF cdclW-o.bj[OF
    cdclW-bj.backtrack[OF bt]]]] M-level no-taut confl by auto
ultimately have card (set-mset (learned-clss T))  $\leq 3 \wedge$  card (atms-of-msu (learned-clss T))
  by (auto simp: clauses-def learned-clss-less-upper-bound)
then have H: card (set-mset ({#D + {#L#}#} + learned-clss S))
   $\leq 3 \wedge$  card (atms-of-msu ({#D + {#L#}#} + learned-clss S))
  using T undef decomp lev by (auto simp: cdclW-M-level-inv-decomp)
moreover
  have atms-of-msu ({#D + {#L#}#} + learned-clss S)  $\subseteq$  atms-of-msu (init-clss S)
    using alien conf unfolding no-strange-atm-def by auto
  then have card-f: card (atms-of-msu ({#D + {#L#}#} + learned-clss S))
     $\leq$  card (atms-of-msu (init-clss S))
    by (meson atms-of-ms-finite card-mono finite-set-mset)
  then have (3::nat)  $\wedge$  card (atms-of-msu ({#D + {#L#}#} + learned-clss S))
     $\leq 3 \wedge$  card (atms-of-msu (init-clss S)) by simp
ultimately have (3::nat)  $\wedge$  card (atms-of-msu (init-clss S))
   $\geq$  card (set-mset ({#D + {#L#}#} + learned-clss S))
  using le-trans by blast
then show ?case using decomp undef diff-less-mono2 card-T T lev
  by (auto simp: cdclW-M-level-inv-decomp)
next
case restart
then show ?case using alien by (auto simp: state-eq-def simp del: state-simp)
next
case (forget C T)
then have C  $\in \#$  learned-clss S and C  $\notin \#$  learned-clss T
  by auto
then show ?case using forget(9) by (simp add: mset-leD)
qed

lemma propagate-measure-decreasing:
  fixes S :: 'st
  assumes propagate S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S)  $\in \text{lexn } \{(a, b). a < b\} \ 3$ 
  apply (rule cdclW-measure-decreasing)
  using assms(1) propagate apply blast

```

```

    using assms(1) apply (auto simp add: propagate.simps)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def)
done

lemma conflict-measure-decreasing:
  fixes S :: 'st
  assumes conflict S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S) ∈ lexn {(a, b). a < b} 3
  apply (rule cdclW-measure-decreasing)
  using assms(1) conflict apply blast
    using assms(1) apply (auto simp add: propagate.simps)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def)
done

lemma decide-measure-decreasing:
  fixes S :: 'st
  assumes decide S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S) ∈ lexn {(a, b). a < b} 3
  apply (rule cdclW-measure-decreasing)
  using assms(1) decide other apply blast
    using assms(1) apply (auto simp add: propagate.simps)[3]
    using assms(2) apply (auto simp add: cdclW-all-struct-inv-def)
done

lemma trans-le:
  trans {(a, (b::nat)). a < b}
  unfolding trans-def by auto

lemma cdclW-cp-measure-decreasing:
  fixes S :: 'st
  assumes cdclW-cp S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S) ∈ lexn {(a, b). a < b} 3
  using assms
proof induction
  case conflict'
  then show ?case using conflict-measure-decreasing by blast
next
  case propagate'
  then show ?case using propagate-measure-decreasing by blast
qed

lemma tranclp-cdclW-cp-measure-decreasing:
  fixes S :: 'st
  assumes cdclW-cp++ S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S) ∈ lexn {(a, b). a < b} 3
  using assms
proof induction
  case base
  then show ?case using cdclW-cp-measure-decreasing by blast
next
  case (step T U) note st = this(1) and step = this(2) and IH = this(3) and inv = this(4)
  then have (cdclW-measure T, cdclW-measure S) ∈ lexn {a. case a of (a, b) ⇒ a < b} 3 by blast

  moreover have (cdclW-measure U, cdclW-measure T) ∈ lexn {a. case a of (a, b) ⇒ a < b} 3
    using cdclW-cp-measure-decreasing[OF step] rtranclp-cdclW-all-struct-inv inv

```

```

  tranclp-cdclW-cp-tranclp-cdclW[OF st]
  unfolding trans-def rtranclp-unfold
  by blast
ultimately show ?case using lern-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) ∈ lern {(a, b). a < b} 3
proof -
  have cdclW-all-struct-inv S
  using assms
  by (metis rtranclp-unfold rtranclp-cdclW-all-struct-inv-inv tranclp-cdclW-stgy-tranclp-cdclW)
with assms show ?thesis
proof induction
  case (conflict' V) note cp = this(1) and inv = this(5)
  show ?case
    using tranclp-cdclW-cp-measure-decreasing[OF HOL.conjunct1[OF cp[unfolded full1-def]] inv]
    .
next
  case (other' T U) note st = this(1) and H = this(4,5,6,7) and cp = this(3)
  have cdclW-all-struct-inv T
  using cdclW-all-struct-inv-inv other other'.hyps(1) other'.prems(4) by blast
from tranclp-cdclW-cp-measure-decreasing[OF - this]
  have le-or-eq: (cdclW-measure U, cdclW-measure T) ∈ lern {a. case a of (a, b) ⇒ a < b} 3 ∨
  cdclW-measure U = cdclW-measure T
  using cp unfolding full-def rtranclp-unfold by blast
moreover
  have cdclW-M-level-inv S
  using cdclW-all-struct-inv-def other'.prems(4) by blast
with st have (cdclW-measure T, cdclW-measure S) ∈ lern {a. case a of (a, b) ⇒ a < b} 3
proof (induction rule:cdclW-o-induct-lev2)
  case (decide T)
  then show ?case using decide-measure-decreasing H by blast
next
  case (backtrack K i M1 M2 L D T) note decomp = this(1) and undef = this(6) and T =
  this(7)
  have bt: backtrack S T
  apply (rule backtrack-rule)
  using backtrack.hyps by auto
  then have no-relearn: ∀ T. conflicting S = C-Clause T ⟶ T ∉ # learned-clss S
  using cdclW-stgy-no-relearned-clause[of R S T] H
  unfolding cdclW-all-struct-inv-def clauses-def by auto
  have inv: cdclW-all-struct-inv S
  using ⟨cdclW-all-struct-inv S⟩ by blast
  show ?case
  apply (rule cdclW-measure-decreasing)
  using bt cdclW-bj.backtrack cdclW-o.bj other apply simp
  using bt T undef decomp inv unfolding cdclW-all-struct-inv-def
  cdclW-M-level-inv-def apply auto[]
  using bt T undef decomp inv unfolding cdclW-all-struct-inv-def

```

```

      cdclW-M-level-inv-def apply auto[]
    using bt no-relearn apply auto[]
    using inv unfolding cdclW-all-struct-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def apply simp
    using inv unfolding cdclW-all-struct-inv-def by simp
  next
    case skip
    then show ?case by force
  next
    case resolve
    then show ?case by force
  qed
ultimately show ?case
  by (metis lern-transI transD trans-le)
qed
qed

```

```

lemma tranclp-cdclW-stgy-decreasing:
  fixes R S T :: 'st
  assumes cdclW-stgy++ R S
  trail R = [] and
  cdclW-all-struct-inv R
  shows (cdclW-measure S, cdclW-measure R) ∈ lern {(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]
  lern-transI[OF trans-le, of 3] unfolding trans-def by blast

```

```

lemma tranclp-cdclW-stgy-S0-decreasing:
  fixes R S T :: 'st
  assumes pl: cdclW-stgy++ (init-state N) S and
  no-dup: distinct-mset-mset N
  shows (cdclW-measure S, cdclW-measure (init-state N)) ∈ lern {(a, b). a < b} 3
proof -
  have cdclW-all-struct-inv (init-state N)
    using no-dup unfolding cdclW-all-struct-inv-def by auto
  then show ?thesis using pl tranclp-cdclW-stgy-decreasing init-state-trail by blast
qed

```

```

lemma wf-tranclp-cdclW-stgy:
  wf {(S::'st, init-state N) | S N. distinct-mset-mset N ∧ cdclW-stgy++ (init-state N) S}
  apply (rule wf-wf-if-measure'-notation2[of lern {(a, b). a < b} 3 - cdclW-measure])
  apply (simp add: wf wf-lern)
  using tranclp-cdclW-stgy-S0-decreasing by blast
end

```

```

end
theory DPLL-CDCL-W-Implementation
imports Partial-Annotated-Clausal-Logic
begin

```

18 Simple Implementation of the DPLL and CDCL

18.1 Common Rules

18.1.1 Propagation

The following theorem holds:

lemma *lits-of-unfold*[iff]:

$(\forall c \in \text{set } C. -c \in \text{lits-of } Ms) \longleftrightarrow Ms \models_{as} CNot \ (mset \ C)$

unfolding *true-annots-def Ball-def true-annot-def CNot-def mem-set-multiset-eq* **by** *auto*

The right-hand version is written at a high-level, but only the left-hand side is executable.

definition *is-unit-clause* :: 'a literal list \Rightarrow ('a, 'b, 'c) marked-lit list \Rightarrow 'a literal option

where

is-unit-clause *l* *M* =

(case *List.filter* ($\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of } M$) *l* of
 $a \# [] \Rightarrow \text{if } M \models_{as} CNot \ (mset \ l - \{\#a\# \}) \text{ then } Some \ a \text{ else } None$
 $| - \Rightarrow None$)

definition *is-unit-clause-code* :: 'a literal list \Rightarrow ('a, 'b, 'c) marked-lit list

\Rightarrow 'a literal option **where**

is-unit-clause-code *l* *M* =

(case *List.filter* ($\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of } M$) *l* of
 $a \# [] \Rightarrow \text{if } (\forall c \in \text{set } (remove1 \ a \ l). -c \in \text{lits-of } M) \text{ then } Some \ a \text{ else } None$
 $| - \Rightarrow None$)

lemma *is-unit-clause-is-unit-clause-code*[code]:

is-unit-clause *l* *M* = *is-unit-clause-code* *l* *M*

proof –

have $1: \bigwedge a. (\forall c \in \text{set } (remove1 \ a \ l). -c \in \text{lits-of } M) \longleftrightarrow M \models_{as} CNot \ (mset \ l - \{\#a\# \})$

using *lits-of-unfold*[of *remove1* - *l*, of - *M*] **by** *simp*

thus *?thesis*

unfolding *is-unit-clause-code-def is-unit-clause-def 1* **by** *blast*

qed

lemma *is-unit-clause-some-undef*:

assumes *is-unit-clause* *l* *M* = *Some a*

shows *undefined-lit* *M* *a*

proof –

have (case [*a* ← *l* . *atm-of* *a* \notin *atm-of ' lits-of* *M*] of [] \Rightarrow *None*
 $| [a] \Rightarrow \text{if } M \models_{as} CNot \ (mset \ l - \{\#a\# \}) \text{ then } Some \ a \text{ else } None$
 $| a \# ab \# xa \Rightarrow Map.empty \ xa) = Some \ a$

using *assms* **unfolding** *is-unit-clause-def* .

hence *a* \in *set* [*a* ← *l* . *atm-of* *a* \notin *atm-of ' lits-of* *M*]

apply (case-tac [*a* ← *l* . *atm-of* *a* \notin *atm-of ' lits-of* *M*])

apply *simp*

apply (case-tac *list*) **by** (*auto split: split-if-asm*)

hence *atm-of* *a* \notin *atm-of ' lits-of* *M* **by** *auto*

thus *?thesis*

by (*simp add: Marked-Propagated-in-iff-in-lits-of*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)

qed

lemma *is-unit-clause-some-CNot*: *is-unit-clause* *l* *M* = *Some a* $\implies M \models_{as} CNot \ (mset \ l - \{\#a\# \})$

unfolding *is-unit-clause-def*

proof –

```

assume (case [a ← l . atm-of a ∉ atm-of ' lits-of 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 (case-tac [a ← l . atm-of a ∉ atm-of ' lits-of M], simp)
apply simp
apply (case-tac list) by (auto split: split-if-asm)
qed

```

lemma *is-unit-clause-some-in*: *is-unit-clause l M = Some a ⇒ a ∈ set l*
unfolding *is-unit-clause-def*

proof –

```

assume (case [a ← l . atm-of a ∉ atm-of ' lits-of 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 (case-tac [a ← l . atm-of a ∉ atm-of ' lits-of M])
  (fastforce dest: filter-eq-ConsD split: split-if-asm split: list.splits)+
qed

```

lemma *is-unit-clause-nil*[simp]: *is-unit-clause [] M = None*
unfolding *is-unit-clause-def* **by** auto

18.1.2 Unit propagation for all clauses

Finding the first clause to propagate

```

fun find-first-unit-clause :: 'a literal list list ⇒ ('a, 'b, 'c) marked-lit list
  ⇒ ('a literal × 'a literal list) option where
find-first-unit-clause (a # l) M =
  (case is-unit-clause a M of
    None ⇒ find-first-unit-clause l M
  | Some L ⇒ Some (L, a)) |
find-first-unit-clause [] - = None

```

lemma *find-first-unit-clause-some*:

```

find-first-unit-clause l M = Some (a, c)
⇒ c ∈ set l ∧ M ⊨as CNot (mset c - {#a#}) ∧ undefined-lit M a ∧ a ∈ set c
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 ⊨as CNot (mset c - {#a#}) and
undef: undefined-lit M a and
ac: a ∈ set c
shows is-unit-clause c M ≠ None

```

proof –

```

have [a ← c . atm-of a ∉ atm-of ' lits-of M] = [a]
using assms
proof (induction c)
case Nil thus ?case by simp
next
case (Cons ac c)

```

```

show ?case
  proof (cases a = ac)
    case True
      thus ?thesis using Cons
        by (auto simp del: lits-of-unfold
          simp add: lits-of-unfold[symmetric] Marked-Propagated-in-iff-in-lits-of
            atm-of-eq-atm-of atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)
    next
      case False
        hence T: mset c + {#ac#} - {#a#} = mset c - {#a#} + {#ac#}
          by (auto simp add: multiset-eq-iff)
        show ?thesis using False Cons
          by (auto simp add: T atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)
      qed
    qed
thus ?thesis
using M unfolding is-unit-clause-def by auto
qed

```

```

lemma find-first-unit-clause-none:
  distinct c  $\implies$  c  $\in$  set l  $\implies$  M  $\models$ as CNot (mset c - {#a#})  $\implies$  undefined-lit M a  $\implies$  a  $\in$  set c
 $\implies$  find-first-unit-clause l M  $\neq$  None
by (induction l)
  (auto split: option.split simp add: propagate-is-unit-clause-not-None)

```

18.1.3 Decide

```

fun find-first-unused-var :: 'a literal list list  $\Rightarrow$  'a literal set  $\Rightarrow$  'a literal option where
find-first-unused-var (a # l) M =
  (case List.find ( $\lambda$ lit. lit  $\notin$  M  $\wedge$   $\neg$ lit  $\notin$  M) a of
    None  $\Rightarrow$  find-first-unused-var l M
  | Some a  $\Rightarrow$  Some a) |
find-first-unused-var [] - = None

```

```

lemma find-none[iff]:
  List.find ( $\lambda$ lit. lit  $\notin$  M  $\wedge$   $\neg$ lit  $\notin$  M) a = None  $\longleftrightarrow$  atm-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 ( $\lambda$ lit. lit  $\notin$  M  $\wedge$   $\neg$ lit  $\notin$  M) a = Some b  $\implies$  b  $\in$  set a  $\wedge$  b  $\notin$  M  $\wedge$   $\neg$ b  $\notin$  M
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  $\in$  set l. atm-of ' set a  $\subseteq$  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  $\in$  set l. atm-of ' set a  $\subseteq$  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  $\in$  set l. atm-of ' set a  $\subseteq$  atm-of ' M) by auto

```

qed

lemma *find-first-unused-var-Some*:

find-first-unused-var l $M = \text{Some } a \implies (\exists m \in \text{set } l. a \in \text{set } m \wedge a \notin M \wedge -a \notin M)$
by (induct l) (auto split: option.splits dest: find-some)

lemma *find-first-unused-var-undefined*:

find-first-unused-var l (lits-of Ms) = $\text{Some } a \implies \text{undefined-lit } Ms$ a
using *find-first-unused-var-Some*[of l lits-of Ms a] *Marked-Propagated-in-iff-in-lits-of*
by blast

end

theory *DPLL-W-Implementation*

imports *DPLL-CDCL-W-Implementation* *DPLL-W* $\sim\sim$ /src/HOL/Library/Code-Target-Numeral

begin

18.2 Simple Implementation of DPLL

18.2.1 Combining the propagate and decide: a DPLL step

definition *DPLL-step* :: $\text{int dpll}_W\text{-marked-lits} \times \text{int literal list list}$

$\Rightarrow \text{int dpll}_W\text{-marked-lits} \times \text{int literal list list}$ **where**

DPLL-step = $(\lambda(Ms, N).$

(case *find-first-unit-clause* N Ms of

Some $(L, -) \Rightarrow (\text{Propagated } L \ () \ \# \ Ms, N)$

| - \Rightarrow

if $\exists C \in \text{set } N. (\forall c \in \text{set } C. -c \in \text{lits-of } Ms)$

then

(case *backtrack-split* Ms of

$(-, L \ \# \ M) \Rightarrow (\text{Propagated } (- \ (\text{lit-of } L)) \ () \ \# \ M, N)$

| $(-, -) \Rightarrow (Ms, N)$

)

else

(case *find-first-unused-var* N (lits-of Ms) of

Some $a \Rightarrow (\text{Marked } a \ () \ \# \ Ms, N)$

| None $\Rightarrow (Ms, N))))$

Example of propagation:

value *DPLL-step* ($[\text{Marked } (Neg \ 1) \ ()], [[Pos \ (1::int), Neg \ 2]]$)

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

abbreviation *toS* $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list})$

$(N:: \text{int literal list list}). (Ms, \text{mset } (\text{map mset } N))$

abbreviation *toS'* $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list},$

$N:: \text{int literal list list}). (Ms, \text{mset } (\text{map mset } N))$

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-dpll_W-step*:

assumes *step*: $(Ms', N') = \text{DPLL-step } (Ms, N)$

and *neg*: $(Ms, N) \neq (Ms', N')$

shows dpll_W (*toS* Ms N) (*toS* Ms' N')

proof –

let $?S = (Ms, \text{mset } (\text{map mset } N))$

{ 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 ∈ set N and
  Ms: Ms ⊨as CNot (mset C − {#L#}) and
  undef: undefined-lit Ms L and
  L ∈ 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 ∈ set C) unfolding mem-set-multiset-eq by (auto simp add: C)
hence ?thesis using Ms'N by auto
}
moreover
{ assume unit: find-first-unit-clause N Ms = None
  assume exC: ∃ C ∈ set N. Ms ⊨as CNot (mset C)
  then obtain C where C: C ∈ set N and Ms: Ms ⊨as CNot (mset C) by auto
  then obtain L M M' where bt: backtrack-split Ms = (M', L # M)
    using step exC neq unfolding DPLL-step-def prod.case unit
    by (cases backtrack-split Ms, case-tac b) auto
  hence is-marked L using backtrack-split-snd-hd-marked[of Ms] by auto
  have 1: 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 Ms) = Some L
    using step exC neq unfolding DPLL-step-def prod.case unit
    by (cases find-first-unused-var N (lits-of Ms)) auto
  have dpllW (Ms, mset (map mset N))
    (Marked L () # fst (Ms, mset (map mset N)), snd (Ms, mset (map mset N)))
    apply (rule dpllW.decided[of ?S L])
    using find-first-unused-var-Some[OF unused]
    by (auto simp add: Marked-Propagated-in-iff-in-lits-of atms-of-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-dpll_W-state* (*toS Ms N*)

proof −

have *unit*: *find-first-unit-clause* *N Ms* = *None*
using *step unfolding DPLL-step-def* **by** (*auto split:option.splits*)

```

{ assume n:  $\exists C \in \text{set } N. Ms \models_{as} CNot (mset C)$ 
  hence Ms:  $(Ms, N) = (\text{case backtrack-split } Ms \text{ of } (x, []) \Rightarrow (Ms, N) \mid (x, L \# M) \Rightarrow (Propagated (- \text{lit-of } L) () \# M, N))$ 
    using step unfolding DPLL-step-def by (simp add:unit)

have snd (backtrack-split Ms) = []
proof (cases backtrack-split Ms, cases snd (backtrack-split Ms))
  fix a b
  assume backtrack-split Ms = (a, b) and snd (backtrack-split Ms) = []
  thus snd (backtrack-split Ms) = [] by blast
next
  fix a b aa list
  assume
    bt: backtrack-split Ms = (a, b) and
    bt': snd (backtrack-split Ms) = aa # list
  hence Ms: Ms = Propagated (- lit-of aa) () # list using Ms by auto
  have is-marked aa using backtrack-split-snd-hd-marked[of Ms] bt bt' by auto
  moreover have fst (backtrack-split Ms) @ aa # list = Ms
    using backtrack-split-list-eq[of Ms] bt' by auto
  ultimately have False unfolding Ms by auto
  thus snd (backtrack-split Ms) = [] by blast
qed

hence ?thesis
  using n backtrack-snd-empty-not-marked[of Ms] unfolding conclusive-dpllW-state-def
  by (cases backtrack-split Ms) auto
}
moreover {
  assume n:  $\neg (\exists C \in \text{set } N. Ms \models_{as} CNot (mset C))$ 
  hence find-first-unused-var N (lits-of Ms) = None
    using step unfolding DPLL-step-def by (simp add: unit split: option.splits)
  hence a:  $\forall a \in \text{set } N. \text{atm-of 'set } a \subseteq \text{atm-of ' (lits-of Ms)}$  by auto
  have fst (toS Ms N)  $\models_{asm} \text{snd (toS Ms N)}$  unfolding true-annots-def CNot-def Ball-def
  proof clarify
    fix x
    assume x:  $x \in \text{set-mset (clauses (toS Ms N))}$ 
    hence  $\neg Ms \models_{as} CNot x$  using n unfolding true-annots-def CNot-def Ball-def by auto
    moreover have total-over-m (lits-of Ms) {x}
      using a x image-iff in-mono atms-of-s-def
      unfolding total-over-m-def total-over-set-def lits-of-def by fastforce
    ultimately show fst (toS Ms N)  $\models_a x$ 
      using total-not-CNot[of lits-of Ms x] by (simp add: true-annot-def true-annots-true-cls)
    qed
  hence ?thesis unfolding conclusive-dpllW-state-def by blast
}
ultimately show ?thesis by blast
qed

```

18.2.2 Adding invariants

Invariant tested in the function `function DPLL-ci :: int dpllW-marked-lits \Rightarrow int literal list list`
 `\Rightarrow int dpllW-marked-lits \times int literal list list` **where**
`DPLL-ci Ms N =`
`(if $\neg \text{dpll}_W\text{-all-inv (Ms, mset (map mset N))}$`
`then (Ms, N)`

```

else
  let (Ms', N') = DPLL-step (Ms, N) in
  if (Ms', N') = (Ms, N) then (Ms, N) else DPLL-ci Ms' N)
by fast+
termination
proof (relation {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}})
  show wf {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}}
    using wf-if-measure-f[OF dpllW-wf, of toS'] by auto
next
fix Ms :: int dpllW-marked-lits and N x xa y
assume ¬ ¬ dpllW-all-inv (toS Ms N)
and step: x = DPLL-step (Ms, N)
and x: (xa, y) = x
and (xa, y) ≠ (Ms, N)
thus ((xa, N), Ms, N) ∈ {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}}
  using DPLL-step-is-a-dpllW-step dpllW-same-clauses split-conv by fastforce
qed

No invariant tested  function (domintros) DPLL-part:: int dpllW-marked-lits ⇒ int literal list list
⇒
  int dpllW-marked-lits × int literal list list where
DPLL-part Ms N =
  (let (Ms', N') = DPLL-step (Ms, N) in
  if (Ms', N') = (Ms, N) then (Ms, N) else DPLL-part Ms' N)
by fast+

lemma snd-DPLL-step[simp]:
  snd (DPLL-step (Ms, N)) = N
  unfolding DPLL-step-def by (auto split: split-if option.splits prod.splits list.splits)

lemma 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 (case-tac DPLL-step (Ms, N)) auto
  have inv': dpllW-all-inv (toS Ms' N) by (metis (mono-tags) 1.prem DPLL-step-is-a-dpllW-step Ms'
    dpllW-all-inv old.prod.inject)
  { assume (Ms', N) ≠ (Ms, N)
    hence DPLL-ci Ms' N = DPLL-part Ms' N ∧ DPLL-part-dom (Ms', N) using 1(1)[of - Ms' N]
  }
  Ms'
  1(2) inv' by auto
  hence DPLL-part-dom (Ms, N) using DPLL-part.domintros Ms' by fastforce
  moreover have DPLL-ci Ms N = DPLL-part Ms N using 1.prem DPLL-part.psimps Ms'
    ⟨DPLL-ci Ms' N = DPLL-part Ms' N ∧ DPLL-part-dom (Ms', N)⟩ ⟨DPLL-part-dom (Ms, N)⟩ by
  auto
  ultimately have ?case by blast
}
moreover {
  assume (Ms', N) = (Ms, N)
  hence ?case using DPLL-part.domintros DPLL-part.psimps Ms' by fastforce
}
ultimately show ?case by blast

```

qed

lemma *DPLL-ci-dpll_W-rtrancp*:

assumes *DPLL-ci* *Ms N* = (*Ms'*, *N'*)
shows *dpll_W*^{**} (*toS Ms N*) (*toS Ms' N*)
using *assms*

proof (*induct Ms N arbitrary: Ms' N' rule: DPLL-ci.induct*)

case (*1 Ms N Ms' N'*) **note** *IH* = *this(1)* **and** *step* = *this(2)*

obtain *S₁ S₂* **where** *S*: (*S₁, S₂*) = *DPLL-step* (*Ms, N*) **by** (*case-tac DPLL-step* (*Ms, N*)) *auto*

{ **assume** \neg *dpll_W-all-inv* (*toS Ms N*)
hence (*Ms, N*) = (*Ms', N*) **using** *step* **by** *auto*
hence ?*case* **by** *auto*

}

moreover

{ **assume** *dpll_W-all-inv* (*toS Ms N*)
and (*S₁, S₂*) = (*Ms, N*)
hence ?*case* **using** *S step* **by** *auto*

}

moreover

{ **assume** *dpll_W-all-inv* (*toS Ms N*)
and (*S₁, S₂*) \neq (*Ms, N*)

moreover obtain *S₁' S₂'* **where** *DPLL-ci S₁ N* = (*S₁'*, *S₂'*) **by** (*case-tac DPLL-ci S₁ N*) *auto*

moreover have *DPLL-ci Ms N* = *DPLL-ci S₁ N* **using** *DPLL-ci.simps*[*of Ms N*] *calculation*

proof –

have (*case* (*S₁, S₂*) *of* (*ms, lss*) \Rightarrow

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 *dpll_W*^{**} (*toS S₁' N*) (*toS Ms' N*) **using** *IH*[*of* (*S₁, S₂*) *S₁ S₂*] *S step* **by** *simp*

moreover have *dpll_W* (*toS Ms N*) (*toS S₁ N*)

by (*metis DPLL-step-is-a-dpll_W-step S* \langle (*S₁, S₂*) \neq (*Ms, N*) \rangle *prod.sel(2)* *snd-DPLL-step*)

ultimately have ?*case* **by** (*metis* (*mono-tags, hide-lams*) *IH S* \langle (*S₁, S₂*) \neq (*Ms, N*) \rangle

\langle *DPLL-ci Ms N* = *DPLL-ci S₁ N* \rangle \langle *dpll_W-all-inv* (*toS Ms N*) \rangle *converse-rtrancp-into-rtrancp*
local.step)

}

ultimately show ?*case* **by** *blast*

qed

lemma *dpll_W-all-inv-dpll_W-trancp-irrefl*:

assumes *dpll_W-all-inv* (*Ms, N*)

and *dpll_W*⁺⁺ (*Ms, N*) (*Ms, N*)

shows *False*

proof –

have *1*: *wf* $\{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W^{++} S S'\}$ **using** *dpll_W-wf-trancp* **by** *auto*

have ((*Ms, N*), (*Ms, N*)) $\in \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W^{++} S S'\}$ **using** *assms* **by** *auto*

thus *False* **using** *wf-not-refl*[*OF 1*] **by** *blast*

qed

lemma *DPLL-ci-final-state*:

assumes *step*: $DPLL\text{-}ci\ Ms\ N = (Ms, N)$
and *inv*: $dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
shows *conclusive-dpll_W-state* ($toS\ Ms\ N$)
proof –
have *st*: $dpll_W^{**}\ (toS\ Ms\ N)\ (toS\ Ms\ N)$ **using** $DPLL\text{-}ci\text{-}dpll_W\text{-}rtrancpl[OF\ step]$.
have $DPLL\text{-}step\ (Ms, N) = (Ms, N)$
proof (*rule ccontr*)
obtain $Ms'\ N'$ **where** $Ms'N$: $(Ms', N') = DPLL\text{-}step\ (Ms, N)$
by (*case-tac* $DPLL\text{-}step\ (Ms, N)$) *auto*
assume $\neg\ ?thesis$
hence $DPLL\text{-}ci\ Ms'\ N = (Ms, N)$ **using** *step inv st Ms'N[symmetric]* **by** *fastforce*
hence $dpll_W^{++}\ (toS\ Ms\ N)\ (toS\ Ms\ N)$
by (*metis* $DPLL\text{-}ci\text{-}dpll_W\text{-}rtrancpl\ DPLL\text{-}step\text{-}is\text{-}a\text{-}dpll_W\text{-}step\ Ms'\ N\ \langle DPLL\text{-}step\ (Ms, N) \neq (Ms,$
 $N) \rangle$
prod.sel(2) rtrancpl-into-trancpl2 snd-DPLL-step)
thus *False* **using** $dpll_W\text{-}all\text{-}inv\text{-}dpll_W\text{-}trancpl\text{-}irrefl\ inv$ **by** *auto*
qed
thus *?thesis* **using** $DPLL\text{-}step\text{-}stuck\text{-}final\text{-}state[of\ Ms\ N]$ **by** *simp*
qed

lemma *DPLL-step-obtains*:

obtains Ms' **where** $(Ms', N) = DPLL\text{-}step\ (Ms, N)$
unfolding $DPLL\text{-}step\text{-}def$ **by** (*metis* (*no-types, lifting*) $DPLL\text{-}step\text{-}def\ prod.collapse\ snd\text{-}DPLL\text{-}step$)

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-ci-no-more-step*:

assumes *step*: $DPLL\text{-}ci\ Ms\ N = (Ms', N')$

shows $DPLL\text{-}ci\ Ms'\ N' = (Ms', N')$

using *assms*

proof (*induct arbitrary: Ms' N' rule: DPLL-ci.induct*)

case ($1\ Ms\ N\ Ms'\ N'$) **note** $IH = this(1)$ **and** $step = this(2)$

obtain S_1 **where** S : $(S_1, N) = DPLL\text{-}step\ (Ms, N)$ **using** $DPLL\text{-}step\text{-}obtains$ **by** *auto*

```

{ assume  $\neg dpll_W\text{-all-inv}$  (toS Ms N)
  hence ?case using step by auto
}
moreover {
  assume  $dpll_W\text{-all-inv}$  (toS Ms N)
  and  $(S_1, N) = (Ms, N)$ 
  hence ?case using S step by auto
}
moreover
{ assume inv:  $dpll_W\text{-all-inv}$  (toS Ms N)
  assume n:  $(S_1, N) \neq (Ms, N)$ 
  obtain  $S_1'$  where SS:  $(S_1', N) = DPLL\text{-ci } S_1 N$  using DPLL-ci-obtains by blast
  moreover have  $DPLL\text{-ci } Ms N = DPLL\text{-ci } S_1 N$ 
  proof -
    have (case  $(S_1, N)$  of (ms, lss)  $\Rightarrow$  if (ms, lss) = (Ms, N) then (Ms, N) else  $DPLL\text{-ci } ms N$ )
      =  $DPLL\text{-ci } Ms N$ 
    using S DPLL-ci.simps[of Ms N] calculation inv by presburger
    hence (if  $(S_1, N) = (Ms, N)$  then (Ms, N) else  $DPLL\text{-ci } S_1 N$ ) =  $DPLL\text{-ci } Ms N$ 
    by fastforce
    thus ?thesis
    using calculation n by presburger
  qed
  moreover
    have  $DPLL\text{-ci } S_1' N = (S_1', N)$  using step IH[OF - - S n SS[symmetric]] inv by blast
  ultimately have ?case using step by fastforce
}
ultimately show ?case by blast
qed

```

lemma *DPLL-part-dpll_W-all-inv-final*:

```

fixes M Ms': (int, unit, unit) marked-lit list and
  N :: int literal list list
assumes inv:  $dpll_W\text{-all-inv}$  (Ms, mset (map mset N))
and MsN:  $DPLL\text{-part } Ms N = (Ms', N)$ 
shows conclusive-dpllW-state (toS Ms' N)  $\wedge$   $dpll_W^{**}$  (toS Ms N) (toS Ms' N)
proof -
  have 2:  $DPLL\text{-ci } Ms N = DPLL\text{-part } Ms N$  using inv  $dpll_W\text{-all-inv-implieS-2-eq3-and-dom}$  by blast
  hence star:  $dpll_W^{**}$  (toS Ms N) (toS Ms' N) unfolding MsN using DPLL-ci-dpllW-rtranclp by blast
  hence inv':  $dpll_W\text{-all-inv}$  (toS Ms' N) using inv rtranclp-dpllW-all-inv by blast
  show ?thesis using star DPLL-ci-final-state[OF DPLL-ci-no-more-step inv] 2 unfolding MsN by blast
qed

```

Embedding the invariant into the type

Defining the type `typedef dpllW-state =`

```

{(M::(int, unit, unit) marked-lit list, N::int literal list list).
   $dpll_W\text{-all-inv}$  (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:  $dpll_W\text{-all-inv-def}$ )
```

qed

lemma

DPLL-part-dom (\square , N)

using *assms* *dpll_W-all-inv-implicS-2-eq3-and-dom*[*of* \square 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) $\in \{(M, N). \text{dpll}_W\text{-all-inv } (toS\ M\ N)\}$

by (*smt DPLL-ci.simps DPLL-ci-dpll_W-rtrancpl case-prodE case-prodI2 rough-state-of mem-Collect-eq old.prod.case prod.sel(2) rtrancpl-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*:: *dpll_W-state* \Rightarrow *dpll_W-state* **where**

DPLL-tot S =

(*let* $S' = \text{DPLL-step}'\ S$ *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)

$\in \{(S', S). (toS'\ S', toS'\ S)$

$\in \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W\ S\ S')\}\}$)

show *wf* $\{(b, a).$

(*rough-state-of* b , *rough-state-of* a)

$\in \{(b, a). (toS'\ b, toS'\ a)$

$\in \{(b, a). \text{dpll}_W\text{-all-inv } a \wedge \text{dpll}_W\ a\ b)\}\}$

using *wf-if-measure-f*[*OF wf-if-measure-f*[*OF dpll_W-wf*, *of toS'*], *of rough-state-of*] .

next

fix $S\ x$

assume $x: x = \text{DPLL-step}'\ S$

and $x \neq S$

have *dpll_W-all-inv* (*case rough-state-of* S *of* (Ms, N) \Rightarrow ($Ms, \text{mset } (\text{map } \text{mset } N)$))

by (*metis* (*no-types*, *lifting*) *case-prodE mem-Collect-eq old.prod.case rough-state-of*)

moreover have *dpll_W* (*case rough-state-of* S *of* (Ms, N) \Rightarrow ($Ms, \text{mset } (\text{map } \text{mset } N)$))

(*case rough-state-of* (*DPLL-step'* S) *of* (Ms, N) \Rightarrow ($Ms, \text{mset } (\text{map } \text{mset } N)$))

proof –

obtain $Ms\ N$ **where** $Ms: (Ms, N) = \text{rough-state-of } S$ **by** (*cases rough-state-of* S) *auto*

have *dpll_W-all-inv* (*toS'* (Ms, N)) **using** *calculation unfolding Ms* **by** *blast*

moreover obtain $Ms'\ N'$ **where** $Ms': (Ms', N') = \text{rough-state-of } (\text{DPLL-step}'\ S)$

by (*cases rough-state-of* (*DPLL-step'* S)) *auto*

ultimately have *dpll_W-all-inv* (*toS'* (Ms', N')) **unfolding** Ms'

```

    by (metis (no-types, lifting) case-prod-unfold mem-Collect-eq rough-state-of)

  have dpllW (toS Ms N) (toS Ms' N')
    apply (rule DPLL-step-is-a-dpllW-step[of Ms' N' Ms N])
    unfolding Ms Ms' using ⟨x ≠ S⟩ rough-state-of-inject x by fastforce+
    thus ?thesis unfolding Ms[symmetric] Ms'[symmetric] by auto
  qed
ultimately show (x, S) ∈ {(T', T). (rough-state-of T', rough-state-of T)
  ∈ {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}}}
  by (auto simp add: x)
qed

lemma [code]:
DPLL-tot S =
  (let S' = DPLL-step' S in
   if S' = S then S else DPLL-tot S') by auto

lemma DPLL-tot-DPLL-step-DPLL-tot[simp]: DPLL-tot (DPLL-step' S) = DPLL-tot S
  apply (cases DPLL-step' S = S)
  apply simp
  unfolding DPLL-tot.simps[of S] by (simp del: DPLL-tot.simps)

lemma DOPLL-step'-DPLL-tot[simp]:
  DPLL-step' (DPLL-tot S) = DPLL-tot S
  by (rule DPLL-tot.induct[of λS. DPLL-step' (DPLL-tot S) = DPLL-tot S S])
  (metis (full-types) DPLL-tot.simps)

lemma DPLL-tot-final-state:
  assumes DPLL-tot S = S
  shows conclusive-dpllW-state (toS' (rough-state-of S))
proof -
  have DPLL-step' S = S using assms[symmetric] DOPLL-step'-DPLL-tot by metis
  hence DPLL-step (rough-state-of S) = (rough-state-of S)
    unfolding DPLL-step'-def using DPLL-step-dpllW-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 dpllW** (toS' (rough-state-of S)) (toS' S')
  using assms
proof (induction arbitrary: S' rule: DPLL-tot.induct)
  case (1 S S')
  let ?x = DPLL-step' S
  { assume ?x = S
    then have ?case using 1(2) by simp
  }
  moreover {
    assume S: ?x ≠ S
    have ?case
      apply (cases DPLL-step' S = S)

```



```

    using S apply blast
  by (smt 1.IH 1.prem1 DPLL-step-is-a-dpllW-step DPLL-tot.simps case-prodE2
      rough-state-of-DPLL-step'-DPLL-step rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl
      rtranclp-idemp split-conv)
}
ultimately show ?case by auto
qed

```

```

lemma rough-state-of-rough-state-of-nil[simp]:
  rough-state-of (state-of ([], N)) = ([], N)
  apply (rule DPLL-W-Implementation.dpllW-state.state-of-inverse)
  unfolding dpllW-all-inv-def by auto

```

Theorem of correctness

```

lemma DPLL-tot-correct:
  assumes rough-state-of (DPLL-tot (state-of ([], N))) = (M, N')
  and (M', N'') = toS' (M, N')
  shows M' ⊨asm N'' ⟷ satisfiable (set-mset N'')
proof -
  have dpllW** (toS' ([], N)) (toS' (M, N')) using DPLL-tot-star[OF assms(1)] by auto
  moreover have conclusive-dpllW-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 dpllW-conclusive-state-correct by (smt DPLL-ci.simps
      DPLL-ci-dpllW-rtranclp assms(2) dpllW-all-inv-def prod.case prod.sel(1) prod.sel(2)
      rtranclp-dpllW-inv(3) rtranclp-dpllW-inv-starting-from-0)
qed

```

18.2.3 Code export

A conversion to $DPLL\text{-}W\text{-Implementation.dpll}_W\text{-state}$ **definition** $Con :: (int, unit, unit) \text{ marked-lit list} \times int \text{ literal list list}$

$\Rightarrow dpll_W\text{-state}$ **where**
 $Con\ xs = \text{state-of (if } dpll_W\text{-all-inv (toS (fst xs)) (snd xs) \text{ then } xs \text{ else } ([], []))$

```

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-dpllW-conc-inv mem-Collect-eq
    prod.case-eq-if)

```

A slightly different version of $DPLL\text{-}tot$ where the returned boolean indicates the result.

definition $DPLL\text{-}tot\text{-}rep$ **where**

```

DPLL-tot-rep S =
  (let (M, N) = (rough-state-of (DPLL-tot S)) in (∀ A ∈ set N. (∃ a ∈ set A. a ∈ lits-of (M)), M))

```

One version of the generated SML code is here, but not included in the generated document. The only differences are:

- export $'a$ literal from the SML Module *Clausal-Logic*;
- export the constructor Con from $DPLL\text{-}W\text{-Implementation}$;

- 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 cdclW-mark = 'a clause
type-synonym cdclW-marked-level = nat

type-synonym 'v cdclW-marked-lit = ('v, cdclW-marked-level, 'v cdclW-mark) marked-lit
type-synonym 'v cdclW-marked-lits = ('v, cdclW-marked-level, 'v cdclW-mark) marked-lits
type-synonym 'v cdclW-state =
  'v cdclW-marked-lits × 'v clauses × 'v clauses × nat × 'v clause conflicting-clause

abbreviation trail :: 'a × 'b × 'c × 'd × 'e ⇒ 'a where
trail ≡ (λ(M, -). M)

abbreviation cons-trail :: 'a ⇒ 'a list × 'b × 'c × 'd × 'e ⇒ 'a list × 'b × 'c × 'd × 'e where
cons-trail ≡ (λL (M, S). (L#M, S))

abbreviation tl-trail :: 'a list × 'b × 'c × 'd × 'e ⇒ 'a list × 'b × 'c × 'd × 'e where
tl-trail ≡ (λ(M, S). (tl M, S))

abbreviation clauses :: 'a × 'b × 'c × 'd × 'e ⇒ 'b where
clauses ≡ λ(M, N, -). N

abbreviation learned-clss :: 'a × 'b × 'c × 'd × 'e ⇒ 'c where
learned-clss ≡ λ(M, N, U, -). U

abbreviation backtrack-lvl :: 'a × 'b × 'c × 'd × 'e ⇒ 'd where
backtrack-lvl ≡ λ(M, N, U, k, -). k

abbreviation update-backtrack-lvl :: 'd ⇒ 'a × 'b × 'c × 'd × 'e ⇒ 'a × 'b × 'c × 'd × 'e
where
update-backtrack-lvl ≡ λk (M, N, U, -, S). (M, N, U, k, S)

abbreviation conflicting :: 'a × 'b × 'c × 'd × 'e ⇒ 'e where
conflicting ≡ λ(M, N, U, k, D). D

abbreviation update-conflicting :: 'e ⇒ 'a × 'b × 'c × 'd × 'e ⇒ 'a × 'b × 'c × 'd × 'e
where
update-conflicting ≡ λS (M, N, U, k, -). (M, N, U, k, S)

abbreviation S0-cdclW N ≡ ([], N, {#}, 0, C-True):: 'v cdclW-state

abbreviation add-learned-cls where
add-learned-cls ≡ λC (M, N, U, S). (M, N, {#C#} + U, S)

abbreviation remove-cls where
remove-cls ≡ λC (M, N, U, S). (M, remove-mset C N, remove-mset C U, S)
interpretation cdclW: stateW trail clauses learned-clss backtrack-lvl conflicting

```

```

λL (M, S). (L # M, S)
λ(M, S). (tl M, S)
λC (M, N, S). (M, {#C#} + N, S)
λC (M, N, U, S). (M, N, {#C#} + U, S)
λC (M, N, U, S). (M, remove-mset C N, remove-mset C U, S)
λ(k::nat) (M, N, U, -, D). (M, N, U, k, D)
λD (M, N, U, k, -). (M, N, U, k, D)
λN. ([], N, {#}, 0, C-True)
λ(-, N, U, -). ([], N, U, 0, C-True)
by unfold-locales auto

```

lemma *trail-conv*: trail (M, N, U, k, D) = M **and**
clauses-conv: clauses (M, N, U, k, D) = N **and**
learned-clss-conv: learned-clss (M, N, U, k, D) = U **and**
conflicting-conv: conflicting (M, N, U, k, D) = D **and**
backtrack-lvl-conv: backtrack-lvl (M, N, U, k, D) = k
by auto

lemma *state-conv*:
S = (trail S, clauses S, learned-clss S, backtrack-lvl S, conflicting S)
by (cases S) auto

interpretation *cdcl_W-termination trail clauses learned-clss backtrack-lvl conflicting*

```

λL (M, S). (L # M, S)
λ(M, S). (tl M, S)
λC (M, N, S). (M, {#C#} + N, S)
λC (M, N, U, S). (M, N, {#C#} + U, S)
λC (M, N, U, S). (M, remove-mset C N, remove-mset C U, S)
λ(k::nat) (M, N, U, -, D). (M, N, U, k, D)
λD (M, N, U, k, -). (M, N, U, k, D)
λN. ([], N, {#}, 0, C-True)
λ(-, N, U, -). ([], N, U, 0, C-True)
by intro-locales

```

lemmas *cdcl_W.clauses-def[simp]*

lemma *cdcl_W-state-eq-equality[iff]*: *cdcl_W.state-eq* S T \longleftrightarrow S = T
unfolding *cdcl_W.state-eq-def* **by** (cases S, cases T) auto
declare *cdcl_W.state-simp[simp del]*

18.3 CDCL Implementation

18.3.1 Definition of the rules

Types **lemma** *true-clss-remdups[simp]*:
I \models_s (mset \circ remdups) ‘ N \longleftrightarrow I \models_s mset ‘ N
by (simp add: true-clss-def)

lemma *satisfiable-mset-remdups[simp]*:
satisfiable ((mset \circ remdups) ‘ N) \longleftrightarrow satisfiable (mset ‘ N)
unfolding *satisfiable-carac[symmetric]* **by** simp

declare *mset-map[symmetric, simp]*

value *backtrack-split* [Marked (Pos (Suc 0)) Level]
value $\exists C \in \text{set } [[\text{Pos (Suc 0), Neg (Suc 0)}]]$. ($\forall c \in \text{set } C$. $-c \in \text{lits-of } [\text{Marked (Pos (Suc 0)) Level}]$)

type-synonym $cdcl_W\text{-state-inv-st} = (\text{nat}, \text{nat}, \text{nat literal list}) \text{ marked-lit list} \times \text{nat literal list list} \times \text{nat literal list list} \times \text{nat} \times \text{nat literal list conflicting-clause}$

We need some functions to convert between our abstract state $nat\ cdcl_W\text{-state}$ and the concrete state $cdcl_W\text{-state-inv-st}$.

fun $convert :: ('a, 'b, 'c \text{ list}) \text{ marked-lit} \Rightarrow ('a, 'b, 'c \text{ multiset}) \text{ marked-lit}$ **where**
 $convert\ (Propagated\ L\ C) = Propagated\ L\ (mset\ C) \mid$
 $convert\ (Marked\ K\ i) = Marked\ K\ i$

fun $convertC :: 'a \text{ list conflicting-clause} \Rightarrow 'a \text{ multiset conflicting-clause}$ **where**
 $convertC\ (C\text{-Clause}\ C) = C\text{-Clause}\ (mset\ C) \mid$
 $convertC\ C\text{-True} = C\text{-True}$

lemma $convert\text{-}C\text{True}[iff]:$
 $convertC\ e = C\text{-True} \iff e = C\text{-True}$
by (cases e) auto

lemma $convert\text{-}Propagated[elim!]:$
 $convert\ z = Propagated\ L\ C \implies (\exists C'. z = Propagated\ L\ C' \wedge C = mset\ C')$
by (cases z) auto

lemma $get\text{-}rev\text{-}level\text{-}map\text{-}convert:$
 $get\text{-}rev\text{-}level\ x\ n\ (map\ convert\ M) = get\text{-}rev\text{-}level\ x\ n\ M$
by (induction M arbitrary; n rule: marked-lit-list-induct) auto

lemma $get\text{-}level\text{-}map\text{-}convert[simp]:$
 $get\text{-}level\ x\ (map\ convert\ M) = get\text{-}level\ x\ M$
using $get\text{-}rev\text{-}level\text{-}map\text{-}convert[of\ x\ 0\ rev\ M]$ **by** (simp add: rev-map)

lemma $get\text{-}maximum\text{-}level\text{-}map\text{-}convert[simp]:$
 $get\text{-}maximum\text{-}level\ D\ (map\ convert\ M) = get\text{-}maximum\text{-}level\ D\ M$
by (induction D)
(auto simp add: get-maximum-level-plus)

lemma $get\text{-}all\text{-}levels\text{-}of\text{-}marked\text{-}map\text{-}convert[simp]:$
 $get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (map\ convert\ M) = (get\text{-}all\text{-}levels\text{-}of\text{-}marked\ M)$
by (induction M rule: marked-lit-list-induct) auto

Conversion function

fun $toS :: cdcl_W\text{-state-inv-st} \Rightarrow nat\ cdcl_W\text{-state}$ **where**
 $toS\ (M, N, U, k, C) = (map\ convert\ M, mset\ (map\ mset\ N),\ mset\ (map\ mset\ U), k, convertC\ C)$

Definition an abstract type

typedef $cdcl_W\text{-state-inv} = \{S :: cdcl_W\text{-state-inv-st}. cdcl_W\text{-all-struct-inv}\ (toS\ S)\}$
morphisms $rough\text{-state-of}\ state\text{-of}$
proof
show $([], [], [], 0, C\text{-True}) \in \{S. cdcl_W\text{-all-struct-inv}\ (toS\ S)\}$
by (auto simp add: $cdcl_W\text{-all-struct-inv-def}$)
qed

instantiation $cdcl_W\text{-state-inv} :: equal$
begin

definition *equal-cdcl_W-state-inv* :: *cdcl_W-state-inv* \Rightarrow *cdcl_W-state-inv* \Rightarrow *bool* **where**
equal-cdcl_W-state-inv *S S'* = (*rough-state-of* *S* = *rough-state-of* *S'*)

instance

by *standard* (*simp add: rough-state-of-inject equal-cdcl_W-state-inv-def*)
end

lemma *lits-of-map-convert[simp]*: *lits-of* (*map convert* *M*) = *lits-of* *M*
by (*induction M rule: marked-lit-list-induct*) *simp-all*

lemma *undefined-lit-map-convert[iff]*:
undefined-lit (*map convert* *M*) *L* \longleftrightarrow *undefined-lit* *M* *L*
by (*auto simp add: Marked-Propagated-in-iff-in-lits-of*)

lemma *true-annot-map-convert[simp]*: *map convert* *M* \models_a *N* \longleftrightarrow *M* \models_a *N*
by (*induction M rule: marked-lit-list-induct*) (*simp-all add: true-annot-def*)

lemma *true-annots-map-convert[simp]*: *map convert* *M* \models_{as} *N* \longleftrightarrow *M* \models_{as} *N*
unfolding *true-annots-def* **by** *auto*

lemmas *propagateE*

lemma *find-first-unit-clause-some-is-propagate*:

assumes *H*: *find-first-unit-clause* (*N @ U*) *M* = *Some* (*L*, *C*)
shows *propagate* (*toS* (*M*, *N*, *U*, *k*, *C-True*)) (*toS* (*Propagated* *L* *C* # *M*, *N*, *U*, *k*, *C-True*))
using *assms*
by (*auto dest!: find-first-unit-clause-some simp add: propagate.simps*
intro!: exI[of - mset C - {#L#}])

18.3.2 Propagate

definition *do-propagate-step* **where**

do-propagate-step *S* =
(*case S of*
(*M*, *N*, *U*, *k*, *C-True*) \Rightarrow
(*case find-first-unit-clause* (*N @ U*) *M of*
Some (*L*, *C*) \Rightarrow (*Propagated* *L* *C* # *M*, *N*, *U*, *k*, *C-True*)
| *None* \Rightarrow (*M*, *N*, *U*, *k*, *C-True*))
| *S* \Rightarrow *S*)

lemma *do-propagate-step*:

do-propagate-step *S* \neq *S* \Longrightarrow *propagate* (*toS* *S*) (*toS* (*do-propagate-step* *S*))
apply (*cases S, cases conflicting S*)
using *find-first-unit-clause-some-is-propagate* [*of clauses S learned-clss S trail S - - backtrack-lvl S*]
by (*auto simp add: do-propagate-step-def split: option.splits*)

lemma *do-propagate-step-conflicting-clause[simp]*:

conflicting *S* \neq *C-True* \Longrightarrow *do-propagate-step* *S* = *S*
unfolding *do-propagate-step-def* **by** (*cases S, cases conflicting S*) *auto*

lemma *do-propagate-step-no-step*:

assumes *dist*: $\forall c \in \text{set } (\text{clauses } S @ \text{learned-clss } S).$ *distinct* *c* **and**
prop-step: *do-propagate-step* *S* = *S*
shows *no-step propagate* (*toS* *S*)

proof (*standard, standard*)

fix *T*

assume *propagate* (*toS* *S*) *T*

```

then obtain  $M\ N\ U\ k\ C\ L$  where
   $toSS: toS\ S = (M, N, U, k, C-True)$  and
   $T: T = (Propagated\ L\ (C + \{\#L\#\}) \# M, N, U, k, C-True)$  and
   $MC: M \models_{as} CNot\ C$  and
   $undef: undefined-lit\ M\ L$  and
   $CL: C + \{\#L\#\} \in\# N + U$ 
  apply  $-$  by  $(cases\ toS\ S)\ auto$ 
let  $?M = trail\ S$ 
let  $?N = clauses\ S$ 
let  $?U = learned-clss\ S$ 
let  $?k = backtrack-lvl\ S$ 
let  $?D = C-True$ 
have  $S: S = (?M, ?N, ?U, ?k, ?D)$ 
  using  $toSS$  by  $(cases\ S, cases\ conflicting\ S)\ simp-all$ 
have  $S: toS\ S = toS\ (?M, ?N, ?U, ?k, ?D)$ 
  unfolding  $S[symmetric]$  by  $simp$ 

have
   $M: M = map\ convert\ ?M$  and
   $N: N = mset\ (map\ mset\ ?N)$  and
   $U: U = mset\ (map\ mset\ ?U)$ 
  using  $toSS[unfolded\ S]$  by  $auto$ 

obtain  $D$  where
   $DCL: mset\ D = C + \{\#L\#\}$  and
   $D: D \in set\ (?N\ @\ ?U)$ 
  using  $CL$  unfolding  $N\ U$  by  $auto$ 
obtain  $C'\ L'$  where
   $setD: set\ D = set\ (L' \# C')$  and
   $C': mset\ C' = C$  and
   $L: L = L'$ 
  using  $DCL$  by  $(metis\ ex-mset\ mset.simps(2)\ mset-eq-setD)$ 
have  $find-first-unit-clause\ (?N\ @\ ?U)\ ?M \neq None$ 
  apply  $(rule\ dist\ find-first-unit-clause-none[of\ D\ ?N\ @\ ?U\ ?M\ L, OF - D])$ 
  using  $D\ assms(1)$  apply  $auto[1]$ 
  using  $MC\ setD\ DCL\ M\ MC$  unfolding  $C'[symmetric]$  apply  $auto[1]$ 
  using  $M\ undef$  apply  $auto[1]$ 
  unfolding  $setD\ L$  by  $auto$ 
then show  $False$  using  $prop-step\ S$  unfolding  $do-propagate-step-def$  by  $(cases\ S)\ auto$ 
qed

Conflict fun  $find-conflict$  where
   $find-conflict\ M\ [] = None$  |
   $find-conflict\ M\ (N \# Ns) = (if\ (\forall c \in set\ N. -c \in lits-of\ M)\ then\ Some\ N\ else\ find-conflict\ M\ Ns)$ 

lemma  $find-conflict-Some$ :
   $find-conflict\ M\ Ns = Some\ N \implies N \in set\ Ns \wedge M \models_{as} CNot\ (mset\ N)$ 
  by  $(induction\ Ns\ rule: find-conflict.induct)$ 
   $(auto\ split: split-if-asm)$ 

lemma  $find-conflict-None$ :
   $find-conflict\ M\ Ns = None \longleftrightarrow (\forall N \in set\ Ns. \neg M \models_{as} CNot\ (mset\ N))$ 
  by  $(induction\ Ns)\ auto$ 

lemma  $find-conflict-None-no-confl$ :

```

find-conflict $M (N @ U) = \text{None} \longleftrightarrow \text{no-step conflict } (\text{toS } (M, N, U, k, C\text{-True}))$
by (*auto simp add: find-conflict-None conflict.simps*)

definition *do-conflict-step* **where**

do-conflict-step $S =$
 (*case* S of
 $(M, N, U, k, C\text{-True}) \Rightarrow$
 (*case* *find-conflict* $M (N @ U)$ of
 Some $a \Rightarrow (M, N, U, k, C\text{-Clause } a)$
 None $\Rightarrow (M, N, U, k, C\text{-True})$)
 | $S \Rightarrow S$)

lemma *do-conflict-step*:

do-conflict-step $S \neq S \implies \text{conflict } (\text{toS } S) (\text{toS } (\text{do-conflict-step } S))$
apply (*cases* S , *cases conflicting* S)
unfolding *conflict.simps do-conflict-step-def*
by (*auto dest!: find-conflict-Some split: option.splits*)

lemma *do-conflict-step-no-step*:

do-conflict-step $S = S \implies \text{no-step conflict } (\text{toS } S)$
apply (*cases* S , *cases conflicting* S)
unfolding *do-conflict-step-def*
using *find-conflict-None-no-conf[of trail S clauses S learned-clss S backtrack-lvl S]*
by (*auto split: option.splits*)

lemma *do-conflict-step-conflicting-clause[simp]*:

conflicting $S \neq C\text{-True} \implies \text{do-conflict-step } S = S$
unfolding *do-conflict-step-def* **by** (*cases* S , *cases conflicting* S) *auto*

lemma *do-conflict-step-conflicting[dest]*:

do-conflict-step $S \neq S \implies \text{conflicting } (\text{do-conflict-step } S) \neq C\text{-True}$
unfolding *do-conflict-step-def* **by** (*cases* S , *cases conflicting* S) (*auto split: option.splits*)

definition *do-cp-step* **where**

do-cp-step $S =$
 (*do-propagate-step* o *do-conflict-step*) S

lemma *cp-step-is-cdcl_W-cp*:

assumes H : *do-cp-step* $S \neq S$
shows *cdcl_W-cp* (*toS* S) (*toS* (*do-cp-step* S))

proof —

show *?thesis*

proof (*cases* *do-conflict-step* $S \neq S$)

case *True*

then show *?thesis*

by (*auto simp add: do-conflict-step do-conflict-step-conflicting do-cp-step-def*)

next

case *False*

then have *confl[simp]*: *do-conflict-step* $S = S$ **by** *simp*

show *?thesis*

proof (*cases* *do-propagate-step* $S = S$)

case *True*

then show *?thesis*

using H **by** (*simp add: do-cp-step-def*)

```

next
  case False
  let ?S = toS S
  let ?T = toS (do-propagate-step S)
  let ?U = toS (do-conflict-step (do-propagate-step S))
  have propa: propagate (toS S) ?T using False do-propagate-step by blast
  moreover have ns: no-step conflict (toS S) using confl do-conflict-step-no-step by blast
  ultimately show ?thesis
    using cdclW-cp.intros(2)[of ?S ?T] confl unfolding do-cp-step-def by auto
qed
qed
qed

lemma do-cp-step-eq-no-prop-no-conf:
  do-cp-step S = S  $\implies$  do-conflict-step S = S  $\wedge$  do-propagate-step S = S
  by (cases S, cases conflicting S)
    (auto simp add: do-conflict-step-def do-propagate-step-def do-cp-step-def split: option.splits)

lemma no-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  $\forall c \in \text{set } (\text{clauses } S @ \text{learned-clss } S).$  distinct c
  shows no-step cdclW-cp (toS 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-conf[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-cp-wf-all-inv: wf {(S', S::'v::linorder cdclW-state). cdclW-all-struct-inv S  $\wedge$  cdclW-cp S S'}
  (is wf ?R)
proof (rule wf-bounded-measure[of -  $\lambda S.$  card (atms-of-msu (clauses S))+1
   $\lambda S.$  length (trail S) + (if conflicting S = C-True 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 rtranclp-cdclW-all-struct-inv-inv cdclW-cp-cdclW-st by blast
  ultimately show ?case
    by (auto simp add: cdclW-cp.simps elim!: conflictE propagateE
      dest: length-model-le-vars-all-inv)
qed

lemma cdclW-all-struct-inv-rough-state[simp]: cdclW-all-struct-inv (toS (rough-state-of S))
  using rough-state-of by auto

lemma [simp]: cdclW-all-struct-inv (toS S)  $\implies$  rough-state-of (state-of S) = S
  by (simp add: state-of-inverse)

lemma rough-state-of-state-of-do-cp-step[simp]:

```


$rough_state_of (state_of (do_cp_step (rough_state_of S))) = do_cp_step (rough_state_of S)$
proof –
have $cdcl_W\text{-all-struct-inv} (toS (do_cp_step (rough_state_of S)))$
apply (cases $do_cp_step (rough_state_of S) = (rough_state_of S)$)
apply *simp*
using $cp_step\text{-is-}cdcl_W\text{-cp}[of\ rough_state_of\ S]$
 $cdcl_W\text{-all-struct-inv-rough-state}[of\ S] \ cdcl_W\text{-cp-}cdcl_W\text{-st} \ rtrancp\text{-}cdcl_W\text{-all-struct-inv-inv}$ **by** *blast*
then show *?thesis* **by** *auto*
qed

Skip fun $do_skip_step :: cdcl_W\text{-state-inv-st} \Rightarrow cdcl_W\text{-state-inv-st}$ **where**
 $do_skip_step (Propagated\ L\ C \# Ls, N, U, k, C\text{-Clause}\ D) =$
 $(if\ \neg L \notin set\ D \wedge D \neq []$
 $\text{ then } (Ls, N, U, k, C\text{-Clause}\ D)$
 $\text{ else } (Propagated\ L\ C \# Ls, N, U, k, C\text{-Clause}\ D)) \mid$
 $do_skip_step\ S = S$

lemma do_skip_step :
 $do_skip_step\ S \neq S \implies skip\ (toS\ S)\ (toS\ (do_skip_step\ S))$
apply (*induction* S *rule: do-skip-step.induct*)
by (*auto simp add: skip.simps*)

lemma $do_skip_step\text{-no}$:
 $do_skip_step\ S = S \implies no_step\ skip\ (toS\ S)$
by (*induction* S *rule: do-skip-step.induct*)
 $(auto\ simp\ add: other\ split: split\text{-if-asm})$

lemma $do_skip_step\text{-trail-is-}C\text{-True}[iff]$:
 $do_skip_step\ S = (a, b, c, d, C\text{-True}) \longleftrightarrow S = (a, b, c, d, C\text{-True})$
by (cases S *rule: do-skip-step.cases*) *auto*

Resolve fun $maximum_level_code :: 'a\ literal\ list \Rightarrow ('a, nat, 'a\ literal\ list)\ marked\text{-lit}\ list \Rightarrow nat$ **where**
 $maximum_level_code\ [] = 0 \mid$
 $maximum_level_code\ (L \# Ls)\ M = \max\ (get_level\ L\ M)\ (maximum_level_code\ Ls\ M)$

lemma $maximum_level_code\text{-eq-}get\text{-maximum-level}[code, simp]$:
 $maximum_level_code\ D\ M = get_maximum_level\ (mset\ D)\ M$
by (*induction* D) (*auto simp add: get-maximum-level-plus*)

fun $do_resolve_step :: cdcl_W\text{-state-inv-st} \Rightarrow cdcl_W\text{-state-inv-st}$ **where**
 $do_resolve_step (Propagated\ L\ C \# Ls, N, U, k, C\text{-Clause}\ D) =$
 $(if\ \neg L \in set\ D \wedge (maximum_level_code\ (remove1\ (\neg L)\ D)\ (Propagated\ L\ C \# Ls) = k \vee k = 0)$
 $\text{ then } (Ls, N, U, k, C\text{-Clause}\ (remdups\ (remove1\ L\ C\ @\ remove1\ (\neg L)\ D)))$
 $\text{ else } (Propagated\ L\ C \# Ls, N, U, k, C\text{-Clause}\ D)) \mid$
 $do_resolve_step\ S = S$

lemma $distinct\text{-mset-remdups-union-mset}$:
assumes $distinct\text{-mset}\ A$ **and** $distinct\text{-mset}\ B$
shows $A \# \cup B = remdups\text{-mset}\ (A + B)$
using *assms* **unfolding** $remdups\text{-mset-def}$ **apply** (*auto simp: multiset-eq-iff max-def*)
apply (*metis* $Un\text{-iff}\ count\text{-mset-set}(1)\ count\text{-mset-set}(3)\ distinct\text{-mset-set-mset-ident}$
 $finite\text{-UnI}\ finite\text{-set-mset}\ mem\text{-set-mset-iff}\ not\text{-le}$)
by (*simp add: distinct-mset-def*)

lemma *do-resolve-step*:

$cdcl_W\text{-all-struct-inv } (toS\ S) \implies do\text{-resolve-step } S \neq S$
 $\implies resolve\ (toS\ S)\ (toS\ (do\text{-resolve-step } S))$

proof (*induction S rule: do-resolve-step.induct*)

case $(1\ L\ C\ M\ N\ U\ k\ D)$

moreover

{ **assume** $[simp]: k = 0$

have $get\text{-all-levels-of-marked } (Propagated\ L\ C\ \# M) = []$

using $1(1)\ \text{unfolding } cdcl_W\text{-all-struct-inv-def } cdcl_W\text{-M-level-inv-def}$ **by** *simp*

then have $H: \bigwedge L'. get\text{-level } L' (Propagated\ L\ C\ \# M) = 0$

by (*metis* (*no-types*, *hide-lams*) *Un-insert-left empty-iff get-all-levels-of-marked.simps(3)*
 $get\text{-level-in-levels-of-marked insert-iff list.set}(1)\ sup\text{-bot.left-neutral}$)

} note $H = this$

ultimately have

$- L \in set\ D$ **and**

$M: maximum\text{-level-code } (remove1\ (-L)\ D)\ (Propagated\ L\ C\ \# M) = k$

by (*cases* $mset\ D - \{\#-L\} = \{\#\}$,
 $auto\ dest!: get\text{-maximum-level-exists-lit-of-max-level}[of\ -\ Propagated\ L\ C\ \# M]$
 $split: split\text{-if-asm simp add: } H$) $+$

have *every-mark-is-a-conflict* $(toS\ (Propagated\ L\ C\ \# M, N, U, k, C\text{-Clause } D))$

using $1(1)\ \text{unfolding } cdcl_W\text{-all-struct-inv-def } cdcl_W\text{-conflicting-def}$ **by** *fast*

then have $L \in set\ C$ **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 $\langle -L \in set\ D \rangle$ **by** (*metis* *add commute in-multiset-in-set insert-DiffM*)

have $D'L: D' + \{\#-L\} - \{\#-L\} = D'$ **by** (*auto simp add: multiset-eq-iff*)

have $CL: mset\ C - \{\#L\} + \{\#L\} = mset\ C$ **using** $\langle L \in set\ C \rangle$ **by** (*auto simp add: multiset-eq-iff*)

have

$resolve$

$(map\ convert\ (Propagated\ L\ C\ \# M), mset\ '\# mset\ N, mset\ '\# mset\ U, k, C\text{-Clause } (mset\ D))$
 $(map\ convert\ M, mset\ '\# mset\ N, mset\ '\# mset\ U, k,$
 $C\text{-Clause } (((mset\ D - \{\#-L\}) \# \cup (mset\ C - \{\#L\}))))$

unfolding *resolve.simps*

apply (*simp add: C D*)

using $M[simplified]\ \text{unfolding } maximum\text{-level-code-eq-get-maximum-level } C[symmetric]\ CL$

by (*metis* $D\ D'L\ convert.simps(1)\ get\text{-maximum-level-map-convert list.simps}(9)$)

moreover have

$(map\ convert\ (Propagated\ L\ C\ \# M), mset\ '\# mset\ N, mset\ '\# mset\ U, k, C\text{-Clause } (mset\ D))$
 $= toS\ (Propagated\ L\ C\ \# M, N, U, k, C\text{-Clause } D)$

by *auto*

moreover

have *distinct-mset* $(mset\ C)$ **and** *distinct-mset* $(mset\ D)$

using $\langle cdcl_W\text{-all-struct-inv } (toS\ (Propagated\ L\ C\ \# M, N, U, k, C\text{-Clause } D)) \rangle$

unfolding $cdcl_W\text{-all-struct-inv-def } distinct\text{-cdcl}_W\text{-state-def}$

by *auto*

then have $(mset\ C - \{\#L\}) \# \cup (mset\ D - \{\#-L\}) =$
 $remdups\text{-mset } (mset\ C - \{\#L\} + (mset\ D - \{\#-L\}))$

apply $-$

apply (*rule distinct-mset-remdups-union-mset*)

by *auto*

then have $(map\ convert\ M, mset\ '\# mset\ N, mset\ '\# mset\ U, k,$
 $C\text{-Clause } (((mset\ D - \{\#-L\}) \# \cup (mset\ C - \{\#L\}))))$
 $= toS\ (do\text{-resolve-step } (Propagated\ L\ C\ \# M, N, U, k, C\text{-Clause } D))$

```

    using <- L ∈ set D> M by (auto simp:ac-simps )
ultimately show ?case
  by simp
qed auto

```

```

lemma do-resolve-step-no:
  do-resolve-step S = S ⟹ no-step resolve (toS S)
  apply (cases S; cases hd (trail S); cases conflicting S)
  by (auto
      elim!: resolveE split: split-if-asm
      dest!: union-single-eq-member
      simp del: in-multiset-in-set get-maximum-level-map-convert
      simp add: in-multiset-in-set[symmetric] get-maximum-level-map-convert[symmetric])

```

```

lemma rough-state-of-state-of-resolve[simp]:
  cdclW-all-struct-inv (toS S) ⟹ rough-state-of (state-of (do-resolve-step S)) = do-resolve-step S
  apply (rule state-of-inverse)
  by (smt CollectI bj cdclW-all-struct-inv-inv do-resolve-step other resolve)

```

```

lemma do-resolve-step-trail-is-C-True[iff]:
  do-resolve-step S = (a, b, c, d, C-True) ⟷ S = (a, b, c, d, C-True)
  by (cases S rule: do-resolve-step.cases)
      auto

```

Backjumping fun find-level-decomp where

```

find-level-decomp M [] D k = None |
find-level-decomp M (L # Ls) D k =
  (case (get-level L M, maximum-level-code (D @ Ls) M) of
    (i, j) ⇒ if i = k ∧ j < i then Some (L, j) else find-level-decomp M Ls (L#D) k
  )

```

```

lemma find-level-decomp-some:
  assumes find-level-decomp M Ls D k = Some (L, j)
  shows L ∈ set Ls ∧ get-maximum-level (mset (remove1 L (Ls @ D))) M = j ∧ get-level L M = k
  using assms
  apply (induction Ls arbitrary: D)
  apply simp
  apply (auto split: split-if-asm simp add: ac-simps)
  apply (smt ab-semigroup-add-class.add-ac(1) add.commute diff-union-swap mset.simps(2))
  apply (smt add.commute add.left-commute diff-union-cancelL mset.simps(2))
  apply (smt add.commute add.left-commute diff-union-swap mset.simps(2))
  done

```

```

lemma find-level-decomp-none:
  assumes find-level-decomp M Ls E k = None and mset (L#D) = mset (Ls @ E)
  shows ¬(L ∈ set Ls ∧ get-maximum-level (mset D) M < k ∧ k = get-level L M)
  using assms
  proof (induction Ls arbitrary: E L D)
    case Nil
    then show ?case by simp
  next
    case (Cons L' Ls) note IH = this(1) and find-none = this(2) and LD = this(3)
    have mset D + {#L'#} = mset E + (mset Ls + {#L'#}) ⟹ mset D = mset E + mset Ls
      by (metis add-right-imp-eq union-assoc)

```

then show ?case
using find-none IH[of L' # E L D] LD **by** (auto simp add: ac-simps split: split-if-asm)
qed

fun bt-cut **where**

bt-cut i (Propagated - - # Ls) = bt-cut i Ls |
bt-cut i (Marked K k # Ls) = (if k = Suc i then Some (Marked K k # Ls) else bt-cut i Ls) |
bt-cut i [] = None

lemma bt-cut-some-decomp:

bt-cut i M = Some M' $\implies \exists K M2 M1. M = M2 @ M' \wedge M' = \text{Marked } K (i+1) \# M1$
by (induction i M rule: bt-cut.induct) (auto split: split-if-asm)

lemma bt-cut-not-none: M = M2 @ Marked K (Suc i) # M' \implies bt-cut i M \neq None

by (induction M2 arbitrary: M rule: marked-lit-list-induct) auto

lemma get-all-marked-decomposition-ex:

$\exists N. (\text{Marked } K (\text{Suc } i) \# M', N) \in \text{set } (\text{get-all-marked-decomposition } (M2 @ \text{Marked } K (\text{Suc } i) \# M'))$

apply (induction M2 rule: marked-lit-list-induct)

apply auto[2]

by (case-tac get-all-marked-decomposition (xs @ Marked K (Suc i) # M')) auto

lemma bt-cut-in-get-all-marked-decomposition:

bt-cut i M = Some M' $\implies \exists M2. (M', M2) \in \text{set } (\text{get-all-marked-decomposition } M)$

by (auto dest!: bt-cut-some-decomp simp add: get-all-marked-decomposition-ex)

fun do-backtrack-step **where**

do-backtrack-step (M, N, U, k, C-Clause D) =
(case find-level-decomp M D [] k of
None \Rightarrow (M, N, U, k, C-Clause D)
| Some (L, j) \Rightarrow
(case bt-cut j M of
Some (Marked - - # Ls) \Rightarrow (Propagated L D # Ls, N, D # U, j, C-True)
| - \Rightarrow (M, N, U, k, C-Clause D))
) |
do-backtrack-step S = S

lemma get-all-marked-decomposition-map-convert:

(get-all-marked-decomposition (map convert M)) =
map ($\lambda(a, b). (\text{map convert } a, \text{map convert } b)$) (get-all-marked-decomposition M)

apply (induction M rule: marked-lit-list-induct)

apply simp

by (case-tac get-all-marked-decomposition xs, auto)+

lemma do-backtrack-step:

assumes db: do-backtrack-step S \neq S

and inv: cdcl_W-all-struct-inv (toS S)

shows backtrack (toS S) (toS (do-backtrack-step S))

proof (cases S, cases conflicting S, goal-cases)

case (1 M N U k E)

then show ?case **using** db **by** auto

next

case (2 M N U k E C) **note** S = this(1) **and** confl = this(2)

have E: E = C-Clause C **using** S confl **by** auto

obtain $L\ j$ **where** $fd: find\text{-}level\text{-}decomp\ M\ C\ []\ k = Some\ (L, j)$
using $db\ unfolding\ S\ E$ **by** $(cases\ C)\ (auto\ split: split\text{-}if\text{-}asm\ option.splits)$
have $L \in set\ C$ **and** $get\text{-}maximum\text{-}level\ (mset\ (remove1\ L\ C))\ M = j$ **and**
 $levL: get\text{-}level\ L\ M = k$
using $find\text{-}level\text{-}decomp\ some[OF\ fd]$ **by** $auto$
obtain C' **where** $C: mset\ C = mset\ C' + \{\#L\#\}$
using $\langle L \in set\ C \rangle$ **by** $(metis\ add.commute\ ex\ mset\ in\ multiset\ in\ set\ insert\ DiffM)$
obtain M_2 **where** $M_2: bt\text{-}cut\ j\ M = Some\ M_2$
using $db\ fd\ unfolding\ S\ E$ **by** $(auto\ split: option.splits)$
obtain $M1\ K$ **where** $M1: M_2 = Marked\ K\ (Suc\ j)\ \# \ M1$
using $bt\text{-}cut\ some\ decomp[OF\ M_2]$ **by** $(cases\ M_2)\ auto$
obtain c **where** $c: M = c\ @\ Marked\ K\ (Suc\ j)\ \# \ M1$
using $bt\text{-}cut\ in\ get\ all\ marked\ decomposition[OF\ M_2]$
unfolding $M1$ **by** $fastforce$
have $get\text{-}all\ levels\ of\ marked\ (map\ convert\ M) = rev\ [1..<Suc\ k]$
using $inv\ unfolding\ cdcl_W\ all\ struct\ inv\ def\ cdcl_W\ M\ level\ inv\ def\ S$ **by** $auto$
from $arg\text{-}cong[OF\ this, of\ \lambda a. Suc\ j \in set\ a]$ **have** $j \leq k$ **unfolding** c **by** $auto$
have $max\text{-}l\text{-}j: maximum\text{-}level\text{-}code\ C'\ M = j$
using $db\ fd\ M_2\ C\ unfolding\ S\ E$ **by** $(auto\ split: option.splits\ list.splits\ marked\ lit.splits\ dest!: find\text{-}level\text{-}decomp\ some)[1]$
have $get\text{-}maximum\text{-}level\ (mset\ C)\ M \geq k$
using $\langle L \in set\ C \rangle\ get\text{-}maximum\text{-}level\ ge\ get\text{-}level\ levL$ **by** $blast$
moreover **have** $get\text{-}maximum\text{-}level\ (mset\ C)\ M \leq k$
using $get\text{-}maximum\text{-}level\ exists\ lit\ of\ max\ level[of\ mset\ C\ M]\ inv\ cdcl_W\ M\ level\ inv\ get\ level\ le\ backtrack\ lvl[of\ toS\ S]$
unfolding $C\ cdcl_W\ all\ struct\ inv\ def\ S$
by $auto\ metis+$
ultimately **have** $get\text{-}maximum\text{-}level\ (mset\ C)\ M = k$ **by** $auto$

obtain $M2$ **where** $M2: (M_2, M2) \in set\ (get\text{-}all\ marked\ decomposition\ M)$
using $bt\text{-}cut\ in\ get\ all\ marked\ decomposition[OF\ M_2]$ **by** $metis$
have $H: (cdcl_W.reduce\ trail\ to\ (map\ convert\ M1)\ (add\ learned\ cls\ (mset\ C' + \{\#L\#\})\ (map\ convert\ M, mset\ (map\ mset\ N), mset\ (map\ mset\ U), j, C\ True))) =$
 $(map\ convert\ M1, mset\ (map\ mset\ N), \{\#mset\ C' + \{\#L\#\}\#\} + mset\ (map\ mset\ U), j, C\ True)$
apply $(subst\ state\ conv[of\ cdcl_W.reduce\ trail\ to\ -])$
using $M2\ unfolding\ M1$ **by** $auto$
have
 $backtrack$
 $(map\ convert\ M, mset\ \#\ mset\ N, mset\ \#\ mset\ U, k, C\ Clause\ (mset\ C))$
 $(Propagated\ L\ (mset\ C)\ \# \ map\ convert\ M1, mset\ \#\ mset\ N, mset\ \#\ mset\ U + \{\#mset\ C\#\},$

$j,$
 $C\ True)$
apply $(rule\ backtrack\ rule)$
unfolding C **apply** $simp$
using $Set.imageI[of\ (M_2, M2)\ set\ (get\text{-}all\ marked\ decomposition\ M)\ (\lambda(a, b). (map\ convert\ a, map\ convert\ b))]\ M2$
apply $(auto\ simp: get\text{-}all\ marked\ decomposition\ map\ convert\ M1)[1]$
using $max\text{-}l\text{-}j\ levL\ \langle j \leq k \rangle$ **apply** $(simp\ add: get\text{-}maximum\text{-}level\ plus)$
using $C\ \langle get\text{-}maximum\text{-}level\ (mset\ C)\ M = k \rangle\ levL$ **apply** $auto[1]$
using $max\text{-}l\text{-}j$ **apply** $simp$
apply $(cases\ cdcl_W.reduce\ trail\ to\ (map\ convert\ M1)\ (add\ learned\ cls\ (mset\ C' + \{\#L\#\}))$

```

      (map convert M, mset (map mset N), mset (map mset U), j, C-True)))
    using M2 M1 H by (auto simp: ac-simps)
  then show ?case
    using M2 fd unfolding S E M1 by auto
  obtain M2 where (M2, M2) ∈ set (get-all-marked-decomposition M)
    using bt-cut-in-get-all-marked-decomposition[OF M2] by metis
qed

lemma do-backtrack-step-no:
  assumes db: do-backtrack-step S = S
  and inv: cdclW-all-struct-inv (toS S)
  shows no-step backtrack (toS S)
proof (rule ccontr, cases S, cases conflicting S, goal-cases)
  case 1
  then show ?case using db by (auto split: option.splits)
next
  case (2 M N U k E C) note bt = this(1) and S = this(2) and confl = this(3)
  obtain D L K b z M1 j where
    levL: get-level L M = get-maximum-level (D + {#L#}) M and
    k: k = get-maximum-level (D + {#L#}) M and
    j: j = get-maximum-level D M and
    CE: convertC E = C-Clause (D + {#L#}) and
    decomp: (z # M1, b) ∈ set (get-all-marked-decomposition M) and
    z: Marked K (Suc j) = convert z using bt unfolding S
    by (auto split: option.splits elim!: backtrackE
        simp: get-all-marked-decomposition-map-convert)
  have z: z = Marked K (Suc j) using z by (cases z) auto
  obtain c where c: M = c @ b @ Marked K (Suc j) # M1
    using decomp unfolding z by blast
  have get-all-levels-of-marked (map convert M) = rev [1..W-all-struct-inv-def cdclW-M-level-inv-def S by auto
  from arg-cong[OF this, of λa. Suc j ∈ set a] have k > j unfolding c by auto
  obtain C D' where
    E: E = C-Clause C and
    C: mset C = mset (L # D')
    using CE apply (cases E)
    apply simp
    by (metis conflicting-clause.inject convertC.simps(1) ex-mset mset.simps(2))
  have D'D: mset D' = D
    using C CE E by auto
  have find-level-decomp M C [] k ≠ None
    apply rule
    apply (drule find-level-decomp-none[of - - - L D'])
    using C ⟨k > j⟩ mset-eq-setD unfolding k[symmetric] D'D j[symmetric] levL by fastforce+
  then obtain L' j' where fd-some: find-level-decomp M C [] k = Some (L', j')
    by (cases find-level-decomp M C [] k) auto
  have L': L' = L
  proof (rule ccontr)
    assume ¬ ?thesis
    then have L' ∈# D
      by (metis C D'D fd-some find-level-decomp-some in-multiset-in-set insert-iff list.simps(15))
    then have get-level L' M ≤ get-maximum-level D M
      using get-maximum-level-ge-get-level by blast
    then show False using ⟨k > j⟩ j find-level-decomp-some[OF fd-some] by auto
  qed

```

then have $j': j' = j$ using *find-level-decomp-some*[*OF fd-some*] j C $D'D$ by *auto*

have *bt-cut*: *bt-cut* j $M \neq \text{None}$
 apply (*rule bt-cut-not-none*[*of M - @ -*])
 using *c* by *simp*
 show ?*case* using *db unfolding S E*
 by (*auto split: option.splits list.splits marked-lit.splits*
simp add: fd-some L' j' bt-cut-none
dest: bt-cut-some-decomp)

qed

lemma *rough-state-of-state-of-backtrack*[*simp*]:

assumes *inv*: *cdcl_W-all-struct-inv* (*toS S*)
 shows *rough-state-of* (*state-of* (*do-backtrack-step S*)) = *do-backtrack-step S*

proof (*rule state-of-inverse*)

have *f2*: *backtrack* (*toS S*) (*toS* (*do-backtrack-step S*)) \vee *do-backtrack-step S* = *S*
 using *do-backtrack-step inv* by *blast*
 have $\bigwedge p. \neg \text{cdcl}_W\text{-o}$ (*toS S*) $p \vee \text{cdcl}_W\text{-all-struct-inv}$ p
 using *inv cdcl_W-all-struct-inv-inv other* by *blast*
 then have *do-backtrack-step S* = *S* \vee *cdcl_W-all-struct-inv* (*toS* (*do-backtrack-step S*))
 using *f2* by *blast*
 then show *do-backtrack-step S* \in {*S*. *cdcl_W-all-struct-inv* (*toS S*)}
 using *inv* by *fastforce*

qed

Decide fun *do-decide-step* where

do-decide-step (*M*, *N*, *U*, *k*, *C-True*) =
 (case *find-first-unused-var N* (*lits-of M*) of
None \Rightarrow (*M*, *N*, *U*, *k*, *C-True*)
 | *Some L* \Rightarrow (*Marked L* (*Suc k*) # *M*, *N*, *U*, *k+1*, *C-True*)) |
do-decide-step S = *S*

lemma *do-decide-step*:

do-decide-step S $\neq S \implies \text{decide}$ (*toS S*) (*toS* (*do-decide-step S*))
 apply (*cases S*, *cases conflicting S*)
 defer
 apply (*auto split: option.splits simp add: decide.simps Marked-Propagated-in-iff-in-lits-of*
dest: find-first-unused-var-undefined find-first-unused-var-Some
intro: atms-of-atms-of-ms-mono)[1]

proof –

fix *a b c d e*
 {
 fix *a* :: (*nat*, *nat*, *nat literal list*) *marked-lit list* and
b :: *nat literal list list* and *c* :: *nat literal list list* and
d :: *nat* and *x2* :: *nat literal* and *m* :: *nat literal list*
 assume *a1*: $m \in \text{set } b$
 assume *x2* $\in \text{set } m$
 then have *f2*: *atm-of* *x2* $\in \text{atms-of}$ (*mset m*)
 by *simp*
 have $\bigwedge f. (f m :: \text{nat literal multiset}) \in f \text{ ' set } b$
 using *a1* by *blast*
 then have $\bigwedge f. (\text{atms-of } (f m) :: \text{nat set}) \subseteq \text{atms-of-ms } (f \text{ ' set } b)$
 using *atms-of-atms-of-ms-mono* by *blast*
 then have $\bigwedge n f. (n :: \text{nat}) \in \text{atms-of-ms } (f \text{ ' set } b) \vee n \notin \text{atms-of } (f m)$
 by (*meson contra-subsetD*)

```

    then have atm-of  $x2 \in$  atms-of-ms (mset ‘ set b)
      using f2 by blast
  } note  $H =$  this
  assume do-decide-step  $S \neq S$  and
     $S = (a, b, c, d, e)$  and
    conflicting  $S = C\text{-True}$ 
  then show decide (toS S) (toS (do-decide-step S))

  apply (auto split: option.splits simp add: decide.simps Marked-Propagated-in-iff-in-lits-of
    dest!: find-first-unused-var-Some dest: H)
  by (meson atm-of-in-atm-of-set-in-uminus contra-subsetD rev-image-eqI)+
qed

```

```

lemma do-decide-step-no:
  do-decide-step  $S = S \implies$  no-step decide (toS S)
  apply (cases S, cases conflicting S)
  apply (auto
    simp add: atms-of-ms-mset-unfold atm-of-eq-atm-of Marked-Propagated-in-iff-in-lits-of
    split: option.splits
    elim!: decideE)
  apply (meson atm-of-in-atm-of-set-in-uminus image-subset-iff)
  apply (meson atm-of-in-atm-of-set-in-uminus image-subset-iff)
done

```

```

lemma rough-state-of-state-of-do-decide-step[simp]:
   $cdcl_W\text{-all-struct-inv (toS S)} \implies$  rough-state-of (state-of (do-decide-step S)) = do-decide-step S
  apply (subst state-of-inverse)
  apply (smt  $cdcl_W\text{-all-struct-inv-inv}$  decide do-decide-step mem-Collect-eq other)
  apply simp
done

```

```

lemma rough-state-of-state-of-do-skip-step[simp]:
   $cdcl_W\text{-all-struct-inv (toS S)} \implies$  rough-state-of (state-of (do-skip-step S)) = do-skip-step S
  apply (subst state-of-inverse)
  apply (smt  $cdcl_W\text{-all-struct-inv-inv}$  skip do-skip-step mem-Collect-eq other bj)
  apply simp
done

```

18.3.3 Code generation

Type definition There are two invariants: one while applying conflict and propagate and one for the other rules

```

declare rough-state-of-inverse[simp add]
definition Con where
  Con  $xs =$  state-of (if  $cdcl_W\text{-all-struct-inv (toS (fst xs, snd xs))}$  then  $xs$ 
    else ( $[], [], [], 0, C\text{-True}$ ))

```

```

lemma [code abstype]:
  Con (rough-state-of S) = S
  using rough-state-of[of S] unfolding Con-def by (simp add: rough-state-of-inverse)

```

```

definition do-cp-step' where
  do-cp-step' S = state-of (do-cp-step (rough-state-of S))

```



```

typedef cdclW-state-inv-from-init-state = {S::cdclW-state-inv-st. cdclW-all-struct-inv (toS S)
  ∧ cdclW-stgy** (S0-cdclW (clauses (toS S)) (toS S))}
morphisms rough-state-from-init-state-of state-from-init-state-of
proof
  show ( $\square, \square, \square, \theta, C\text{-True}$ ) ∈ {S. cdclW-all-struct-inv (toS S)
    ∧ cdclW-stgy** (S0-cdclW (clauses (toS S)) (toS S))}
    by (auto simp add: cdclW-all-struct-inv-def)
qed

instantiation cdclW-state-inv-from-init-state :: equal
begin
definition equal-cdclW-state-inv-from-init-state :: cdclW-state-inv-from-init-state ⇒
  cdclW-state-inv-from-init-state ⇒ bool where
equal-cdclW-state-inv-from-init-state S S' ⇔
  (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-cdclW-state-inv-from-init-state-def)
end

definition ConI where
ConI S = state-from-init-state-of (if cdclW-all-struct-inv (toS (fst S, snd S))
  ∧ cdclW-stgy** (S0-cdclW (clauses (toS S)) (toS S))) then S else ( $\square, \square, \square, \theta, C\text{-True}$ ))

lemma [code abstype]:
  ConI (rough-state-from-init-state-of S) = S
  using rough-state-from-init-state-of[of S] unfolding ConI-def by (simp add: rough-state-from-init-state-of-inverse)

definition id-of-I-to:: cdclW-state-inv-from-init-state ⇒ cdclW-state-inv where
id-of-I-to S = state-of (rough-state-from-init-state-of S)

lemma [code abstract]:
  rough-state-of (id-of-I-to S) = rough-state-from-init-state-of S
  unfolding id-of-I-to-def using rough-state-from-init-state-of by auto

Conflict and Propagate function do-full1-cp-step :: cdclW-state-inv ⇒ cdclW-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).
  (toS S', toS 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 toS], of rough-state-of] .
next
  case (2 S' S)
  then show ?case
    unfolding do-cp-step'-def
    apply simp
    by (metis cp-step-is-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 $\lambda S. \text{do-cp-step(rough-state-of (do-full1-cp-step S))}$]
 $= (\text{rough-state-of (do-full1-cp-step S)})$])
(*metis (full-types) do-full1-cp-step.elims rough-state-of-state-of-do-cp-step do-cp-step'-def*)

lemma *in-clauses-rough-state-of-is-distinct*:
 $c \in \text{set (clauses (rough-state-of S) @ learned-clss (rough-state-of S))} \implies \text{distinct } c$
apply (*cases rough-state-of S*)
using *rough-state-of*[of *S*] **by** (*auto simp add: distinct-mset-set-distinct cdcl_W-all-struct-inv-def*
distinct-cdcl_W-state-def)

lemma *do-full1-cp-step-full*:
full cdcl_W-cp (toS (rough-state-of S))
(toS (rough-state-of (do-full1-cp-step S)))
unfolding *full-def* **apply** *standard*
apply (*induction S rule: do-full1-cp-step.induct*)
apply (*smt cp-step-is-cdcl_W-cp do-cp-step'-def do-full1-cp-step.simps*
rough-state-of-state-of-do-cp-step rtranclp.rtrancl-refl rtranclp-into-tranclp2
tranclp-into-rtranclp)

apply (*rule do-cp-step-eq-no-step*[*OF do-full1-cp-step-fix-point-of-do-full1-cp-step*[of *S*]])
using *in-clauses-rough-state-of-is-distinct* **unfolding** *do-cp-step'-def* **by** *blast*

lemma [*code abstract*]:
rough-state-of (do-cp-step' S) = do-cp-step (rough-state-of S)
unfolding *do-cp-step'-def* **by** *auto*

The other rules **fun** *do-other-step* **where**

do-other-step S =
(*let T = do-skip-step S in*
if T \neq S
then T
else
(*let U = do-resolve-step T in*
if U \neq T
then U else
(*let V = do-backtrack-step U in*
if V \neq U then V else do-decide-step V)))

lemma *do-other-step*:
assumes *inv: cdcl_W-all-struct-inv (toS S)* **and**
st: do-other-step S \neq S
shows *cdcl_W-o (toS S) (toS (do-other-step S))*
using *st inv* **by** (*auto split: split-if-asm*
simp add: Let-def
intro: do-skip-step do-resolve-step do-backtrack-step do-decide-step)

lemma *do-other-step-no*:
assumes *inv: cdcl_W-all-struct-inv (toS S)* **and**
st: do-other-step S = S
shows *no-step cdcl_W-o (toS S)*
using *st inv* **by** (*auto split: split-if-asm elim: cdcl_W-bjE*
simp add: Let-def cdcl_W-bj.simps elim!: cdcl_W-o.cases
dest!: do-skip-step-no do-resolve-step-no do-backtrack-step-no do-decide-step-no)

lemma *rough-state-of-state-of-do-other-step*[simp]:
 $\text{rough-state-of } (\text{state-of } (\text{do-other-step } (\text{rough-state-of } S))) = \text{do-other-step } (\text{rough-state-of } S)$
proof (cases $\text{do-other-step } (\text{rough-state-of } S) = \text{rough-state-of } S$)
 case *True*
 then show ?thesis **by** simp
next
 case *False*
 have $\text{cdcl}_W\text{-o } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{do-other-step } (\text{rough-state-of } S)))$
 by (metis *False cdcl_W-all-struct-inv-rough-state do-other-step*[of *rough-state-of S*])
 then have $\text{cdcl}_W\text{-all-struct-inv } (\text{toS } (\text{do-other-step } (\text{rough-state-of } S)))$
 using $\text{cdcl}_W\text{-all-struct-inv-inv cdcl}_W\text{-all-struct-inv-rough-state other}$ **by** blast
 then show ?thesis
 by (simp add: *CollectI state-of-inverse*)
qed

definition *do-other-step'* **where**
 $\text{do-other-step}' S =$
 $\text{state-of } (\text{do-other-step } (\text{rough-state-of } S))$

lemma *rough-state-of-do-other-step'*[code abstract]:
 $\text{rough-state-of } (\text{do-other-step}' S) = \text{do-other-step } (\text{rough-state-of } S)$
apply (cases $\text{do-other-step } (\text{rough-state-of } S) = \text{rough-state-of } S$)
 unfolding *do-other-step'-def* **apply** simp
using *do-other-step*[of *rough-state-of S*] **by** (smt $\text{cdcl}_W\text{-all-struct-inv-inv}$
 $\text{cdcl}_W\text{-all-struct-inv-rough-state mem-Collect-eq other state-of-inverse}$)

definition *do-cdcl_W-stgy-step* **where**
 $\text{do-cdcl}_W\text{-stgy-step } S =$
 $(\text{let } T = \text{do-full1-cp-step } S \text{ in}$
 if $T \neq S$
 then T
 else
 $(\text{let } U = (\text{do-other-step}' T) \text{ in}$
 $(\text{do-full1-cp-step } U)))$

definition *do-cdcl_W-stgy-step'* **where**
 $\text{do-cdcl}_W\text{-stgy-step}' S = \text{state-from-init-state-of } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{id-of-I-to } S)))$

lemma *toS-do-full1-cp-step-not-eq*: $\text{do-full1-cp-step } S \neq S \implies$
 $\text{toS } (\text{rough-state-of } S) \neq \text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } S))$
proof –
 assume $a1: \text{do-full1-cp-step } S \neq S$
 then have $S \neq \text{do-cp-step}' S$
 by fastforce
 then show ?thesis
 by (metis (no-types) *cp-step-is-cdcl_W-cp do-cp-step'-def do-cp-step-eq-no-step*
 do-full1-cp-step-fix-point-of-do-full1-cp-step in-clauses-rough-state-of-is-distinct
 rough-state-of-inverse)
qed

do-full1-cp-step should not be unfolded anymore:

declare *do-full1-cp-step.simps*[simp del]

Correction of the transformation **lemma** *do-cdcl_W-stgy-step*:

```

assumes  $\text{do-cdcl}_W\text{-stgy-step } S \neq S$ 
shows  $\text{cdcl}_W\text{-stgy } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } S)))$ 
proof ( $\text{cases } \text{do-full1-cp-step } S = S$ )
  case False
  then show ?thesis
    using  $\text{assms } \text{do-full1-cp-step-full}[\text{of } S]$  unfolding  $\text{full-unfold } \text{do-cdcl}_W\text{-stgy-step-def}$ 
    by ( $\text{auto intro! : } \text{cdcl}_W\text{-stgy.intros dest : toS-do-full1-cp-step-not-eq}$ )
  next
  case True
  have  $\text{cdcl}_W\text{-o } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-other-step'} S)))$ 
    by ( $\text{smt True assms } \text{cdcl}_W\text{-all-struct-inv-rough-state } \text{do-cdcl}_W\text{-stgy-step-def } \text{do-other-step}$ 
       $\text{rough-state-of-do-other-step'} \text{ rough-state-of-inverse}$ )
  moreover
  have
     $\text{np : no-step propagate } (\text{toS } (\text{rough-state-of } S))$  and
     $\text{nc : no-step conflict } (\text{toS } (\text{rough-state-of } S))$ 
    apply ( $\text{metis True do-cp-step-eq-no-prop-no-conf}$ 
       $\text{do-full1-cp-step-fix-point-of-do-full1-cp-step do-propagate-step-no-step}$ 
       $\text{in-clauses-rough-state-of-is-distinct}$ )
    by ( $\text{metis True do-conflict-step-no-step do-cp-step-eq-no-prop-no-conf}$ 
       $\text{do-full1-cp-step-fix-point-of-do-full1-cp-step}$ )
  then have  $\text{no-step } \text{cdcl}_W\text{-cp } (\text{toS } (\text{rough-state-of } S))$ 
    by ( $\text{simp add : } \text{cdcl}_W\text{-cp.simps}$ )
  moreover have  $\text{full } \text{cdcl}_W\text{-cp } (\text{toS } (\text{rough-state-of } (\text{do-other-step'} S)))$ 
     $(\text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } (\text{do-other-step'} S))))$ 
    using  $\text{do-full1-cp-step-full}$  by auto
  ultimately show ?thesis
    using  $\text{assms True unfolding } \text{do-cdcl}_W\text{-stgy-step-def}$ 
    by ( $\text{auto intro! : } \text{cdcl}_W\text{-stgy.other'} \text{ dest : toS-do-full1-cp-step-not-eq}$ )
qed

```

```

lemma  $\text{length-trail-toS}[\text{simp}]$ :
   $\text{length } (\text{trail } (\text{toS } S)) = \text{length } (\text{trail } S)$ 
  by ( $\text{cases } S$ ) auto

```

```

lemma  $\text{conflicting-noTrue-iff-toS}[\text{simp}]$ :
   $\text{conflicting } (\text{toS } S) \neq C\text{-True} \longleftrightarrow \text{conflicting } S \neq C\text{-True}$ 
  by ( $\text{cases } S$ ) auto

```

```

lemma  $\text{trail-toS-neq-imp-trail-neq}$ :
   $\text{trail } (\text{toS } S) \neq \text{trail } (\text{toS } S') \implies \text{trail } S \neq \text{trail } S'$ 
  by ( $\text{cases } S, \text{cases } S'$ ) auto

```

```

lemma  $\text{do-skip-step-trail-changed-or-conflict}$ :
  assumes  $d : \text{do-other-step } S \neq S$ 
  and  $\text{inv : } \text{cdcl}_W\text{-all-struct-inv } (\text{toS } S)$ 
  shows  $\text{trail } S \neq \text{trail } (\text{do-other-step } S)$ 

```

```

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  $\text{cdcl}_W\text{-M-level-inv } (\text{toS } S)$ 
    using  $\text{inv unfolding } \text{cdcl}_W\text{-all-struct-inv-def}$  by auto
  have  $\text{cdcl}_W\text{-o } (\text{toS } S) (\text{toS } (\text{do-other-step } S))$  using  $\text{do-other-step}[OF \text{ inv } d]$  .
  then show ?thesis
    using  $\langle \text{cdcl}_W\text{-M-level-inv } (\text{toS } S) \rangle$ 

```

```

proof (induction toS (do-other-step S) rule: cdclW-o-induct-lev2)
  case decide
  then show ?thesis
    by (auto simp add: trail-toS-neq-imp-trail-neq)[]
next
case (skip)
then show ?case
  by (cases S; cases do-other-step S) force
next
case (resolve)
then show ?case
  by (cases S, cases do-other-step S) force
next
case (backtrack K i M1 M2 L D) note decomp = this(1) and confl-S = this(3) and undef =
this(6) and
  U = this(7)
have [simp]: cons-trail (Propagated L (D + {#L#}))
  (cdclW.reduce-trail-to M1
  (add-learned-cls (D + {#L#})
  (update-backtrack-lvl (get-maximum-level D (trail (toS S)))
  (update-conflicting C-True (toS S)))))
  =
  (Propagated L (D + {#L#})# M1, mset (map mset (clauses S)),
  {#D + {#L#}#} + mset (map mset (learned-clss S)),
  get-maximum-level D (trail (toS S)), C-True)
apply (subst state-conv[of cons-trail - -])
using decomp undef by (cases S) auto
then show ?case
apply auto
apply (cases do-other-step S; auto split: split-if-asm simp: Let-def)
  apply (cases S rule: do-skip-step.cases; auto split: split-if-asm)
  apply (cases S rule: do-skip-step.cases; auto split: split-if-asm)

  apply (cases S rule: do-backtrack-step.cases;
  auto split: split-if-asm option.splits list.splits marked-lit.splits
  dest!: bt-cut-some-decomp)[]
using d apply (cases S rule: do-decide-step.cases; auto split: option.splits)[]
done
qed
qed

```

lemma do-full1-cp-step-induct:

$(\bigwedge S. (S \neq \text{do-cp-step}' S \implies P (\text{do-cp-step}' S)) \implies P S) \implies P a0$
using do-full1-cp-step.induct **by** metis

lemma do-cp-step-neq-trail-increase:

$\exists c. \text{trail} (\text{do-cp-step } S) = c @ \text{trail } S \wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$
by (cases S, cases conflicting S)
 (auto simp add: do-cp-step-def do-conflict-step-def do-propagate-step-def split: option.splits)

lemma do-full1-cp-step-neq-trail-increase:

$\exists c. \text{trail} (\text{rough-state-of } (\text{do-full1-cp-step } S)) = c @ \text{trail} (\text{rough-state-of } S)$
 $\wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$
apply (induction rule: do-full1-cp-step-induct)
apply (case-tac do-cp-step' S = S)

apply (*simp add: do-full1-cp-step.simps*)
by (*smt Un-iff append-assoc do-cp-step'-def do-cp-step-neq-trail-increase do-full1-cp-step.simps*
rough-state-of-state-of-do-cp-step set-append)

lemma *do-cp-step-conflicting:*

conflicting (rough-state-of S) ≠ C-True ⇒ do-cp-step' S = S
unfolding *do-cp-step'-def do-cp-step-def* **by** *simp*

lemma *do-full1-cp-step-conflicting:*

conflicting (rough-state-of S) ≠ C-True ⇒ do-full1-cp-step S = S
unfolding *do-cp-step'-def do-cp-step-def*
apply (*induction rule: do-full1-cp-step-induct*)
by (*case-tac S ≠ do-cp-step' S*)
(auto simp add: rough-state-of-inverse do-full1-cp-step.simps dest: do-cp-step-conflicting)

lemma *do-decide-step-not-conflicting-one-more-decide:*

assumes
conflicting S = C-True and
do-decide-step S ≠ S
shows *Suc (length (filter is-marked (trail S)))*
= length (filter is-marked (trail (do-decide-step S)))
using *assms unfolding do-other-step'-def*
by (*cases S*) (*auto simp: Let-def split: split-if-asm option.splits*
dest!: find-first-unused-var-Some-not-all-incl)

lemma *do-decide-step-not-conflicting-one-more-decide-bt:*

assumes *conflicting S ≠ C-True and*
do-decide-step S ≠ S
shows *length (filter is-marked (trail S)) < length (filter is-marked (trail (do-decide-step S)))*
using *assms unfolding do-other-step'-def* **by** (*cases S, cases conflicting S*)
(auto simp add: Let-def split: split-if-asm option.splits)

lemma *do-other-step-not-conflicting-one-more-decide-bt:*

assumes *conflicting (rough-state-of S) ≠ C-True and*
conflicting (rough-state-of (do-other-step' S)) = C-True and
do-other-step' S ≠ S
shows *length (filter is-marked (trail (rough-state-of S)))*
> length (filter is-marked (trail (rough-state-of (do-other-step' S))))

proof (*cases S, goal-cases*)

case (*1 y*) **note** *S = this(1) and inv = this(2)*
obtain *M N U k E* **where** *y: y = (M, N, U, k, C-Clause E)*
using *assms(1) S inv* **by** (*cases y, cases conflicting y*) *auto*
have *M: rough-state-of (state-of (M, N, U, k, C-Clause E)) = (M, N, U, k, C-Clause 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))*

using *assms(1,2)* **apply** (*cases rough-state-of (do-other-step' S)*)
apply (*auto simp add: Let-def do-other-step'-def*)
apply (*cases rough-state-of S rule: do-decide-step.cases*)
apply *auto*
done

show *?case*

using *assms(2) S unfolding bt y inv*
apply *simp*
by (*auto simp add: M*)

split: *option.splits*
dest: *bt-cut-some-decomp arg-cong[of - - λu. length (filter is-marked u)]*
qed

lemma *do-other-step-not-conflicting-one-more-decide*:
assumes *conflicting (rough-state-of S) = C-True* **and**
do-other-step' S ≠ S
shows $1 + \text{length (filter is-marked (trail (rough-state-of S)))}$
 $= \text{length (filter is-marked (trail (rough-state-of (do-other-step' S))))}$
proof (*cases S, goal-cases*)
case (*1 y*) **note** *S = this(1)* **and** *inv = this(2)*
obtain *M N U k* **where** *y: y = (M, N, U, k, C-True)* **using** *assms(1) S inv* **by** (*cases y*) *auto*
have *M: rough-state-of (state-of (M, N, U, k, C-True)) = (M, N, U, k, C-True)*
using *inv y* **by** (*auto simp add: state-of-inverse*)
have *state-of (do-decide-step (M, N, U, k, C-True)) ≠ state-of (M, N, U, k, C-True)*
using *assms(2) unfolding do-other-step'-def y inv S* **by** (*auto simp add: M*)
then have *f4: do-skip-step (rough-state-of S) = rough-state-of S*
unfolding *S M y* **by** (*metis (full-types) do-skip-step.simps(4)*)
have *f5: do-resolve-step (rough-state-of S) = rough-state-of S*
unfolding *S M y* **by** (*metis (no-types) do-resolve-step.simps(4)*)
have *f6: do-backtrack-step (rough-state-of S) = rough-state-of S*
unfolding *S M y* **by** (*metis (no-types) do-backtrack-step.simps(2)*)
have *do-other-step (rough-state-of S) ≠ rough-state-of S*
using *assms(2) unfolding S M y do-other-step'-def* **by** (*metis (no-types)*)
then show *?case*
using *f6 f5 f4* **by** (*simp add: assms(1) do-decide-step-not-conflicting-one-more-decide*
do-other-step'-def)
qed

lemma *rough-state-of-state-of-do-skip-step-rough-state-of[simp]*:
rough-state-of (state-of (do-skip-step (rough-state-of S))) = do-skip-step (rough-state-of S)
by (*smt do-other-step.simps rough-state-of-inverse rough-state-of-state-of-do-other-step*)

lemma *conflicting-do-resolve-step-iff[iff]*:
conflicting (do-resolve-step S) = C-True \longleftrightarrow conflicting S = C-True
by (*cases S rule: do-resolve-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *conflicting-do-skip-step-iff[iff]*:
conflicting (do-skip-step S) = C-True \longleftrightarrow conflicting S = C-True
by (*cases S rule: do-skip-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *conflicting-do-decide-step-iff[iff]*:
conflicting (do-decide-step S) = C-True \longleftrightarrow conflicting S = C-True
by (*cases S rule: do-decide-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *conflicting-do-backtrack-step-imp[simp]*:
do-backtrack-step S ≠ S \implies conflicting (do-backtrack-step S) = C-True
by (*cases S rule: do-backtrack-step.cases*)
(auto simp add: Let-def split: list.splits option.splits marked-lit.splits)

lemma *do-skip-step-eq-iff-trail-eq*:

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*:
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 $S = S \longleftrightarrow \text{trail } (\text{do-backtrack-step } S) = \text{trail } S$
by (cases S rule: *do-backtrack-step.cases*)
(*auto split: option.split list.splits marked-lit.splits*
dest!: bt-cut-in-get-all-marked-decomposition)

lemma *do-resolve-step-eq-iff-trail-eq*:
do-resolve-step $S = S \longleftrightarrow \text{trail } (\text{do-resolve-step } S) = \text{trail } S$
by (cases S rule: *do-resolve-step.cases*) *auto*

lemma *do-other-step-eq-iff-trail-eq*:
 $\text{trail } (\text{do-other-step } S) = \text{trail } S \longleftrightarrow \text{do-other-step } S = S$
by (*auto simp add: Let-def do-skip-step-eq-iff-trail-eq[symmetric]*
do-decide-step-eq-iff-trail-eq[symmetric] do-backtrack-step-eq-iff-trail-eq[symmetric]
do-resolve-step-eq-iff-trail-eq[symmetric])

lemma *do-full1-cp-step-do-other-step'-normal-form[dest!]*:

assumes H : *do-full1-cp-step* (*do-other-step'* S) = S
shows *do-other-step'* $S = S \wedge \text{do-full1-cp-step } S = S$

proof –

let $?T = \text{do-other-step}' S$
{ **assume** *confl*: *conflicting* (*rough-state-of* $?T$) $\neq C\text{-True}$
then have *tr*: $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } ?T)) = \text{trail } (\text{rough-state-of } ?T)$
using *do-full1-cp-step-conflicting* **by** *auto*
have $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } (\text{do-other-step}' S))) = \text{trail } (\text{rough-state-of } S)$
using *arg-cong[OF H, of $\lambda S. \text{trail } (\text{rough-state-of } S)$]* .
then have $\text{trail } (\text{rough-state-of } (\text{do-other-step}' S)) = \text{trail } (\text{rough-state-of } S)$
by (*auto simp add: do-full1-cp-step-conflicting confl*)
then have *do-other-step'* $S = S$
by (*simp add: do-other-step-eq-iff-trail-eq do-other-step'-def rough-state-of-inverse*
del: do-other-step.simps)

}

moreover {
assume *eq[simp]*: *do-other-step'* $S = S$
obtain c **where** c : $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } S)) = c @ \text{trail } (\text{rough-state-of } S)$
using *do-full1-cp-step-neq-trail-increase* **by** *auto*

moreover have $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } S)) = \text{trail } (\text{rough-state-of } S)$
using *arg-cong[OF H, of $\lambda S. \text{trail } (\text{rough-state-of } S)$]* **by** *simp*

finally have $c = []$ **by** *blast*

then have *do-full1-cp-step* $S = S$ **using** *assms* **by** *auto*

}

moreover {
assume *confl*: *conflicting* (*rough-state-of* $?T$) = $C\text{-True}$ **and** *neg*: *do-other-step'* $S \neq S$
obtain c **where**
 c : $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } ?T)) = c @ \text{trail } (\text{rough-state-of } ?T)$ **and**
 nm : $\forall m \in \text{set } c. \neg \text{is-marked } m$


```

    using do-full1-cp-step-neq-trail-increase by auto
  have length (filter is-marked (trail (rough-state-of (do-full1-cp-step ?T))))
    = length (filter is-marked (trail (rough-state-of ?T))) using nm unfolding c by force
  moreover have length (filter is-marked (trail (rough-state-of S)))
    ≠ length (filter is-marked (trail (rough-state-of ?T)))
  using do-other-step-not-conflicting-one-more-decide[OF - neq]
  do-other-step-not-conflicting-one-more-decide-bt[of S, OF - confl neq]
  by linarith
  finally have False unfolding H by blast
}
ultimately show ?thesis by blast
qed

```

```

lemma do-cdclW-stgy-step-no:
  assumes S: do-cdclW-stgy-step S = S
  shows no-step cdclW-stgy (toS (rough-state-of S))
proof -
  {
    fix S'
    assume full1 cdclW-cp (toS (rough-state-of S)) S'
    then have False
      using do-full1-cp-step-full[of S] unfolding full-def S rtranclp-unfold full1-def
      by (smt assms do-cdclW-stgy-step-def tranclpD)
  }
  moreover {
    fix S' S''
    assume cdclW-o (toS (rough-state-of S)) S' and
      no-step propagate (toS (rough-state-of S)) and
      no-step conflict (toS (rough-state-of S)) and
      full cdclW-cp S' S''
    then have False
      using assms unfolding do-cdclW-stgy-step-def
      by (smt cdclW-all-struct-inv-rough-state do-full1-cp-step-do-other-step'-normal-form
        do-other-step-no rough-state-of-do-other-step')
  }
  ultimately show ?thesis using assms by (force simp: cdclW-cp.simps cdclW-stgy.simps)
qed

```

```

lemma toS-rough-state-of-state-of-rough-state-from-init-state-of[simp]:
  toS (rough-state-of (state-of (rough-state-from-init-state-of S)))
    = toS (rough-state-from-init-state-of S)
  using rough-state-from-init-state-of[of S] by (auto simp add: state-of-inverse)

```

```

lemma cdclW-cp-is-rtranclp-cdclW: cdclW-cp S T ⇒ 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 ⇒ cdclW** S T
  apply (induction rule: rtranclp-induct)
  apply simp
  by (fastforce dest!: cdclW-cp-is-rtranclp-cdclW)

```

```

lemma cdclW-stgy-is-rtranclp-cdclW:
  cdclW-stgy S T ⇒ cdclW** 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!*:*cdcl_W.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 *clauses S = clauses 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]*:
clauses (toS (rough-state-of (do-cdcl_W-stgy-step (state-of (rough-state-from-init-state-of S))))
 $=$ *clauses (toS (rough-state-from-init-state-of S)) (is - = clauses (toS ?S))*
apply (*cases do-cdcl_W-stgy-step (state-of ?S) = state-of ?S*)
apply *simp*
by (*smt cdcl_W-all-struct-inv-def cdcl_W-all-struct-inv-rough-state cdcl_W-stgy-no-more-init-clss*
do-cdcl_W-stgy-step toS-rough-state-of-state-of-rough-state-from-init-state-of)

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** (S0-cdcl_W (clauses (toS (rough-state-from-init-state-of S))))*
(toS (rough-state-from-init-state-of S))
using *rough-state-from-init-state-of[of S]* **by** *auto*
moreover have *cdcl_W-stgy***
(toS (rough-state-from-init-state-of S))
(toS (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 (toS (rough-state-from-init-state-of T)),
cdcl_W-measure (toS (rough-state-from-init-state-of S)))
 \in *learn {(a, b). a < b} 3*}, *goal-cases*)

case *1*

show *?case* **by** (*rule wf-if-measure-f*) (*auto intro!*: *wf-learn wf-less*)

next

case (*2 S T*) **note** *T = this(1)* **and** *ST = this(2)*

let *?S = rough-state-from-init-state-of S*

have *S: cdcl_W-stgy** (S0-cdcl_W (clauses (toS ?S))) (toS ?S)*

using *rough-state-from-init-state-of[of S]* **by** *auto*

moreover have *cdcl_W-stgy (toS (rough-state-from-init-state-of S))*

(toS (rough-state-from-init-state-of T))

using *ST do-cdcl_W-stgy-step* **unfolding** *T*

by (*smt id-of-I-to-def mem-Collect-eq rough-state-from-init-state-of*
rough-state-from-init-state-of-do-cdcl_W-stgy-step' rough-state-from-init-state-of-inject)

```

    state-of-inverse)
  moreover
    have cdclW-all-struct-inv (toS (rough-state-from-init-state-of S))
      using rough-state-from-init-state-of[of S] by auto
    then have cdclW-all-struct-inv (S0-cdclW (clauses (toS (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 - - S0-cdclW (clauses (toS ?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 no-step-cdclW-stgy-cdclW-all:
  no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy S)))
  apply (induction S rule:do-all-cdclW-stgy.induct)
  apply (case-tac do-cdclW-stgy-step' S  $\neq$  S)
proof -
  fix Sa :: 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  $\neg cdcl_W-stgy (toS (rough-state-from-init-state-of (do-all-cdcl_W-stgy Sa)))$  pp
      using a1 by (metis (no-types) do-cdclW-stgy-step-no id-of-I-to-def
          rough-state-from-init-state-of-do-cdclW-stgy-step' rough-state-of-inverse) }
    then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa)))
      by fastforce
  }
next
  fix Sa :: cdclW-state-inv-from-init-state
  assume a1: do-cdclW-stgy-step' Sa  $\neq$  Sa
     $\implies no-step cdcl_W-stgy (toS (rough-state-from-init-state-of (do-all-cdcl_W-stgy (do-cdcl_W-stgy-step'$ 
    Sa))))
  assume a2: do-cdclW-stgy-step' Sa  $\neq$  Sa
  have do-all-cdclW-stgy Sa = do-all-cdclW-stgy (do-cdclW-stgy-step' Sa)
    by (metis (full-types) do-all-cdclW-stgy.simps)
  then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa)))
    using a2 a1 by presburger
qed

lemma do-all-cdclW-stgy-is-rtranclp-cdclW-stgy:
  cdclW-stgy** (toS (rough-state-from-init-state-of S))
    (toS (rough-state-from-init-state-of (do-all-cdclW-stgy S)))
  apply (induction S rule: do-all-cdclW-stgy.induct)
  apply (case-tac do-cdclW-stgy-step' S = S)
  apply simp
  by (smt converse-rtranclp-into-rtranclp do-all-cdclW-stgy.simps do-cdclW-stgy-step id-of-I-to-def
      rough-state-from-init-state-of-do-cdclW-stgy-step'
      toS-rough-state-of-state-of-rough-state-from-init-state-of)

```

Final theorem:

lemma *DPLL-tot-correct*:

assumes

r: *rough-state-from-init-state-of* (*do-all-cdcl_W-stgy* (*state-from-init-state-of* ($\llbracket \cdot \rrbracket$, *map remdups* *N*, $\llbracket \cdot \rrbracket$, 0, *C-True*)))) = *S* **and**

S: (*M'*, *N'*, *U'*, *k*, *E*) = *toS S*

shows (*E* \neq *C-Clause* {#} \wedge *satisfiable* (*set* (*map mset* *N*)))
 \vee (*E* = *C-Clause* {#} \wedge *unsatisfiable* (*set* (*map mset* *N*)))

proof –

let ?*N* = *map remdups N*

have *inv*: *cdcl_W-all-struct-inv* (*toS* ($\llbracket \cdot \rrbracket$, *map remdups* *N*, $\llbracket \cdot \rrbracket$, 0, *C-True*))

unfolding *cdcl_W-all-struct-inv-def* *distinct-cdcl_W-state-def* *distinct-mset-set-def* **by** *auto*

then have *S0*: *rough-state-of* (*state-of* ($\llbracket \cdot \rrbracket$, *map remdups* *N*, $\llbracket \cdot \rrbracket$, 0, *C-True*))

= ($\llbracket \cdot \rrbracket$, *map remdups* *N*, $\llbracket \cdot \rrbracket$, 0, *C-True*) **by** *simp*

have 1: *full cdcl_W-stgy* (*toS* ($\llbracket \cdot \rrbracket$, ?*N*, $\llbracket \cdot \rrbracket$, 0, *C-True*)) (*toS S*)

unfolding *full-def* **apply** *rule*

using *do-all-cdcl_W-stgy-is-rtranclp-cdcl_W-stgy*[*of*

state-from-init-state-of ($\llbracket \cdot \rrbracket$, *map remdups* *N*, $\llbracket \cdot \rrbracket$, 0, *C-True*)] *inv*

no-step-cdcl_W-stgy-cdcl_W-all

by (*auto simp del: do-all-cdcl_W-stgy.simps simp: state-from-init-state-of-inverse*
r[symmetric]) +

moreover have 2: *finite* (*set* (*map mset* ?*N*)) **by** *auto*

moreover have 3: *distinct-mset-set* (*set* (*map mset* ?*N*))

unfolding *distinct-mset-set-def* **by** *auto*

moreover

have *cdcl_W-all-struct-inv* (*toS S*)

by (*metis* (*no-types*) *cdcl_W-all-struct-inv-rough-state* *r*
toS-rough-state-of-state-of-rough-state-from-init-state-of)

then have *cons*: *consistent-interp* (*lits-of* *M'*)

unfolding *cdcl_W-all-struct-inv-def* *cdcl_W-M-level-inv-def* *S[symmetric]* **by** *auto*

moreover

have *clauses* (*toS* ($\llbracket \cdot \rrbracket$, ?*N*, $\llbracket \cdot \rrbracket$, 0, *C-True*)) = *clauses* (*toS S*)

apply (*rule rtranclp-cdcl_W-init-clss*)

using 1 **unfolding** *full-def* **by** (*auto simp add: rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*)

then have *N'*: *mset* (*map mset* ?*N*) = *N'*

using *S[symmetric]* **by** *auto*

have (*E* \neq *C-Clause* {#} \wedge *satisfiable* (*set* (*map mset* ?*N*)))

\vee (*E* = *C-Clause* {#} \wedge *unsatisfiable* (*set* (*map mset* ?*N*)))

using *full-cdcl_W-stgy-final-state-conclusive* **unfolding** *N'* **apply** *rule*

using 1 **apply** *simp*

using 2 **apply** *simp*

using 3 **apply** *simp*

using *S[symmetric]* *N'* **apply** *auto*[1]

using *S[symmetric]* *N'* **cons** **by** (*fastforce simp: true-annots-true-cls*)

then show ?*thesis* **by** *auto*

qed

The Code The SML code is skipped in the documentation, but stays to ensure that some version of the exported code is working

end

theory *CDCL-WNOT*

imports *CDCL-W-Termination CDCL-NOT*

begin

19 Link between Weidenbach's and NOT's CDCL

19.1 Inclusion of the states

```

declare upt.simps(2)[simp del]
sledgehammer-params[verbose]

context cdclW-ops
begin

lemma backtrack-levE:
  backtrack S S'  $\implies$  cdclW-M-level-inv S  $\implies$ 
  ( $\bigwedge D L K M1 M2$ .
    (Marked K (Suc (get-maximum-level D (trail S))) # M1, M2)
     $\in$  set (get-all-marked-decomposition (trail S))  $\implies$ 
    get-level L (trail S) = get-maximum-level (D + {#L#}) (trail S)  $\implies$ 
    undefined-lit M1 L  $\implies$ 
    S'  $\sim$  cons-trail (Propagated L (D + {#L#}))
    (reduce-trail-to M1 (add-learned-cls (D + {#L#}))
      (update-backtrack-lvl (get-maximum-level D (trail S)) (update-conflicting C-True S))))  $\implies$ 
    backtrack-lvl S = get-maximum-level (D + {#L#}) (trail S)  $\implies$ 
    conflicting S = C-Clause (D + {#L#})  $\implies$  P  $\implies$ 
    P
  )
using assms by (induction rule: backtrack-induction-lev2)metis

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-conflicting cdclW-M-level-inv-decomp)
done

abbreviation skip-or-resolve :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
skip-or-resolve  $\equiv$  ( $\lambda S T$ . skip S T  $\vee$  resolve S T)

lemma rtrancp-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 rtrancp-unfold by blast
then show ?case
  proof cases

```

```

case SUp
have  $\bigwedge T. \text{skip-or-resolve}^{**} S T \implies \text{cdcl}_W^{**} S T$ 
  using mono-rtrancpl[of skip-or-resolve cdclW] other by blast
then have  $\text{skip-or-resolve}^{**} S U$ 
  using bj IH inv backtrack-no-cdclW-bj rtrancpl-cdclW-consistent-inv[OF - inv] by meson
then show ?thesis
  using bj by (metis (no-types, lifting) cdclW-bj.cases rtrancpl.simps)
next
case SU
then show ?thesis
  using bj by (metis (no-types, lifting) cdclW-bj.cases rtrancpl.simps)
qed
qed

```

lemma *rtrancpl-skip-or-resolve-rtrancpl-cdcl_W*:
 $\text{skip-or-resolve}^{**} S T \implies \text{cdcl}_W^{**} S T$
by (*induction rule: rtrancpl-induct*) (*auto dest!: cdcl_W-bj.intros cdcl_W.intros cdcl_W-o.intros*)

definition *backjump-l-cond* :: '*v clause* \Rightarrow '*v clause* \Rightarrow '*v literal* \Rightarrow '*st* \Rightarrow *bool* **where**
backjump-l-cond $\equiv \lambda C C' L' S. \text{True}$

definition *inv_{NOT}* :: '*st* \Rightarrow *bool* **where**
inv_{NOT} $\equiv \lambda S. \text{no-dup (trail } S)$

declare *inv_{NOT}-def*[*simp*]
end

fun *convert-marked-lit-from-W* **where**
convert-marked-lit-from-W (*Propagated L -*) = *Propagated L ()* |
convert-marked-lit-from-W (*Marked L -*) = *Marked L ()*

abbreviation *convert-trail-from-W* ::
 ('*v*, '*vl*, '*a*) *marked-lit list*
 \Rightarrow ('*v*, *unit*, *unit*) *marked-lit list* **where**
convert-trail-from-W $\equiv \text{map convert-marked-lit-from-W}$

lemma *atm-convert-trail-from-W*[*simp*]:
 $(\lambda l. \text{atm-of (lit-of } l)) \text{ ' set (convert-trail-from-W } xs) = (\lambda l. \text{atm-of (lit-of } l)) \text{ ' set } xs$
by (*induction rule: marked-lit-list-induct*) *simp-all*

lemma *lits-of-convert-trail-from-W*[*simp*]:
 $\text{lits-of (convert-trail-from-W } M) = \text{lits-of } M$
by (*induction rule: marked-lit-list-induct*) *simp-all*

lemma *lit-of-convert-trail-from-W*[*simp*]:
 $\text{lit-of (convert-marked-lit-from-W } L) = \text{lit-of } L$
by (*cases L*) *auto*

lemma *no-dup-convert-from-W*[*simp*]:
 $\text{no-dup (convert-trail-from-W } M) \longleftrightarrow \text{no-dup } M$
by (*auto simp: comp-def*)

lemma *convert-trail-from-W-true-annots*[*simp*]:
 $\text{convert-trail-from-W } M \models_{\text{as}} C \longleftrightarrow M \models_{\text{as}} C$
by (*auto simp: true-annots-true-cl*)

lemma *defined-lit-convert-trail-from-W*[simp]:
defined-lit (convert-trail-from-W S) L \longleftrightarrow defined-lit S L
by (auto simp: *defined-lit-map image-comp*)

The values 0 and $\{\#\}$ are dummy values.

fun *convert-marked-lit-from-NOT*
 $:: ('a, 'e, 'b) \text{ marked-lit} \Rightarrow ('a, \text{nat}, 'a \text{ literal multiset}) \text{ marked-lit}$ **where**
convert-marked-lit-from-NOT (Propagated L -) = Propagated L $\{\#\}$ |
convert-marked-lit-from-NOT (Marked L -) = Marked L 0

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT \equiv map convert-marked-lit-from-NOT

lemma *convert-trail-from-W-from-NOT*[simp]:
convert-trail-from-W (convert-trail-from-NOT M) = M
by (induction rule: *marked-lit-list-induct*) auto

lemma *convert-trail-from-W-convert-lit-from-NOT*[simp]:
convert-marked-lit-from-W (convert-marked-lit-from-NOT L) = L
by (cases L) auto

abbreviation *trail_{NOT}* **where**
trail_{NOT} S \equiv convert-trail-from-W (fst S)

lemma *undefined-lit-convert-trail-from-W*[iff]:
undefined-lit (convert-trail-from-W M) L \longleftrightarrow undefined-lit M L
by (auto simp: *defined-lit-map image-comp*)

lemma *lit-of-convert-marked-lit-from-NOT*[iff]:
lit-of (convert-marked-lit-from-NOT L) = lit-of L
by (cases L) auto

sublocale *state_W \subseteq dpll-state*
 $\lambda S. \text{ convert-trail-from-W (trail S)}$
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-simp_{NOT}*[simp del]
end

sublocale *cdcl_W-ops \subseteq cdcl_{NOT}-merge-bj-learn-ops*
 $\lambda S. \text{ convert-trail-from-W (trail S)}$
clauses
 $\lambda L S. \text{ cons-trail (convert-marked-lit-from-NOT L) S}$
 $\lambda S. \text{ tl-trail S}$
 $\lambda C S. \text{ add-learned-cls C S}$
 $\lambda C S. \text{ remove-cls C S}$
 $\lambda - . \text{ True}$

λ - S . conflicting $S = C\text{-True}$
 $\lambda C\ C'\ L'\ S$. backjump-l-cond $C\ C'\ L'\ S \wedge \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$
by *unfold-locals*

sublocale $\text{cdcl}_W\text{-ops} \subseteq \text{cdcl}_{NOT}\text{-merge-bj-learn-proxy}$
 λS . *convert-trail-from-W* (*trail* S)
clauses
 $\lambda L\ S$. *cons-trail* (*convert-marked-lit-from-NOT* L) S
 λS . *tl-trail* S
 $\lambda C\ S$. *add-learned-cls* $C\ S$
 $\lambda C\ S$. *remove-cls* $C\ S$
 λ - \cdot . *True*
 λ - S . conflicting $S = C\text{-True}$ *backjump-l-cond* *inv* $_{NOT}$

proof (*unfold-locals*, *goal-cases*)
case 2
then show ?*case* **using** $\text{cdcl}_{NOT}\text{-merged-bj-learn-no-dup-inv}$ **by** (*auto simp: comp-def*)
next
case (1 $C'\ S\ C\ F'\ K\ F\ L$)
moreover
let ? $C' = \text{remdups-mset } C'$
have $L \notin \# C'$
using $\langle F \models_{as} C\text{Not } C' \rangle \langle \text{undefined-lit } F\ L \rangle \text{Marked-Propagated-in-iff-in-lits-of}$
in-CNot-implies-uminus(2) **by** *blast*
then have $\text{distinct-mset } (?C' + \{\#L'\#\})$
by (*metis count-mset-set*(3) *distinct-mset-remdups-mset distinct-mset-single-add*
less-irrefl-nat mem-set-mset-iff remdups-mset-def)
moreover
have *no-dup* F
using $\langle \text{inv}_{NOT}\ S \rangle \langle \text{convert-trail-from-W } (\text{trail } S) = F' @ \text{Marked } K\ () \# F \rangle$
unfolding *inv* $_{NOT}\text{-def}$
by (*smt comp-apply distinct.simps*(2) *distinct-append list.simps*(9) *map-append*
no-dup-convert-from-W)
then have *consistent-interp* (*lits-of* F)
using *distinctconsistent-interp* **by** *blast*
then have $\neg \text{tautology } (C')$
using $\langle F \models_{as} C\text{Not } C' \rangle \text{consistent-CNot-not-tautology true-annots-true-cls}$ **by** *blast*
then have $\neg \text{tautology } (?C' + \{\#L'\#\})$
using $\langle F \models_{as} C\text{Not } C' \rangle \langle \text{undefined-lit } F\ L \rangle$ **by** (*metis CNot-remdups-mset*
Marked-Propagated-in-iff-in-lits-of add commute in-CNot-uminus tautology-add-single
tautology-remdups-mset true-annot-singleton true-annots-def)
show ?*case*
proof –
have $f2$: *no-dup* (*convert-trail-from-W* (*trail* S))
using $\langle \text{inv}_{NOT}\ S \rangle$ **unfolding** *inv* $_{NOT}\text{-def}$ **by** (*simp add: o-def*)
have $f3$: *atm-of* $L \in \text{atms-of-msu } (\text{clauses } S)$
 $\cup \text{atm-of 'lits-of } (\text{convert-trail-from-W } (\text{trail } S))$
using $\langle \text{convert-trail-from-W } (\text{trail } S) = F' @ \text{Marked } K\ () \# F \rangle$
 $\langle \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of 'lits-of } (F' @ \text{Marked } K\ () \# F) \rangle$ **by** *auto*
have $f4$: $\text{clauses } S \models_{pm} \text{remdups-mset } C' + \{\#L'\#\}$
by (*metis (no-types) L notin C' clauses S models pm C' + {L} remdups-mset-singleton-sum*(2)
true-clss-cls-remdups-mset union-commute)
have $F \models_{as} C\text{Not } (\text{remdups-mset } C')$
by (*simp add: F models as CNot C'*)
then show ?*thesis*
using $f4\ f3\ f2 \hookrightarrow \text{tautology } (\text{remdups-mset } C' + \{\#L'\#\})$


```

      backjump-l.intros[OF - f2] calculation(2-5,9)
      state-eqNOT-ref unfolding backjump-l-cond-def by blast
    qed
  qed

sublocale cdclW-ops  $\subseteq$  cdclNOT-merge-bj-learn-proxy2
   $\lambda S.$  convert-trail-from-W (trail S)
  clauses
   $\lambda L S.$  cons-trail (convert-marked-lit-from-NOT L) S
   $\lambda S.$  tl-trail S
   $\lambda C S.$  add-learned-cls C S
   $\lambda C S.$  remove-cls C S  $\lambda -.$  True invNOT
   $\lambda - S.$  conflicting S = C-True backjump-l-cond
  by unfold-locales

sublocale cdclW-ops  $\subseteq$  cdclNOT-merge-bj-learn
   $\lambda S.$  convert-trail-from-W (trail S)
  clauses
   $\lambda L S.$  cons-trail (convert-marked-lit-from-NOT L) S
   $\lambda S.$  tl-trail S
   $\lambda C S.$  add-learned-cls C S
   $\lambda C S.$  remove-cls C S  $\lambda -.$  True invNOT
   $\lambda - S.$  conflicting S = C-True backjump-l-cond
  apply unfold-locales
  using dpll-bj-no-dup apply (simp add: comp-def)
  using cdclNOT-no-dup by (auto simp add: comp-def cdclNOT.simps)

```

```

context cdclW-ops
begin

```

Notations are lost while proving locale inclusion:

```

notation state-eqNOT (infix  $\sim_{NOT}$  50)

```

19.2 Additional Lemmas between NOT and W states

```

lemma trailW-eq-reduce-trail-toNOT-eq:
  trail S = trail T  $\implies$  trail (reduce-trail-toNOT F S) = trail (reduce-trail-toNOT F T)
proof (induction F S arbitrary: T rule: reduce-trail-toNOT.induct)
  case (1 F S T) note IH = this(1) and tr = this(2)
  then have [] = convert-trail-from-W (trail S)
     $\vee$  length F = length (convert-trail-from-W (trail S))
     $\vee$  trail (reduce-trail-toNOT F (tl-trail S)) = trail (reduce-trail-toNOT F (tl-trail T))
  using IH by (metis (no-types) trail-tl-trail)
  then show trail (reduce-trail-toNOT F S) = trail (reduce-trail-toNOT F T)
    using tr by (metis (no-types) reduce-trail-toNOT.elim)
  qed

```

```

lemma trail-reduce-trail-toNOT-add-learned-cls[simp]:
  no-dup (trail S)  $\implies$ 
  trail (reduce-trail-toNOT M (add-learned-cls D S)) = trail (reduce-trail-toNOT M S)
  by (rule trailW-eq-reduce-trail-toNOT-eq) simp

```

```

lemma reduce-trail-toNOT-reduce-trail-convert:
  reduce-trail-toNOT C S = reduce-trail-to (convert-trail-from-NOT C) S
  apply (induction C S rule: reduce-trail-toNOT.induct)
  apply (subst reduce-trail-toNOT.simps, subst reduce-trail-to.simps)

```

by auto

lemma *reduce-trail-to-length*:

length M = length M' \implies reduce-trail-to M S = reduce-trail-to M' S
apply (*induction M S arbitrary: rule: reduce-trail-to.induct*)
apply (*case-tac trail S $\neq []$; case-tac length (trail S) \neq length M'; simp*)
by (*simp-all add: reduce-trail-to-length-ne*)

19.3 More lemmas conflict-propagate and backjumping

19.3.1 Termination

lemma *cdcl_W-cp-normalized-element-all-inv*:

assumes *inv: cdcl_W-all-struct-inv S*
obtains *T where full cdcl_W-cp S T*
using *assms cdcl_W-cp-normalized-element unfolding cdcl_W-all-struct-inv-def by blast*
thm *backtrackE*

lemma *cdcl_W-bj-measure*:

assumes *cdcl_W-bj S T and cdcl_W-M-level-inv S*
shows *length (trail S) + (if conflicting S = C-True then 0 else 1)*
> length (trail T) + (if conflicting T = C-True then 0 else 1)
using *assms by (induction rule: cdcl_W-bj.induct)*
(force dest:arg-cong[of - - length]
intro: get-all-marked-decomposition-exists-prepend
elim!: backtrack-levE
simp: cdcl_W-M-level-inv-def)+

lemma *wf-cdcl_W-bj*:

wf {(b,a). cdcl_W-bj a b \wedge cdcl_W-M-level-inv a}
apply (*rule wfP-if-measure[of λ -. True*
- λT . length (trail T) + (if conflicting T = C-True then 0 else 1), simplified])
using *cdcl_W-bj-measure by blast*

lemma *cdcl_W-bj-exists-normal-form*:

assumes *lev: cdcl_W-M-level-inv S*
shows $\exists T$. *full cdcl_W-bj S T*
proof –
obtain *T where T: full (λa b. cdcl_W-bj a b \wedge cdcl_W-M-level-inv a) S T*
using *wf-exists-normal-form-full[OF wf-cdcl_W-bj] by auto*
then have *cdcl_W-bj** S T*
by (*auto dest: rtrancp-and-rtrancp-left simp: full-def*)

moreover

then have *cdcl_W** S T*
using *mono-rtrancp[of cdcl_W-bj cdcl_W] cdcl_W.simps by blast*
then have *cdcl_W-M-level-inv T*
using *rtrancp-cdcl_W-consistent-inv lev by auto*
ultimately show *?thesis using T unfolding full-def by auto*

qed

lemma *rtrancp-skip-state-decomp*:

assumes *skip** S T and no-dup (trail S)*
shows
 $\exists M$. *trail S = M @ trail T \wedge ($\forall m \in \text{set } M$. $\neg \text{is-marked } m$) and*
T \sim delete-trail-and-rebuild (trail T) S
using *assms by (induction rule: rtrancp-induct) (auto simp del: state-simp simp: state-eq-def)+*

19.3.2 More backjumping

Backjumping after skipping or jump directly lemma *rtrancpl-skip-backtrack-backtrack*:

```

assumes
  skip** S T and
  backtrack T W and
  cdclW-all-struct-inv S
shows backtrack S W
using assms
proof induction
  case base
  then show ?case by simp
next
  case (step T V) note st = this(1) and skip = this(2) and IH = this(3) and bt = this(4) and
    inv = this(5)
  have skip** S V
  using st skip by auto
  then have cdclW-all-struct-inv V
  using rtrancpl-mono[of skip cdclW] assms(3) rtrancpl-cdclW-all-struct-inv-inv mono-rtrancpl
  by (auto dest!: bj other cdclW-bj.skip)
  then have cdclW-M-level-inv V
  unfolding cdclW-all-struct-inv-def by auto
  then obtain N k M1 M2 K D L U i where
    V: state V = (trail V, N, U, k, C-Clause (D + {#L#})) and
    W: state W = (Propagated L (D + {#L#}) # M1, N, {#D + {#L#}#} + U,
      get-maximum-level D (trail V), C-True) and
    decomp: (Marked K (Suc i) # M1, M2)
      ∈ set (get-all-marked-decomposition (trail V)) and
    k = get-maximum-level (D + {#L#}) (trail V) and
    lev-L: get-level L (trail V) = k and
    undef: undefined-lit M1 L and
    W ~ cons-trail (Propagated L (D + {#L#}))
      (reduce-trail-to M1 (add-learned-cls (D + {#L#}))
        (update-backtrack-lvl (get-maximum-level D (trail V)) (update-conflicting C-True V))) and
    lev-l-D: backtrack-lvl V = get-maximum-level (D + {#L#}) (trail V) and
    conflicting V = C-Clause (D + {#L#}) and
    i: i = get-maximum-level D (trail V)
  using bt by (elim backtrack-levE) (auto simp: cdclW-M-level-inv-decomp)
  let ?D = (D + {#L#})
  obtain L' C' where
    T: state T = (Propagated L' C' # trail V, N, U, k, C-Clause ?D) and
    V ~ tl-trail T and
    -L' ∉ # ?D and
    ?D ≠ {#}
  using skip V by force

  let ?M = Propagated L' C' # trail V
  have cdclW** S T using bj cdclW-bj.skip mono-rtrancpl[of skip cdclW S T] other st by meson
  then have inv': cdclW-all-struct-inv T
  using rtrancpl-cdclW-all-struct-inv-inv inv by blast
  have M-lev: cdclW-M-level-inv T using inv' unfolding cdclW-all-struct-inv-def by auto
  then have n-d': no-dup ?M
  using T unfolding cdclW-M-level-inv-def by auto

  have k > 0
  using decomp M-lev T V unfolding cdclW-M-level-inv-def by auto

```

```

then have atm-of L ∈ atm-of ‘ lits-of (trail V)
  using lev-L get-rev-level-ge-0-atm-of-in V by fastforce
then have L-L': atm-of L ≠ atm-of L'
  using n-d' unfolding lits-of-def by auto
have L'-M: atm-of L' ∉ atm-of ‘ lits-of (trail V)
  using n-d' unfolding lits-of-def by auto
have ?M ⊨as CNot ?D
  using inv' T unfolding cdclW-conflicting-def cdclW-all-struct-inv-def by auto
then have L' ∉ # ?D
  using L-L' L'-M unfolding true-annots-def by (auto simp add: true-annot-def true-cls-def
    atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set Ball-mset-def
    split: split-if-asm)
have [simp]: trail (reduce-trail-to M1 T) = M1
  by (metis (mono-tags, lifting) One-nat-def Pair-inject T ⟨V ∼ tl-trail T⟩ decomp
    diff-less in-get-all-marked-decomposition-trail-update-trail length-greater-0-conv
    length-tl lessI list.distinct(1) reduce-trail-to-length-ne state-eq-trail
    trail-reduce-trail-to-length-le trail-tl-trail)
have skip** S V
  using st skip by auto
have no-dup (trail S)
  using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
then have [simp]: init-cls S = N and [simp]: learned-cls S = U
  using rtrnclp-skip-state-decomp[OF ⟨skip** S V⟩] V
  by (auto simp del: state-simp simp: state-eq-def)
then have W-S: W ∼ cons-trail (Propagated L (D + {#L#})) (reduce-trail-to M1
  (add-learned-cls (D + {#L#}) (update-backtrack-lvl i (update-conflicting C-True T))))
  using W i T undef M-lev by (auto simp del: state-simp simp: state-eq-def cdclW-M-level-inv-def)

obtain M2' where
  (Marked K (i+1) # M1, M2') ∈ set (get-all-marked-decomposition ?M)
  using decomp V by (cases hd (get-all-marked-decomposition (trail V)),
    cases get-all-marked-decomposition (trail V)) auto
moreover
  from L-L' have get-level L ?M = k
    using lev-L ⟨-L' ∉ # ?D⟩ V by (auto split: split-if-asm)
moreover
  have atm-of L' ∉ atms-of D
    using ⟨L' ∉ # ?D⟩ ⟨-L' ∉ # ?D⟩ by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      atms-of-def)
  then have get-level L ?M = get-maximum-level (D + {#L#}) ?M
    using lev-l-D[symmetric] L-L' V lev-L by simp
moreover have i = get-maximum-level D ?M
  using i ⟨atm-of L' ∉ atms-of D⟩ by auto
moreover

ultimately have backtrack T W
  using T(1) W-S by blast
then show ?thesis using IH inv by blast
qed

```

```

lemma fst-get-all-marked-decomposition-prepend-not-marked:
  assumes ∀ m ∈ set MS. ¬ is-marked m
  shows set (map fst (get-all-marked-decomposition M))
    = set (map fst (get-all-marked-decomposition (MS @ M)))
  using assms apply (induction MS rule: marked-lit-list-induct)

```

apply *auto*[2]
by (*case-tac get-all-marked-decomposition* (*xs @ M*)) *simp-all*

See also $\llbracket \text{skip}^{**} ?S ?T; \text{backtrack} ?T ?W; \text{cdcl}_W\text{-all-struct-inv} ?S \rrbracket \implies \text{backtrack} ?S ?W$

lemma *rtrancpl-skip-backtrack-backtrack-end*:

assumes

skip: *skip*^{**} *S T* **and**

bt: *backtrack S W* **and**

inv: *cdcl_W-all-struct-inv S*

shows *backtrack T W*

using *assms*

proof –

have *M-lev*: *cdcl_W-M-level-inv S*

using *bt inv unfolding cdcl_W-all-struct-inv-def* **by** (*auto elim!*: *backtrack-levE*)

then obtain *k M M1 M2 K i D L N U* **where**

S: *state S = (M, N, U, k, C-Clause (D + {#L#}))* **and**

W: *state W = (Propagated L ((D + {#L#})) # M1, N, {#D + {#L#}#} + U,*

get-maximum-level D M, C-True) **and**

decomp: *(Marked K (i+1) # M1, M2) ∈ set (get-all-marked-decomposition M)* **and**

lev-l: *get-level L M = k* **and**

lev-l-D: *get-level L M = get-maximum-level (D+{#L#}) M* **and**

i: *i = get-maximum-level D M* **and**

undef: *undefined-lit M1 L*

using *bt* **by** (*elim backtrack-levE*) (*force simp: cdcl_W-M-level-inv-def*) +

let *?D = (D + {#L#})*

have [*simp*]: *no-dup (trail S)*

using *M-lev* **by** (*auto simp: cdcl_W-M-level-inv-decomp*)

have *cdcl_W-all-struct-inv T*

using *mono-rtrancpl[of skip cdcl_W]* **by** (*smt bj cdcl_W-bj.skip inv local.skip other*
rtrancpl-cdcl_W-all-struct-inv-inv)

then have [*simp*]: *no-dup (trail T)*

unfolding *cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def* **by** *auto*

obtain *MS M_T* **where** *M: M = MS @ M_T* **and** *M_T: M_T = trail T* **and** *nm: ∀ m ∈ set MS. ¬is-marked*
m

using *rtrancpl-skip-state-decomp(1)[OF skip] S M-lev* **by** *auto*

have *T*: *state T = (M_T, N, U, k, C-Clause ?D)*

using *M_T rtrancpl-skip-state-decomp(2)[of S T] skip S*

by (*auto simp del: state-simp simp: state-eq-def*)

have *cdcl_W-all-struct-inv T*

apply (*rule rtrancpl-cdcl_W-all-struct-inv-inv[OF - inv]*)

using *bj cdcl_W-bj.skip local.skip other rtrancpl-mono[of skip cdcl_W]* **by** *blast*

then have *M_T ⊨_{as} CNot ?D*

unfolding *cdcl_W-all-struct-inv-def cdcl_W-conflicting-def* **using** *T* **by** *blast*

have $\forall L \in \#?D. \text{atm-of } L \in \text{atm-of } \text{'lits-of } M_T$

proof –

have *f1*: $\bigwedge l. \neg M_T \models_a \{ \# - l \# \} \vee \text{atm-of } l \in \text{atm-of } \text{'lits-of } M_T$

by (*simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-lit-of-true-annot*
lits-of-def)

have $\bigwedge l. l \notin \# D \vee - l \in \text{lits-of } M_T$

using $\langle M_T \models_{as} CNot (D + \{ \# L \# \}) \rangle$ *multi-member-split* **by** *fastforce*

then show *?thesis*

using *f1* **by** (*meson* $\langle M_T \models_{as} CNot (D + \{ \# L \# \}) \rangle$ *ball-msetI true-annots-CNot-all-atms-defined*)

```

qed
moreover have no-dup M
  using inv S unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
ultimately have  $\forall L \in \#D. \text{atm-of } L \notin \text{atm-of ' lits-of } MS$ 
  unfolding M unfolding lits-of-def by auto
then have H:  $\bigwedge L. L \in \#D \implies \text{get-level } L M = \text{get-level } L M_T$ 
  unfolding M by (fastforce simp: lits-of-def)
have [simp]:  $\text{get-maximum-level } ?D M = \text{get-maximum-level } ?D M_T$ 
  by (metis  $\langle M_T \models_{as} CNot (D + \{\#L\# \}) \rangle M \text{ nm ball-msetI true-annots-CNot-all-atms-defined}$ 
    get-maximum-level-skip-un-marked-not-present)

have lev-l':  $\text{get-level } L M_T = k$ 
  using lev-l by (auto simp: H)
have [simp]:  $\text{trail } (\text{reduce-trail-to } M1 T) = M1$ 
  using T decomp M nm by (smt M_T append-assoc beginning-not-marked-invert
    get-all-marked-decomposition-exists-prepend reduce-trail-to-trail-tl-trail-decomp)
have W:  $W \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \})) (\text{reduce-trail-to } M1$ 
   $(\text{add-learned-cls } (D + \{\#L\# \})) (\text{update-backtrack-lvl } i (\text{update-conflicting } C\text{-True } T))))$ 
  using W T i decomp undef by (auto simp del: state-simp simp: state-eq-def)

have lev-l-D':  $\text{get-level } L M_T = \text{get-maximum-level } (D + \{\#L\# \}) M_T$ 
  using lev-l-D by (auto simp: H)
have [simp]:  $\text{get-maximum-level } D M = \text{get-maximum-level } D M_T$ 
proof -
  have  $\bigwedge ms m. \neg (ms::('v, nat, 'v \text{ literal multiset}) \text{ marked-lit list}) \models_{as} CNot m$ 
     $\vee (\forall l \in \#m. \text{atm-of } l \in \text{atm-of ' lits-of } ms)$ 
    by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-CNot-implies-uminus(2))
  then have  $\forall l \in \#D. \text{atm-of } l \in \text{atm-of ' lits-of } M_T$ 
    using  $\langle M_T \models_{as} CNot (D + \{\#L\# \}) \rangle$  by auto
  then show ?thesis
    by (metis M get-maximum-level-skip-un-marked-not-present nm)
qed
then have i':  $i = \text{get-maximum-level } D M_T$ 
  using i by auto
have Marked K (i + 1) # M1  $\in \text{set } (\text{map fst } (\text{get-all-marked-decomposition } M))$ 
  using Set.imageI[OF decomp, of fst] by auto
then have Marked K (i + 1) # M1  $\in \text{set } (\text{map fst } (\text{get-all-marked-decomposition } M_T))$ 
  using fst-get-all-marked-decomposition-prepend-not-marked[OF nm] unfolding M by auto
then obtain M2' where  $\text{decomp':}(\text{Marked } K (i+1) \# M1, M2') \in \text{set } (\text{get-all-marked-decomposition } M_T)$ 
  by auto
then show backtrack T W
  using backtrack.intros[OF T decomp' lev-l'] lev-l-D' i' W by force
qed

lemma cdclW-bj-decomp-resolve-skip-and-bj:
  assumes cdclW-bj** S T and inv: cdclW-M-level-inv S
  shows (skip-or-resolve** S T
     $\vee (\exists U. \text{skip-or-resolve** } S U \wedge \text{backtrack } U T))$ 
  using assms
proof induction
  case base
  then show ?case by simp
next
  case (step T U) note st = this(1) and bj = this(2) and IH = this(3)

```

```

have IH: skip-or-resolve** S T
proof -
  { assume ( $\exists U. \text{skip-or-resolve}^{**} S U \wedge \text{backtrack } U T$ )
    then obtain V where
      bt: backtrack V T and
      skip-or-resolve** S V
      by blast
    have cdclW** S V
      using  $\langle \text{skip-or-resolve}^{**} S V \rangle \text{rtrancpl-skip-or-resolve-rtrancpl-cdcl}_W$  by blast
    then have cdclW-M-level-inv V and cdclW-M-level-inv S
      using rtrancpl-cdclW-consistent-inv inv by blast+
    with bj bt have False using backtrack-no-cdclW-bj by simp
  }
  then show ?thesis using IH inv by blast
qed
show ?case
using bj
proof (cases rule: cdclW-bj.cases)
  case backtrack
  then show ?thesis using IH by blast
qed (metis (no-types, lifting) IH rtrancpl.simps)+
qed

lemma resolve-skip-deterministic:
  resolve S T  $\implies$  skip S U  $\implies$  False
by fastforce

lemma backtrack-unique:
  assumes
    bt-T: backtrack S T and
    bt-U: backtrack S U and
    inv: cdclW-all-struct-inv S
  shows T  $\sim$  U
proof -
  have lev: cdclW-M-level-inv S
    using inv unfolding cdclW-all-struct-inv-def by auto
  then obtain M N U' k D L i K M1 M2 where
    S: state S = (M, N, U', k, C-Clause (D + {#L#})) and
    decomp: (Marked K (i+1) # M1, M2)  $\in$  set (get-all-marked-decomposition M) and
    get-level L M = k and
    get-level L M = get-maximum-level (D+{#L#}) M and
    get-maximum-level D M = i and
    T: state T = (Propagated L ((D+{#L#})) # M1, N, {#D + {#L#}#} + U', i, C-True) and
    undef: undefined-lit M1 L
    using bt-T by (elim backtrack-levE) (force simp: cdclW-M-level-inv-def)+

  obtain D' L' i' K' M1' M2' where
    S': state S = (M, N, U', k, C-Clause (D' + {#L'#})) and
    decomp': (Marked K' (i'+1) # M1', M2')  $\in$  set (get-all-marked-decomposition M) and
    get-level L' M = k and
    get-level L' M = get-maximum-level (D'+{#L'#}) M and
    get-maximum-level D' M = i' and
    U: state U = (Propagated L' ((D'+{#L'#})) # M1', N, {#D' + {#L'#}#} + U', i', C-True) and
    undef: undefined-lit M1' L'
    using bt-U lev S by (elim backtrack-levE) (force simp: cdclW-M-level-inv-def)+

```

```

obtain  $c$  where  $M: M = c @ M2 @ \text{Marked } K (i + 1) \# M1$ 
  using decomp by auto
obtain  $c'$  where  $M': M = c' @ M2' @ \text{Marked } K' (i' + 1) \# M1'$ 
  using decomp' by auto
have marked: get-all-levels-of-marked  $M = \text{rev } [1..<1+k]$ 
  using inv S unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
then have  $i < k$ 
  unfolding  $M$ 
  by (force simp add: rev-swap[symmetric] dest!: arg-cong[of - - set])

have [simp]:  $L = L'$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then have  $L' \in \# D$ 
    using S unfolding S' by (fastforce simp: multiset-eq-iff split: split-if-asm)
  then have get-maximum-level  $D M \geq k$ 
    using  $\langle \text{get-level } L' M = k \rangle$  get-maximum-level-ge-get-level by blast
  then show False using  $\langle \text{get-maximum-level } D M = i \rangle \langle i < k \rangle$  by auto
qed
then have [simp]:  $D = D'$ 
  using S S' by auto
have [simp]:  $i=i'$  using  $\langle \text{get-maximum-level } D' M = i' \rangle \langle \text{get-maximum-level } D M = i \rangle$  by auto

```

Automation in a step later...

```

have  $H: \bigwedge a A B. \text{insert } a A = B \implies a : B$ 
  by blast
have get-all-levels-of-marked  $(c @ M2) = \text{rev } [i+2..<1+k]$  and
  get-all-levels-of-marked  $(c' @ M2') = \text{rev } [i+2..<1+k]$ 
  using marked unfolding M
  using marked unfolding M'
  unfolding rev-swap[symmetric] by (auto dest: append-cons-eq-upt-length-i-end)
from arg-cong[OF this(1), of set] arg-cong[OF this(2), of set]
have
  dropWhile  $(\lambda L. \neg \text{is-marked } L \vee \text{level-of } L \neq \text{Suc } i) (c @ M2) = []$  and
  dropWhile  $(\lambda L. \neg \text{is-marked } L \vee \text{level-of } L \neq \text{Suc } i) (c' @ M2') = []$ 
  unfolding dropWhile-eq-Nil-conv Ball-def
  by (intro allI; case-tac x; auto dest!: H simp add: in-set-conv-decomp) +

then have  $M1 = M1'$ 
  using arg-cong[OF M, of dropWhile  $(\lambda L. \neg \text{is-marked } L \vee \text{level-of } L \neq \text{Suc } i)$ 
  unfolding  $M'$  by auto
then show ?thesis using  $T U$  by (auto simp del: state-simp simp: state-eq-def)
qed

```

lemma *if-can-apply-backtrack-no-more-resolve:*

```

assumes
  skip: skip** S U and
  bt: backtrack S T and
  inv: cdclW-all-struct-inv S
shows  $\neg \text{resolve } U V$ 
proof (rule ccontr)
  assume resolve:  $\neg \neg \text{resolve } U V$ 

```

```

obtain  $L C M N U' k D$  where
   $U: \text{state } U = (\text{Propagated } L ( (C + \{\#L\# \}) ) \# M, N, U', k, C\text{-Clause } (D + \{\#-L\# \}))$  and

```


$get_maximum_level\ D\ (Propagated\ L\ ((C + \{\#L\#\})) \# M) = k$ and
 $state\ V = (M, N, U', k, C\text{-}Clause\ (D \# \cup C))$
using *resolve* **by** *auto*
have $cdcl_W\text{-}all\text{-}struct\text{-}inv\ U$
using $mono\text{-}rtrancp[of\ skip\ cdcl_W]$ **by** (*meson* $bj\ cdcl_W\text{-}bj.skip\ inv\ local.skip\ other$
 $rtrancp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$)
then have $[iff]: no\text{-}dup\ (trail\ S)\ cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and** $[iff]: no\text{-}dup\ (trail\ U)$
using $inv\ unfolding\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** *blast+*
then have
 $S: init\text{-}clss\ S = N$
 $learned\text{-}clss\ S = U'$
 $backtrack\text{-}lvl\ S = k$
 $conflicting\ S = C\text{-}Clause\ (D + \{\#-L\#\})$
using $rtrancp\text{-}skip\text{-}state\text{-}decomp(2)[OF\ skip]\ U$ **by** (*auto* *simp* *del: state-simp* *simp: state-eq-def*)
obtain M_0 **where**
 $tr\text{-}S: trail\ S = M_0 @ trail\ U$ **and**
 $nm: \forall m \in set\ M_0. \neg is\text{-}marked\ m$
using $rtrancp\text{-}skip\text{-}state\text{-}decomp[OF\ skip]$ **by** *blast*

obtain $M'\ D'\ L'\ i\ K\ M1\ M2$ **where**
 $S': state\ S = (M', N, U', k, C\text{-}Clause\ (D' + \{\#L'\#\}))$ **and**
 $decomp: (Marked\ K\ (i+1) \# M1, M2) \in set\ (get\text{-}all\text{-}marked\text{-}decomposition\ M')$ **and**
 $get\text{-}level\ L'\ M' = k$ **and**
 $get\text{-}level\ L'\ M' = get\text{-}maximum\text{-}level\ (D' + \{\#L'\#\})\ M'$ **and**
 $get\text{-}maximum\text{-}level\ D'\ M' = i$ **and**
 $undef: undefined\text{-}lit\ M1\ L'$ **and**
 $T: state\ T = (Propagated\ L'\ (D' + \{\#L'\#\}) \# M1, N, \{\#D' + \{\#L'\#\}\# + U', i, C\text{-}True)$
using $bt\ (cdcl_W\text{-}M\text{-}level\text{-}inv\ S)\ S$ **by** (*elim* *backtrack\text{-}levE*) *fastforce+*
obtain c **where** $M: M' = c @ M2 @ Marked\ K\ (i + 1) \# M1$
using $get\text{-}all\text{-}marked\text{-}decomposition\text{-}exists\text{-}prepend[OF\ decomp]$ **by** *auto*
have $marked: get\text{-}all\text{-}levels\text{-}of\text{-}marked\ M' = rev\ [1..<1+k]$
using $inv\ S'\ unfolding\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** *auto*
then have $i < k$
unfolding M **by** (*force* *simp* *add: rev\text{-}swap[symmetric]* *dest!: arg\text{-}cong[of - - set]*)

have $DD': D' + \{\#L'\#\} = D + \{\#-L\#\}$
using $S\ S'$ **by** *auto*
have $[simp]: L' = -L$
proof (*rule* *ccontr*)
assume $\neg ?thesis$
then have $-L \in \# D'$
using DD' **by** (*metis* *add\text{-}diff\text{-}cancel\text{-}right'* *diff\text{-}single\text{-}trivial* *diff\text{-}union\text{-}swap*
 $multi\text{-}self\text{-}add\text{-}other\text{-}not\text{-}self$)
moreover
have $M': M' = M_0 @ Propagated\ L\ ((C + \{\#L\#\})) \# M$
using $tr\text{-}S\ U\ S\ S'$ **by** (*auto* *simp: lits\text{-}of\text{-}def*)
have $no\text{-}dup\ M'$
using $inv\ U\ S'\ unfolding\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** *auto*
have $atm\text{-}L\text{-}notin\text{-}M: atm\text{-}of\ L \notin atm\text{-}of\ ' (lits\text{-}of\ M)$
using $(no\text{-}dup\ M')\ M'\ U\ S\ S'$ **by** (*auto* *simp: lits\text{-}of\text{-}def*)
have $get\text{-}all\text{-}levels\text{-}of\text{-}marked\ M' = rev\ [1..<1+k]$
using $inv\ U\ S'\ unfolding\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def\ cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}def$ **by** *auto*
then have $get\text{-}all\text{-}levels\text{-}of\text{-}marked\ M = rev\ [1..<1+k]$
using $nm\ M'\ S'\ U$ **by** (*simp* *add: get\text{-}all\text{-}levels\text{-}of\text{-}marked\text{-}no\text{-}marked*)
then have $get\text{-}lev\text{-}L:$

$get_level\ L\ (Propagated\ L\ ((C + \{\#L\#\})) \# M) = k$
using $get_level_get_rev_level_get_all_levels_of_marked[OF\ atm_L_notin_M,$
 $of\ [Propagated\ L\ ((C + \{\#L\#\}))]]\ \text{by}\ simp$
have $atm_of\ L \notin atm_of\ ' (lits_of\ (rev\ M_0))$
using $\langle no_dup\ M' \rangle\ M' \ U\ S' \ \text{by}\ (auto\ simp: lits_of_def)$
then have $get_level\ L\ M' = k$
using $get_rev_level_notin_end[of\ L\ rev\ M_0\ 0$
 $rev\ M\ @\ Propagated\ L\ ((C + \{\#L\#\})) \# []]$
using $tr_S\ get_lev_L\ M' \ U\ S' \ \text{by}\ (simp\ add: nm\ lits_of_def)$
ultimately have $get_maximum_level\ D' \ M' \geq k$
by $(metis\ get_maximum_level_ge_get_level\ get_rev_level_uminus)$
then show $False$
using $\langle i < k \rangle\ \text{unfolding}\ \langle get_maximum_level\ D' \ M' = i \rangle\ \text{by}\ auto$
qed
have $[simp]: D = D' \ \text{using}\ DD' \ \text{by}\ auto$
have $cdcl_W^{**}\ S\ U$
using $bj\ cdcl_W_bj.skip\ local.skip\ mono_rtranclp[of\ skip\ cdcl_W\ S\ U]\ other \ \text{by}\ meson$
then have $cdcl_W_all_struct_inv\ U$
using $inv\ rtranclp_cdcl_W_all_struct_inv_inv \ \text{by}\ blast$
then have $Propagated\ L\ ((C + \{\#L\#\})) \# M \models_{as} CNot\ (D' + \{\#L'\#\})$
using $cdcl_W_all_struct_inv_def\ cdcl_W_conflicting_def\ U \ \text{by}\ auto$
then have $\forall L' \in \#D. atm_of\ L' \in atm_of\ ' lits_of\ (Propagated\ L\ ((C + \{\#L\#\})) \# M)$
by $(metis\ CNot_plus\ CNot_singleton\ Un_insert_right\ \langle D = D' \rangle\ true_annots_insert\ ball_msetI$
 $atm_of_in_atm_of_set_iff_in_set_or_uminus_in_set\ in_CNot_implies_uminus(2)$
 $sup_bot.comm_neutral)$
then have $get_maximum_level\ D\ M' = k$
using $tr_S\ nm\ U\ S'$
 $get_maximum_level_skip_un_marked_not_present[of\ D$
 $Propagated\ L\ ((C + \{\#L\#\})) \# M\ M_0]$
unfolding $\langle get_maximum_level\ D\ (Propagated\ L\ ((C + \{\#L\#\})) \# M) = k \rangle$
unfolding $\langle D = D' \rangle$
by $simp$
show $False$
using $\langle get_maximum_level\ D' \ M' = i \rangle\ \langle get_maximum_level\ D\ M' = k \rangle\ \langle i < k \rangle \ \text{by}\ auto$
qed

lemma *if-can-apply-resolve-no-more-backtrack:*

assumes
 $skip: skip^{**}\ S\ U$ **and**
 $resolve: resolve\ S\ T$ **and**
 $inv: cdcl_W_all_struct_inv\ S$
shows $\neg backtrack\ U\ V$
using $assms$
by $(meson\ if_can_apply_backtrack_no_more_resolve\ rtranclp.rtrancl_refl$
 $rtranclp_skip_backtrack_backtrack)$

lemma *if-can-apply-backtrack-skip-or-resolve-is-skip:*

assumes
 $bt: backtrack\ S\ T$ **and**
 $skip: skip_or_resolve^{**}\ S\ U$ **and**
 $inv: cdcl_W_all_struct_inv\ S$
shows $skip^{**}\ S\ U$
using $assms(2,3,1)$
by $induction\ (simp_all\ add: if_can_apply_backtrack_no_more_resolve)$

lemma *cdcl_W-bj-bj-decomp*:

assumes *cdcl_W-bj** S W* **and** *cdcl_W-all-struct-inv S*

shows

$(\exists T U V. (\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S)^{**} S T$
 $\wedge (\lambda T U. \text{resolve } T U \wedge \text{no-step backtrack } T) T U$
 $\wedge \text{skip}^{**} U V \wedge \text{backtrack } V W)$
 $\vee (\exists T U. (\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S)^{**} S T$
 $\wedge (\lambda T U. \text{resolve } T U \wedge \text{no-step backtrack } T) T U \wedge \text{skip}^{**} U W)$
 $\vee (\exists T. \text{skip}^{**} S T \wedge \text{backtrack } T W)$
 $\vee \text{skip}^{**} S W$ (**is** *?RB S W* \vee *?R S W* \vee *?SB S W* \vee *?S S W*)

using *assms*

proof *induction*

case *base*

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

next

case (*step W X*) **note** *st = this(1)* **and** *bj = this(2)* **and** *IH = this(3)[OF this(4)]* **and** *inv = this(4)*

have $\neg ?RB S W$ **and** $\neg ?SB S W$

proof (*clarify, goal-cases*)

case (*1 T U V*)

have *skip-or-resolve** S T*

using *1(1)* **by** (*auto dest!: rtranclp-and-rtranclp-left*)

then show *False*

by (*metis (no-types, lifting) 1(2) 1(4) 1(5) backtrack-no-cdcl_W-bj*
cdcl_W-all-struct-inv-def cdcl_W-all-struct-inv-inv cdcl_W-o.bj local.bj other
resolve rtranclp-cdcl_W-all-struct-inv-inv rtranclp-skip-backtrack-backtrack
rtranclp-skip-or-resolve-rtranclp-cdcl_W step.premis)

next

case *2*

then show *?case* **by** (*meson assms(2) cdcl_W-all-struct-inv-def backtrack-no-cdcl_W-bj*
local.bj rtranclp-skip-backtrack-backtrack)

qed

then have *IH: ?R S W* \vee *?S S W* **using** *IH* **by** *blast*

have *cdcl_W** S W* **by** (*metis cdcl_W-o.bj mono-rtranclp other st*)

then have *inv-W: cdcl_W-all-struct-inv W* **by** (*simp add: rtranclp-cdcl_W-all-struct-inv-inv*
step.premis)

consider

(*BT*) *X'* **where** *backtrack W X'*

| (*skip*) *no-step backtrack W* **and** *skip W X*

| (*resolve*) *no-step backtrack W* **and** *resolve W X*

using *bj cdcl_W-bj.cases* **by** *meson*

then show *?case*

proof *cases*

case (*BT X'*)

then consider

(*bt*) *backtrack W X*

| (*sk*) *skip W X*

using *bj if-can-apply-backtrack-no-more-resolve[of W W X' X] inv-W cdcl_W-bj.cases* **by** *fast*

then show *?thesis*

proof *cases*

case *bt*

then show *?thesis* **using** *IH* **by** *auto*

next

case *sk*

```

    then show ?thesis using IH by (meson rtrancpl-trans r-into-rtrancpl)
  qed
next
  case skip
  then show ?thesis using IH by (meson rtrancpl.rtrancpl-into-rtrancpl)
next
  case resolve note no-bt = this(1) and res = this(2)
  consider
    (RS) T U where
      ( $\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S$ )** S T and
      resolve T U and
      no-step backtrack T and
      skip** U W
    | (S) skip** S W
  using IH by auto
  then show ?thesis
  proof cases
    case (RS T U)
    have cdclW** S T
      using RS(1) cdclW-bj.resolve cdclW-o.bj other skip
      mono-rtrancpl[of ( $\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S$ ) cdclW S T]
      by meson
    then have cdclW-all-struct-inv U
      by (meson RS(2) cdclW-all-struct-inv-inv cdclW-bj.resolve cdclW-o.bj other
          rtrancpl-cdclW-all-struct-inv-inv step.prem)
    { fix U'
      assume skip** U U' and skip** U' W
      have cdclW-all-struct-inv U'
        using  $\langle \text{cdcl}_W\text{-all-struct-inv } U \rangle \langle \text{skip}^{**} U U' \rangle$  rtrancpl-cdclW-all-struct-inv-inv
          cdclW-o.bj rtrancpl-mono[of skip cdclW] other skip by blast
      then have no-step backtrack U'
        using if-can-apply-backtrack-no-more-resolve[OF  $\langle \text{skip}^{**} U' W \rangle$ ] res by blast
    }
    with  $\langle \text{skip}^{**} U W \rangle$ 
    have ( $\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S$ )** U W
    proof induction
      case base
      then show ?case by simp
    next
      case (step V W) note st = this(1) and skip = this(2) and IH = this(3) and H = this(4)
      have  $\bigwedge U'. \text{skip}^{**} U' V \implies \text{skip}^{**} U' W$ 
        using skip by auto
      then have ( $\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S$ )** U V
        using IH H by blast
      moreover have ( $\lambda S T. \text{skip-or-resolve } S T \wedge \text{no-step backtrack } S$ )** V W
        by (simp add: local.skip r-into-rtrancpl st step.prem)
      ultimately show ?case by simp
    qed
  then show ?thesis
  proof -
    have f1:  $\forall p \text{ pa pb pc. } \neg p \text{ (pa) pb} \vee \neg p^{**} \text{ pb pc} \vee p^{**} \text{ pa pc}$ 
      by (meson converse-rtrancpl-into-rtrancpl)
    have skip-or-resolve T U  $\wedge$  no-step backtrack T
      using RS(2) RS(3) by force

```

```

then have (λp pa. skip-or-resolve p pa ∧ no-step backtrack p)** T W
proof -
  have (∃ vr19 vr16 vr17 vr18. vr19 (vr16::'st) vr17 ∧ vr19** vr17 vr18
    ∧ ¬ vr19** vr16 vr18)
    ∨ ¬ (skip-or-resolve T U ∧ no-step backtrack T)
    ∨ ¬ (λuu uua. skip-or-resolve uu uua ∧ no-step backtrack uu)** U W
    ∨ (λuu uua. skip-or-resolve uu uua ∧ no-step backtrack uu)** T W
  by force
then show ?thesis
  by (metis (no-types) ⟨λS T. skip-or-resolve S T ∧ no-step backtrack S⟩** U W⟩
    ⟨skip-or-resolve T U ∧ no-step backtrack T⟩ f1)
qed
then have (λp pa. skip-or-resolve p pa ∧ no-step backtrack p)** S W
  using RS(1) by force
then show ?thesis
  using no-bt res by blast
qed
next
case S
{ fix U'
  assume skip** S U' and skip** U' W
  then have cdclW** S U'
    using mono-rtranclp[of skip cdclW S U'] by (simp add: cdclW-o.bj other skip)
  then have cdclW-all-struct-inv U'
    by (metis (no-types, hide-lams) ⟨cdclW-all-struct-inv S⟩ rtranclp-cdclW-all-struct-inv-inv)
  then have no-step backtrack U'
    using if-can-apply-backtrack-no-more-resolve[OF ⟨skip** U' W⟩] res by blast
}
with S
have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** S W
proof induction
  case base
  then show ?case by simp
next
case (step V W) note st = this(1) and skip = this(2) and IH = this(3) and H = this(4)
  have ∧ U'. skip** U' V ⟹ skip** U' W
    using skip by auto
  then have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** S V
    using IH H by blast
  moreover have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** V W
    by (simp add: local.skip r-into-rtranclp st step.premis)
  ultimately show ?case by simp
qed
then show ?thesis using res no-bt by blast
qed
qed
qed

```

The case distinction is needed, since $T \sim V$ does not imply that $R^{**} T V$.

lemma *cdcl_W-bj-strongly-confluent*:

assumes

*cdcl_W-bj** S V and*

*cdcl_W-bj** S T and*

n-s: no-step cdcl_W-bj V and

```

    inv: cdclW-all-struct-inv S
  shows  $T \sim V \vee \text{cdcl}_W\text{-bj}^{**} T V$ 
  using assms(2)
proof induction
  case base
  then show ?case by (simp add: assms(1))
next
  case (step T U) note st = this(1) and s-o-r = this(2) and IH = this(3)
  have cdclW** S T
    using st mono-rtrancp[of cdclW-bj cdclW] other by blast
  then have lev-T: cdclW-M-level-inv T
    using inv rtrancp-cdclW-consistent-inv[of S T]
    unfolding cdclW-all-struct-inv-def by auto

  consider
    (TV)  $T \sim V$ 
    | (bj-TV)  $\text{cdcl}_W\text{-bj}^{**} T V$ 
  using IH by blast
  then show ?case
  proof cases
    case TV
    have no-step cdclW-bj T
      using ⟨cdclW-M-level-inv T⟩ n-s cdclW-bj-state-eq-compatible[of T - V] TV by auto
    then show ?thesis
      using s-o-r by auto
  next
    case bj-TV
    then obtain U' where
      T-U':  $\text{cdcl}_W\text{-bj} T U'$  and
      cdclW-bj** U' V
      using IH n-s s-o-r by (metis rtrancp-unfold trancpD)
    have cdclW** S T
      by (metis (no-types, hide-lams) bj mono-rtrancp[of cdclW-bj cdclW] other st)
    then have inv-T: cdclW-all-struct-inv T
      by (metis (no-types, hide-lams) inv rtrancp-cdclW-all-struct-inv-inv)

    have lev-U: cdclW-M-level-inv U
      using s-o-r cdclW-consistent-inv lev-T other by blast
    show ?thesis
      using s-o-r
    proof cases
      case backtrack
      then obtain V0 where skip** T V0 and backtrack V0 V
        using IH if-can-apply-backtrack-skip-or-resolve-is-skip[OF backtrack - inv-T]
        cdclW-bj-decomp-resolve-skip-and-bj
        by (meson bj-TV cdclW-bj.backtrack inv-T lev-T n-s
            rtrancp-skip-backtrack-backtrack-end)
      then have cdclW-bj** T V0 and cdclW-bj V0 V
        using rtrancp-mono[of skip cdclW-bj] by blast+
      then show ?thesis
        using ⟨backtrack V0 V⟩ ⟨skip** T V0⟩ backtrack-unique inv-T local.backtrack
        rtrancp-skip-backtrack-backtrack by auto
    next
      case resolve
      then have  $U \sim U'$ 

```

```

    by (meson T-U' cdclW-bj.simps if-can-apply-backtrack-no-more-resolve inv-T
        resolve-skip-deterministic resolve-unique rtrancpl.rtrancpl-refl)
  then show ?thesis
    using ⟨cdclW-bj** U' V⟩ unfolding rtrancpl-unfold
    by (meson T-U' bj cdclW-consistent-inv lev-T other state-eq-ref state-eq-sym
        trancpl-cdclW-bj-state-eq-compatible)
  next
    case skip
    consider
      (sk) skip T U'
    | (bt) backtrack T U'
    using T-U' by (meson cdclW-bj.cases local.skip resolve-skip-deterministic)
  then show ?thesis
    proof cases
      case sk
      then show ?thesis
        using ⟨cdclW-bj** U' V⟩ unfolding rtrancpl-unfold
        by (meson T-U' bj cdclW-all-inv(3) cdclW-all-struct-inv-def inv-T local.skip other
            trancpl-cdclW-bj-state-eq-compatible skip-unique state-eq-ref)
    next
      case bt
      have skip++ T U
        using local.skip by blast
      then show ?thesis
        using bt by (metis ⟨cdclW-bj** U' V⟩ backtrack inv-T trancpl-unfold-begin
            rtrancpl-skip-backtrack-backtrack-end trancpl-into-rtrancpl)
    qed
  qed
qed
qed
qed

```

lemma *cdcl_W-bj-unique-normal-form*:

```

  assumes
    ST: cdclW-bj** S T and SU: cdclW-bj** S U and
    n-s-U: no-step cdclW-bj U and
    n-s-T: no-step cdclW-bj T and
    inv: cdclW-all-struct-inv S
  shows T ~ U
proof -
  have T ~ U ∨ cdclW-bj** T U
    using ST SU cdclW-bj-strongly-confluent inv n-s-U by blast
  then show ?thesis
    by (metis (no-types) n-s-T rtrancpl-unfold state-eq-ref trancpl-unfold-begin)
qed

```

lemma *full-cdcl_W-bj-unique-normal-form*:

```

  assumes full cdclW-bj S T and full cdclW-bj S U and
    inv: cdclW-all-struct-inv S
  shows T ~ U
    using cdclW-bj-unique-normal-form assms unfolding full-def by blast

```

19.4 CDCL FW

inductive *cdcl_W-merge-restart* :: '*st* ⇒ *st* ⇒ bool' **where**
fw-r-propagate: *propagate S S' ⇒ cdcl_W-merge-restart S S' |*

$fw\text{-}r\text{-}conflict: conflict\ S\ T \implies full\ cdcl_W\text{-}bj\ T\ U \implies cdcl_W\text{-}merge\text{-}restart\ S\ U \mid$
 $fw\text{-}r\text{-}decide: decide\ S\ S' \implies cdcl_W\text{-}merge\text{-}restart\ S\ S' \mid$
 $fw\text{-}r\text{-}rf: cdcl_W\text{-}rf\ S\ S' \implies cdcl_W\text{-}merge\text{-}restart\ S\ S'$

lemma $cdcl_W\text{-}merge\text{-}restart\text{-}cdcl_W$:

assumes $cdcl_W\text{-}merge\text{-}restart\ S\ T$

shows $cdcl_W^{**}\ S\ T$

using $assms$

proof *induction*

case $(fw\text{-}r\text{-}conflict\ S\ T\ U)$ **note** $confl = this(1)$ **and** $bj = this(2)$

have $cdcl_W\ S\ T$ **using** $confl$ **by** $(simp\ add: cdcl_W.intros\ r\text{-}into\text{-}rtranclp)$

moreover

have $cdcl_W\text{-}bj^{**}\ T\ U$ **using** bj **unfolding** $full\text{-}def$ **by** $auto$

then have $cdcl_W^{**}\ T\ U$ **by** $(metis\ cdcl_W\text{-}o.bj\ mono\text{-}rtranclp\ other)$

ultimately show $?case$ **by** $auto$

qed $(simp\text{-}all\ add: cdcl_W\text{-}o.intros\ cdcl_W.intros\ r\text{-}into\text{-}rtranclp)$

lemma $cdcl_W\text{-}merge\text{-}restart\text{-}conflicting\text{-}true\text{-}or\text{-}no\text{-}step$:

assumes $cdcl_W\text{-}merge\text{-}restart\ S\ T$

shows $conflicting\ T = C\text{-}True \vee no\text{-}step\ cdcl_W\ T$

using $assms$

proof *induction*

case $(fw\text{-}r\text{-}conflict\ S\ T\ U)$ **note** $confl = this(1)$ **and** $n\text{-}s = this(2)$

{ fix $D\ V$

assume $cdcl_W\ U\ V$ **and** $conflicting\ U = C\text{-}Clause\ D$

then have $False$

using $n\text{-}s$ **unfolding** $full\text{-}def$

by $(induction\ rule: cdcl_W\text{-}all\text{-}rules\text{-}induct)\ (auto\ dest!: cdcl_W\text{-}bj.intros)$

}

then show $?case$ **by** $(cases\ conflicting\ U)\ fastforce+$

qed $(auto\ simp\ add: cdcl_W\text{-}rf.simps)$

inductive $cdcl_W\text{-}merge :: 'st \Rightarrow 'st \Rightarrow bool$ **where**

$fw\text{-}propagate: propagate\ S\ S' \implies cdcl_W\text{-}merge\ S\ S' \mid$

$fw\text{-}conflict: conflict\ S\ T \implies full\ cdcl_W\text{-}bj\ T\ U \implies cdcl_W\text{-}merge\ S\ U \mid$

$fw\text{-}decide: decide\ S\ S' \implies cdcl_W\text{-}merge\ S\ S' \mid$

$fw\text{-}forget: forget\ S\ S' \implies cdcl_W\text{-}merge\ S\ S'$

lemma $cdcl_W\text{-}merge\text{-}cdcl_W\text{-}merge\text{-}restart$:

$cdcl_W\text{-}merge\ S\ T \implies cdcl_W\text{-}merge\text{-}restart\ S\ T$

by $(meson\ cdcl_W\text{-}merge.cases\ cdcl_W\text{-}merge\text{-}restart.simps\ forget)$

lemma $rtranclp\text{-}cdcl_W\text{-}merge\text{-}trancplp\text{-}cdcl_W\text{-}merge\text{-}restart$:

$cdcl_W\text{-}merge^{**}\ S\ T \implies cdcl_W\text{-}merge\text{-}restart^{**}\ S\ T$

using $rtranclp\text{-}mono[of\ cdcl_W\text{-}merge\ cdcl_W\text{-}merge\text{-}restart]\ cdcl_W\text{-}merge\text{-}cdcl_W\text{-}merge\text{-}restart$ **by** $blast$

lemma $cdcl_W\text{-}merge\text{-}rtranclp\text{-}cdcl_W$:

$cdcl_W\text{-}merge\ S\ T \implies cdcl_W^{**}\ S\ T$

using $cdcl_W\text{-}merge\text{-}cdcl_W\text{-}merge\text{-}restart\ cdcl_W\text{-}merge\text{-}restart\text{-}cdcl_W$ **by** $blast$

lemma $rtranclp\text{-}cdcl_W\text{-}merge\text{-}rtranclp\text{-}cdcl_W$:

$cdcl_W\text{-}merge^{**}\ S\ T \implies cdcl_W^{**}\ S\ T$

using $rtranclp\text{-}mono[of\ cdcl_W\text{-}merge\ cdcl_W^{**}]\ cdcl_W\text{-}merge\text{-}rtranclp\text{-}cdcl_W$ **by** $auto$

lemma $cdcl_W\text{-}merge\text{-}is\text{-}cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn$:


```

assumes
  inv: cdclW-all-struct-inv S and
  cdclW:cdclW-merge S T
shows cdclNOT-merged-bj-learn S T
   $\vee$  (no-step cdclW-merge T  $\wedge$  conflicting T  $\neq$  C-True)
using cdclW inv
proof induction
case (fw-propagate S T) note propa = this(1)
then obtain M N U k L C where
  H: state S = (M, N, U, k, C-True) and
  CL: C + {#L#}  $\in$  # clauses S and
  M-C: M  $\models_{as}$  CNot C and
  undef: undefined-lit (trail S) L and
  T: T  $\sim$  cons-trail (Propagated L (C + {#L#})) S
using propa by auto
have propagateNOT S T
apply (rule propagateNOT.propagateNOT[of - C L])
using H CL T undef M-C by (auto simp: state-eqNOT-def state-eq-def clauses-def
  simp del: state-simp)
then show ?case
  using cdclNOT-merged-bj-learn.intros(2) by blast
next
case (fw-decide S T) note dec = this(1) and inv = this(2)
then obtain L where
  undef-L: undefined-lit (trail S) L and
  atm-L: atm-of L  $\in$  atms-of-msu (init-clss S) and
  T: T  $\sim$  cons-trail (Marked L (Suc (backtrack-lvl S)))
  (update-backtrack-lvl (Suc (backtrack-lvl S)) S)
by auto
have decideNOT S T
apply (rule decideNOT.decideNOT)
using undef-L apply simp
using atm-L inv unfolding cdclW-all-struct-inv-def no-strange-atm-def clauses-def apply auto
using T undef-L unfolding state-eq-def state-eqNOT-def by (auto simp: clauses-def)
then show ?case using cdclNOT-merged-bj-learn-decideNOT by blast
next
case (fw-forget S T) note rf = this(1) and inv = this(2)
then obtain M N C U k where
  S: state S = (M, N, {#C#} + U, k, C-True) and
   $\neg$  M  $\models_{asm}$  clauses S and
  C  $\notin$  set (get-all-mark-of-propagated (trail S)) and
  C-init: C  $\notin$  # init-clss S and
  C-le: C  $\in$  # learned-clss S and
  T: T  $\sim$  remove-clss C S
by auto
have init-clss S  $\models_{pm}$  C
using inv C-le unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def
by (meson mem-set-mset-iff true-clss-clss-in-imp-true-clss-clss)
then have S-C: clauses S - replicate-mset (count (clauses S) C) C  $\models_{pm}$  C
using C-init C-le unfolding clauses-def by (simp add: Un-Diff)
moreover have H: init-clss S + (learned-clss S - replicate-mset (count (learned-clss S) C) C)
   $=$  init-clss S + learned-clss S - replicate-mset (count (learned-clss S) C) C
using C-le C-init by (metis clauses-def clauses-remove-clss diff-zero gr0I
  init-clss-remove-clss learned-clss-remove-clss plus-multiset.rep-eq replicate-mset-0
  semiring-normalization-rules(5))

```

```

have forgetNOT S T
  apply (rule forgetNOT.forgetNOT)
    using S-C apply blast
    using S apply simp
  using (C ∈# learned-clss S) apply (simp add: clauses-def)
using T C-le C-init by (auto
  simp: state-eq-def Un-Diff state-eqNOT-def clauses-def ac-simps H
  simp del: state-simp)
then show ?case using cdclNOT-merged-bj-learn-forgetNOT by blast
next
case (fw-conflict S T U) note confl = this(1) and bj = this(2) and inv = this(3)
obtain CS where
  confl-T: conflicting T = C-Clause CS and
  CS: CS ∈# clauses S and
  tr-S-CS: trail S ⊨as CNot CS
  using confl by auto
have cdclW-all-struct-inv T
  using cdclW.simps cdclW-all-struct-inv-inv confl inv by blast
then have cdclW-M-level-inv T
  unfolding cdclW-all-struct-inv-def by auto
then consider
  (no-bt) skip-or-resolve** T U
| (bt) T' where skip-or-resolve** T T' and backtrack T' U
  using bj rtrancp-cdclW-bj-skip-or-resolve-backtrack unfolding full-def by meson
then show ?case
proof cases
case no-bt
  then have conflicting U ≠ C-True
    using confl by (induction rule: rtrancp-induct) auto
  moreover then have no-step cdclW-merge U
    by (auto simp: cdclW-merge.simps)
  ultimately show ?thesis by blast
next
case bt note s-or-r = this(1) and bt = this(2)
have cdclW** T T'
  using s-or-r mono-rtrancp[of skip-or-resolve cdclW] rtrancp-skip-or-resolve-rtrancp-cdclW
  by blast
then have cdclW-M-level-inv T'
  using rtrancp-cdclW-consistent-inv (cdclW-M-level-inv T) by blast
then obtain M1 M2 i D L K where
  confl-T': conflicting T' = C-Clause (D + {#L#}) and
  M1-M2:(Marked K (i+1) # M1, M2) ∈ set (get-all-marked-decomposition (trail T')) and
  get-level L (trail T') = backtrack-lvl T' and
  get-level L (trail T') = get-maximum-level (D+{#L#}) (trail T') and
  get-maximum-level D (trail T') = i and
  undef-L: undefined-lit M1 L and
  U: U ∼ cons-trail (Propagated L (D+{#L#}))
    (reduce-trail-to M1
      (add-learned-cls (D + {#L#})
        (update-backtrack-lvl i
          (update-conflicting C-True T')))))
  using bt by (auto elim: backtrack-levE)
have [simp]: clauses S = clauses T
  using confl by auto
have [simp]: clauses T = clauses T'

```

```

using s-or-r
proof (induction)
  case base
  then show ?case by simp
next
  case (step U V) note st = this(1) and s-o-r = this(2) and IH = this(3)
  have clauses U = clauses V
    using s-o-r by auto
  then show ?case using IH by auto
qed
have inv-T: cdclW-all-struct-inv T
  by (meson cdclW-cp.simps confl inv r-into-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-msu (clauses S)
  using inv-T' confl-T' unfolding cdclW-all-struct-inv-def no-strange-atm-def clauses-def
    by auto
obtain M where tr-T: trail T = M @ trail T'
  using s-or-r by (induction rule: rtrancpl-induct) auto
obtain M' where
  tr-T': trail T' = M' @ Marked K (i+1) # tl (trail U) and
  tr-U: trail U = Propagated L (D + {#L#}) # tl (trail U)
  using U M1-M2 undef-L inv-T' unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
    by fastforce
def M'' ≡ M @ M'
  have tr-T: trail S = M'' @ Marked K (i+1) # tl (trail U)
  using tr-T tr-T' confl unfolding M''-def by auto
have init-clss T' + learned-clss S ⊨pm D + {#L#}
  using inv-T' confl-T' unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def clauses-def
    by simp
have reduce-trail-to (convert-trail-from-NOT (convert-trail-from-W M1)) S =
  reduce-trail-to M1 S
  by (rule reduce-trail-to-length) simp
moreover have trail (reduce-trail-to M1 S) = M1
  apply (rule reduce-trail-to-skip-beginning[of - M @ - @ M2 @ [Marked K (Suc i)]])
  using confl M1-M2 trail T = M @ trail T'
  apply (auto dest!: get-all-marked-decomposition-exists-prepend
    elim!: conflictE)
  by (rule sym) auto
ultimately have [simp]: trail (reduce-trail-toNOT (convert-trail-from-W M1) S) = M1
  using M1-M2 confl by (auto simp add: reduce-trail-toNOT-reduce-trail-convert)
have every-mark-is-a-conflict U
  using inv-U unfolding cdclW-all-struct-inv-def cdclW-conflicting-def by simp
then have tl (trail U) ⊨as CNot D
  by (metis add-diff-cancel-left' append-self-conv2 tr-U union-commute)

```

```

have backjump-l S U
  apply (rule backjump-l[of - - - - L])
    using tr-T apply simp
    using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
    apply (simp add: comp-def)
    using U M1-M2 confl undef-L M1-M2 inv-T' inv unfolding cdclW-all-struct-inv-def
    cdclW-M-level-inv-def apply (auto simp: state-eqNOT-def)[]
    using CS apply simp
    using tr-S-CS apply simp

    using U undef-L M1-M2 inv-T' inv unfolding cdclW-all-struct-inv-def
    cdclW-M-level-inv-def apply auto[]
    using undef-L atm-L apply simp
    using ⟨init-clss T' + learned-clss S ⊨pm D + {#L#}⟩ unfolding clauses-def apply simp
    apply (metis ⟨tl (trail U) ⊨as CNot D⟩ convert-trail-from-W-true-annots)
    using inv-T' inv-U U confl-T' undef-L M1-M2 unfolding cdclW-all-struct-inv-def
    distinct-cdclW-state-def by (simp add: cdclW-M-level-inv-decomp backjump-l-cond-def)
  then show ?thesis using cdclNOT-merged-bj-learn-backjump-l by fast
qed
qed

abbreviation cdclNOT-restart where
cdclNOT-restart ≡ restart-ops.cdclNOT-raw-restart cdclNOT restart

lemma cdclW-merge-restart-is-cdclNOT-merged-bj-learn-restart-no-step:
assumes
  inv: cdclW-all-struct-inv S and
  cdclW:cdclW-merge-restart S T
shows cdclNOT-restart** S T ∨ (no-step cdclW-merge T ∧ conflicting T ≠ C-True)
proof –
consider
  (fw) cdclW-merge S T
  | (fw-r) restart S T
using cdclW by (meson cdclW-merge-restart.simps cdclW-rf.cases fw-conflict fw-decide fw-forget
  fw-propagate)
then show ?thesis
proof cases
  case fw
then have IH: cdclNOT-merged-bj-learn S T ∨ (no-step cdclW-merge T ∧ conflicting T ≠ C-True)
  using inv cdclW-merge-is-cdclNOT-merged-bj-learn by blast
have invS: invNOT S
  using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
have ff2: cdclNOT++ S T → cdclNOT** S T
  by (meson tranclp-into-rtranclp)
have ff3: no-dup (convert-trail-from-W (trail S))
  using invS by (simp add: comp-def)
have cdclNOT ≤ cdclNOT-restart
  by (auto simp: restart-ops.cdclNOT-raw-restart.simps)
then show ?thesis
  using ff3 ff2 IH cdclNOT-merged-bj-learn-is-tranclp-cdclNOT
  rtranclp-mono[of cdclNOT cdclNOT-restart] invS predicate2D by blast
next
  case fw-r
then show ?thesis by (blast intro: restart-ops.cdclNOT-raw-restart.intros)
qed

```

qed

abbreviation $\mu_{FW} :: 'st \Rightarrow nat$ **where**

$\mu_{FW} S \equiv (if\ no_step\ cdcl_W\text{-}merge\ S\ then\ 0\ else\ 1 + \mu_{CDCL}'\text{-}merged\ (set\ mset\ (init\ clss\ S))\ S)$

lemma $cdcl_W\text{-}merge\text{-}\mu_{FW}\text{-}decreasing$:

assumes

$inv: cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ **and**

$fw: cdcl_W\text{-}merge\ S\ T$

shows $\mu_{FW} T < \mu_{FW} S$

proof –

let $?A = init\ clss\ S$

have $atm\text{-}clauses: atm\text{-}of\text{-}msu\ (clauses\ S) \subseteq atm\text{-}of\text{-}msu\ ?A$

using inv **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ $no\text{-}strange\text{-}atm\text{-}def$ $clauses\text{-}def$ **by** $auto$

have $atm\text{-}trail: atm\text{-}of\ ' lits\text{-}of\ (trail\ S) \subseteq atm\text{-}of\text{-}msu\ ?A$

using inv **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ $no\text{-}strange\text{-}atm\text{-}def$ $clauses\text{-}def$ **by** $auto$

have $n\text{-}d: no\text{-}dup\ (trail\ S)$

using inv **unfolding** $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ **by** $(auto\ simp: cdcl_W\text{-}M\text{-}level\text{-}inv\text{-}decomp)$

have $[simp]: \neg no\text{-}step\ cdcl_W\text{-}merge\ S$

using fw **by** $auto$

have $[simp]: init\ clss\ S = init\ clss\ T$

using $cdcl_W\text{-}merge\text{-}restart\text{-}cdcl_W[of\ S\ T]$ inv $rtranclp\text{-}cdcl_W\text{-}init\text{-}clss$

unfolding $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$

by $(meson\ cdcl_W\text{-}merge.simps\ cdcl_W\text{-}merge\text{-}restart.simps\ cdcl_W\text{-}rf.simps\ fw)$

consider

$(merged)\ cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ T$

| $(n\text{-}s)\ no\text{-}step\ cdcl_W\text{-}merge\ T$

using $cdcl_W\text{-}merge\text{-}is\text{-}cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ inv\ fw$ **by** $blast$

then show $?thesis$

proof $cases$

case $merged$

then show $?thesis$

using $cdcl_{NOT}\text{-}decreasing\text{-}measure'[OF\ -\ atm\text{-}clauses]$ $atm\text{-}trail\ n\text{-}d$

by $(auto\ split: split\text{-}if\ simp: comp\text{-}def)$

next

case $n\text{-}s$

then show $?thesis$ **by** $simp$

qed

qed

lemma $wf\text{-}cdcl_W\text{-}merge: wf\ \{(T, S). cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \wedge cdcl_W\text{-}merge\ S\ T\}$

apply $(rule\ wfP\text{-}if\text{-}measure[of\ -\ \mu_{FW}])$

using $cdcl_W\text{-}merge\text{-}\mu_{FW}\text{-}decreasing$ **by** $blast$

lemma $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}trancp\text{-}cdcl_W\text{-}merge\text{-}trancp\text{-}cdcl_W\text{-}merge\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv$:

assumes

$inv: cdcl_W\text{-}all\text{-}struct\text{-}inv\ b$

$cdcl_W\text{-}merge^{++}\ b\ a$

shows $(\lambda S\ T. cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \wedge cdcl_W\text{-}merge\ S\ T)^{++}\ b\ a$

using $assms(2)$

proof $induction$

case $base$

then show $?case$ **using** inv **by** $auto$

next

case $(step\ c\ d)$ **note** $st = this(1)$ **and** $fw = this(2)$ **and** $IH = this(3)$

```

have 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 wf-tranclp-cdclW-merge: wf {(T, S). cdclW-all-struct-inv S ∧ cdclW-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 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
  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 = C-True
  shows (cdclW-merge-restart** S V ∧ conflicting V = C-True)
    ∨ (∃ T U. cdclW-merge-restart** S T ∧ conflicting V ≠ C-True ∧ 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 = C-True
      by auto
    moreover have conflicting V = C-True
      using propagate by auto
    ultimately show ?thesis using IH cdclW-merge-restart.fw-r-propagate[of U V] by auto
  next
    case conflict
    moreover then have conflicting U = C-True
      by auto
    moreover have conflicting V ≠ C-True
      using conflict by auto
    ultimately show ?thesis using IH by auto
  next
    case other
  then show ?thesis

```

```

proof cases
  case decide
    moreover then have conflicting U = C-True
      by auto
    ultimately show ?thesis using IH cdclW-merge-restart.fw-r-decide[of U V] by auto
next
  case bj
    moreover {
      assume skip-or-resolve U V
      have f1: cdclW-bj++ U V
        by (simp add: local.bj tranclp.r-into-trancl)
      obtain T T' :: 'st where
        f2: cdclW-merge-restart** S U
           $\vee$  cdclW-merge-restart** S T  $\wedge$  conflicting U  $\neq$  C-True
           $\wedge$  conflict T T'  $\wedge$  cdclW-bj** T' U
        using IH confl by blast
      then have ?thesis
        proof –
          have conflicting V  $\neq$  C-True  $\wedge$  conflicting U  $\neq$  C-True
            using (skip-or-resolve U V) by auto
          then show ?thesis
            by (metis (no-types) IH f1 rtranclp-trans tranclp-into-rtranclp)
        qed
      }
    moreover {
      assume backtrack U V
      then have conflicting U  $\neq$  C-True by auto
      then obtain T T' where
        cdclW-merge-restart** S T and
        conflicting U  $\neq$  C-True and
        conflict T T' and
        cdclW-bj** T' U
        using IH confl by blast
      have invU: cdclW-M-level-inv U
        using inv rtranclp-cdclW-consistent-inv step.hyps(1) by blast
      then have conflicting V = C-True
        using (backtrack U V) inv by (auto elim: backtrack-levE
          simp: cdclW-M-level-inv-decomp)
      have full cdclW-bj T' V
        apply (rule rtranclp-fullI[of cdclW-bj T' U V])
        using (cdclW-bj** T' U) apply fast
        using (backtrack U V) backtrack-is-full1-cdclW-bj invU unfolding full1-def full-def
        by blast
      then have ?thesis
        using cdclW-merge-restart.fw-r-conflict[of T T' V] (conflict T T')
        (cdclW-merge-restart** S T) (conflicting V = C-True) by auto
      }
    ultimately show ?thesis by (auto simp: cdclW-bj.simps)
  qed
next
  case rf
    moreover then have conflicting U = C-True and conflicting V = C-True
      by (auto simp: cdclW-rf.simps)
    ultimately show ?thesis using IH cdclW-merge-restart.fw-r-rf[of U V] by auto
  qed

```

qed

lemma *no-step-cdcl_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 = C-True 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 *fastforce*

qed

lemma *rtrancpl-cdcl_W-merge-restart-no-step-cdcl_W-bj*:

assumes

*cdcl_W-merge-restart** S T* and

conflicting S = C-True

shows *no-step cdcl_W-bj T*

using *assms*

apply (*induction rule: rtrancpl-induct*)

apply (*fastforce simp: cdcl_W-bj.simps cdcl_W-rf.simps cdcl_W-merge-restart.simps full-def*)

apply (*fastforce simp: cdcl_W-bj.simps cdcl_W-rf.simps cdcl_W-merge-restart.simps full-def*)

done

If *conflicting S* \neq *C-True*, we cannot say anything.

Remark that this theorem does not say anything about well-foundedness: even if you know that one relation is well-founded, it only states that the normal forms are shared.

lemma *conflicting-true-full-cdcl_W-iff-full-cdcl_W-merge*:

assumes *conf1: conflicting S = C-True* 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 *rtrancpl-mono[of cdcl_W-merge-restart cdcl_W**]* *cdcl_W-merge-restart-cdcl_W*

unfolding full-def by auto

have *n-s: no-step cdcl_W-merge-restart V*

using *full unfolding full-def* by auto

have *n-s-bj: no-step cdcl_W-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 = C-True
| (bj) T U where
  cdclW-merge-restart** S T and
  conflicting V  $\neq$  C-True and
  conflict T U and
  cdclW-bj** U V
  using full rtranc-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-cdcl_W-iff-full-cdcl_W-merge:*

shows $\text{full cdcl}_W (\text{init-state } N) V \longleftrightarrow \text{full cdcl}_W\text{-merge-restart } (\text{init-state } N) V$
by (rule conflicting-true-full-cdcl_W-iff-full-cdcl_W-merge) auto

19.5 FW with strategy

19.5.1 The intermediate step

inductive $\text{cdcl}_W\text{-s}' :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**

conflict': $\text{full1 cdcl}_W\text{-cp } S S' \implies \text{cdcl}_W\text{-s}' S S' \mid$

decide': $\text{decide } S S' \implies \text{no-step cdcl}_W\text{-cp } S \implies \text{full cdcl}_W\text{-cp } S' S'' \implies \text{cdcl}_W\text{-s}' S S'' \mid$

bj': $\text{full1 cdcl}_W\text{-bj } S S' \implies \text{no-step cdcl}_W\text{-cp } S \implies \text{full cdcl}_W\text{-cp } S' S'' \implies \text{cdcl}_W\text{-s}' S S''$

inductive-cases $\text{cdcl}_W\text{-s}'E$: $\text{cdcl}_W\text{-s}' S T$

lemma *rtrancp-cdcl_W-bj-full1-cdclp-cdcl_W-stgy:*

$\text{cdcl}_W\text{-bj}^{**} S S' \implies \text{full cdcl}_W\text{-cp } S' S'' \implies \text{cdcl}_W\text{-stgy}^{**} S S''$

```

proof (induction rule: converse-rtrancpl-induct)
  case base
  then show ?case by (metis cdclW-stgy.conflict' full-unfold rtrancpl.simps)
next
  case (step T U) note st = this(2) and bj = this(1) and IH = this(3)[OF this(4)]
  have no-step cdclW-cp T
    using bj by (auto simp add: cdclW-bj.simps)
  consider
    (U) U = S'
  | (U') U' where cdclW-bj U U' and cdclW-bj** U' S'
    using st by (metis converse-rtrancplE)
  then show ?case
  proof cases
    case U
    then show ?thesis
      using (no-step cdclW-cp T) cdclW-o.bj local.bj other' step.prem by (meson r-into-rtrancpl)
  next
    case U' note U' = this(1)
    have no-step cdclW-cp U
      using U' by (fastforce simp: cdclW-cp.simps cdclW-bj.simps)
    then have full cdclW-cp U U
      by (simp add: full-unfold)
    then have cdclW-stgy T U
      using (no-step cdclW-cp T) cdclW-stgy.simps local.bj cdclW-o.bj by meson
    then show ?thesis using IH by auto
  qed
qed

```

```

lemma cdclW-s'-is-rtrancpl-cdclW-stgy:
  cdclW-s' S T  $\implies$  cdclW-stgy** S T
  apply (induction rule: cdclW-s'.induct)
  apply (auto intro: cdclW-stgy.intros)[]
  apply (meson decide other' r-into-rtrancpl)
  by (metis full1-def rtrancpl-cdclW-bj-full1-cdclp-cdclW-stgy trancpl-into-rtrancpl)

```

```

lemma cdclW-cp-cdclW-bj-bissimulation:
  assumes
    full cdclW-cp T U and
    cdclW-bj** T T' and
    cdclW-all-struct-inv T and
    no-step cdclW-bj T'
  shows full cdclW-cp T' U
     $\vee (\exists U' U''. \text{full cdcl}_W\text{-cp } T' U'' \wedge \text{full1 cdcl}_W\text{-bj } U U' \wedge \text{full cdcl}_W\text{-cp } U' U'' \wedge \text{cdcl}_W\text{-s'}^{**} U U'')$ 
  using assms(2,1,3,4)
proof (induction rule: rtrancpl-induct)
  case base
  then show ?case by blast
next
  case (step T' T'') note st = this(1) and bj = this(2) and IH = this(3)[OF this(4,5)] and
    full = this(4) and inv = this(5)
  have cdclW** T T''
    by (metis (no-types, lifting) cdclW-o.bj local.bj mono-rtrancpl[of cdclW-bj cdclW T T''] other
      st rtrancpl.rtrancpl-into-rtrancpl)
  then have inv-T'': cdclW-all-struct-inv T''
    using inv rtrancpl-cdclW-all-struct-inv-inv by blast

```

```

have  $cdcl_W\text{-}bj^{++} \ T \ T''$ 
  using local.bj st by auto
have  $full1 \ cdcl_W\text{-}bj \ T \ T''$ 
  by (metis  $\langle cdcl_W\text{-}bj^{++} \ T \ T'' \rangle \ full1\text{-}def \ step.prem\{3\}$ )
then have  $T = U$ 
proof -
  obtain  $Z$  where  $cdcl_W\text{-}bj \ T \ Z$ 
    by (meson trancpD  $\langle cdcl_W\text{-}bj^{++} \ T \ T'' \rangle$ )
  { assume  $cdcl_W\text{-}cp^{++} \ T \ U$ 
    then obtain  $Z'$  where  $cdcl_W\text{-}cp \ T \ Z'$ 
      by (meson trancpD)
    then have False
      using  $\langle cdcl_W\text{-}bj \ T \ Z \rangle$  by (fastforce simp:  $cdcl_W\text{-}bj.simps \ cdcl_W\text{-}cp.simps$ )
    }
  then show ?thesis
    using full unfolding full-def rtrancp-unfold by blast
qed
obtain  $U''$  where  $full \ cdcl_W\text{-}cp \ T'' \ U''$ 
  using cdcl_W-cp-normalized-element-all-inv inv-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-def)
  then show ?thesis
    using  $f1$  by (metis (no-types)  $\langle T = U \rangle \langle full1 \ cdcl_W\text{-}bj \ T \ T'' \rangle \ bj' \ calculation(1) \ r\text{-}into\text{-}rtrancp$ )
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-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  $full \ cdcl_W\text{-}cp \ T' \ U$ 
     $\vee (\exists U'. \ full1 \ cdcl_W\text{-}bj \ U \ U' \wedge (\forall U''. \ full \ cdcl_W\text{-}cp \ U' \ U'' \longrightarrow full \ cdcl_W\text{-}cp \ T' \ U''$ 
       $\wedge \ cdcl_W\text{-}s^{**} \ U \ U''))$ 
  using assms(2,1,3,4)
proof (induction rule: 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  $cdcl_W^{**} \ T \ T''$ 
    by (metis (no-types, lifting) cdcl_W-o.bj local.bj mono-rtrancp[of cdcl_W-bj cdcl_W T T''] other st
      rtrancp.rtrancp-into-rtrancp)

```

```

then have inv-T'': cdclW-all-struct-inv T''
  using inv rtrancp-cdclW-all-struct-inv-inv by blast
have cdclW-bj++ T T''
  using local.bj st by auto
have full1 cdclW-bj T T''
  by (metis <cdclW-bj++ T T''> full1-def step.premis(3))
then have T = U
proof -
  obtain Z where cdclW-bj T Z
  by (meson trancpD <cdclW-bj++ T T''>)
  { assume cdclW-cp++ T U
    then obtain Z' where cdclW-cp T Z'
    by (meson trancpD)
    then have False
    using <cdclW-bj T Z> by (fastforce simp: cdclW-bj.simps cdclW-cp.simps)
  }
  then show ?thesis
  using full unfolding full-def rtrancp-unfold by blast
qed
{ fix U''
  assume full cdclW-cp T'' U''
  moreover then have cdclW-stgy** U U''
  by (metis <T = U> <cdclW-bj++ T T''> rtrancp-cdclW-bj-full1-cdclp-cdclW-stgy rtrancp-unfold)
  moreover have cdclW-s*** U U''
  proof -
    obtain ss :: 'st ⇒ 'st where
      f1: ∀ x2. (∃ v3. cdclW-cp x2 v3) = cdclW-cp x2 (ss x2)
    by maura
    have ¬ cdclW-cp U (ss U)
    by (meson assms(1) full-def)
    then show ?thesis
    using f1 by (metis (no-types) <T = U> <full1 cdclW-bj T T''> bj' calculation(1)
      r-into-rtrancp)
    qed
  ultimately have full1 cdclW-bj U T'' and cdclW-s*** T'' U''
  using <full1 cdclW-bj T T''> <full cdclW-cp T'' U''> unfolding <T = U>
  apply blast
  by (metis <full cdclW-cp T'' U''> cdclW-s'.simps full-unfold rtrancp.simps)
}
then show ?case
  using <full1 cdclW-bj T T''> full bj' unfolding <T = U> full-def by (metis r-into-rtrancp)
qed

lemma cdclW-stgy-cdclW-s'-connected:
  assumes cdclW-stgy S U and cdclW-all-struct-inv S
  shows cdclW-s' S U
  ∨ (∃ U'. full1 cdclW-bj U U' ∧ (∀ U''. full cdclW-cp U' U'' ⟶ cdclW-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)
              ultimately show ?thesis using cdclW-s'.bj'[of S U] using fbj by blast
            qed
          qed
      qed
  qed

lemma cdclW-stgy-cdclW-s'-connected':
  assumes cdclW-stgy S U and cdclW-all-struct-inv S
  shows cdclW-s' S U
    ∨ (∃ U' U''. cdclW-s' S U'' ∧ full1 cdclW-bj U U' ∧ full cdclW-cp U' U'')
  using assms
proof (induction rule: cdclW-stgy.induct)
  case (conflict' T)
  then have cdclW-s' S T
    using cdclW-s'.conflict' by blast
  then show ?case

```

```

    by blast
next
case (other' T U) note o = this(1) and n-s = this(2) and full = this(3) and inv = this(4)
show ?case
  using o
  proof cases
    case decide
    then show ?thesis using cdclW-s'.simps full n-s by blast
  next
  case bj
  have cdclW-all-struct-inv T
    using cdclW-all-struct-inv-inv o other other'.prems by blast
  then obtain T' where T': full cdclW-bj T T'
    using cdclW-bj-exists-normal-form unfolding full-def cdclW-all-struct-inv-def by metis
  then have full cdclW-bj S T'
    proof -
      have f1: cdclW-bj** T T' ∧ no-step cdclW-bj T'
        by (metis (no-types) T' full-def)
      then have cdclW-bj** S T'
        by (meson converse-rtranclp-into-rtranclp local.bj)
      then show ?thesis
        using f1 by (simp add: full-def)
    qed
  have cdclW-bj** T T'
    using T' unfolding full-def by simp
  have cdclW-all-struct-inv T
    using cdclW-all-struct-inv-inv o other other'.prems by blast
  then consider
    (T'U) full cdclW-cp T' U
  | (U) U' U'' where
    full cdclW-cp T' U'' and
    full1 cdclW-bj U U' and
    full cdclW-cp U' U'' and
    cdclW-s'** U U''
    using cdclW-cp-cdclW-bj-bissimulation[OF full ⟨cdclW-bj** T T'⟩] T' unfolding full-def
    by blast
  then show ?thesis by (metis T' cdclW-s'.simps full-full1 local.bj n-s)
qed
qed

```

lemma *cdcl_W-stgy-cdcl_W-s'-no-step:*
 assumes *cdcl_W-stgy S U* and *cdcl_W-all-struct-inv S* and *no-step cdcl_W-bj U*
 shows *cdcl_W-s' S U*
 using *cdcl_W-stgy-cdcl_W-s'-connected*[OF *assms(1,2)*] *assms(3)*
 by (metis (no-types, lifting) full1-def tranclpD)

lemma *rtranclp-cdcl_W-stgy-connected-to-rtranclp-cdcl_W-s':*
 assumes *cdcl_W-stgy** S U* and *inv: cdcl_W-M-level-inv S*
 shows *cdcl_W-s'** S U ∨ (∃ T. cdcl_W-s'** S T ∧ cdcl_W-bj⁺⁺ T U ∧ conflicting U ≠ C-True)*
 using *assms(1)*
proof *induction*
 case *base*
 then show ?case by *simp*
next
 case (step T V) note *st = this(1)* and *o = this(2)* and *IH = this(3)*

```

from o show ?case
proof cases
  case conflict'
  then have f2:  $cdcl_W-s' T V$ 
    using  $cdcl_W-s'.conflict'$  by blast
  obtain ss :: 'st where
    f3:  $S = T \vee cdcl_W-stgy^{**} S ss \wedge cdcl_W-stgy ss T$ 
    by (metis (full-types) rtranclp.simps st)
  obtain ssa :: 'st where
     $cdcl_W-cp T ssa$ 
    using conflict' by (metis (no-types) full1-def tranclpD)
  then have  $S = T$ 
    using f3 by (metis (no-types)  $cdcl_W-stgy.simps$  full-def full1-def)
  then show ?thesis
    using f2 by blast
next
case (other' U) note o = this(1) and n-s = this(2) and full = this(3)
then show ?thesis
  using o
  proof (cases rule:  $cdcl_W-o-rule-cases$ )
    case decide
    then have  $cdcl_W-s'^{**} S T$ 
      using IH by auto
    then show ?thesis
      by (meson decide decide' full n-s rtranclp.rtrancl-into-rtrancl)
  next
  case backtrack
  consider
    (s')  $cdcl_W-s'^{**} S T$ 
    | (bj)  $S'$  where  $cdcl_W-s'^{**} S S'$  and  $cdcl_W-bj^{++} S' T$  and conflicting  $T \neq C-True$ 
    using IH by blast
  then show ?thesis
    proof cases
      case s'
      moreover
        have  $cdcl_W-M-level-inv T$ 
          using inv local.step(1) rtranclp- $cdcl_W-stgy-consistent-inv$  by auto
        then have full1  $cdcl_W-bj T U$ 
          using backtrack-is-full1- $cdcl_W-bj$  backtrack by blast
        then have  $cdcl_W-s' T V$ 
          using full bj' n-s by blast
        ultimately show ?thesis by auto
    next
    case (bj S') note  $S-S' = this(1)$  and  $bj-T = this(2)$ 
    have no-step  $cdcl_W-cp S'$ 
      using bj-T by (fastforce simp:  $cdcl_W-cp.simps$   $cdcl_W-bj.simps$  dest!: tranclpD)
    moreover
      have  $cdcl_W-M-level-inv T$ 
        using inv local.step(1) rtranclp- $cdcl_W-stgy-consistent-inv$  by auto
      then have full1  $cdcl_W-bj T U$ 
        using backtrack-is-full1- $cdcl_W-bj$  backtrack by blast
      then have full1  $cdcl_W-bj S' U$ 
        using bj-T unfolding full1-def by fastforce
      ultimately have  $cdcl_W-s' S' V$  using full by (simp add: bj')
      then show ?thesis using  $S-S'$  by auto
    end
  end
end

```

```

qed
next
case skip
then have [simp]:  $U = V$ 
  using full converse-rtrancpE unfolding full-def by fastforce

consider
  ( $s'$ )  $cdcl_W-s'^{**} S T$ 
  | ( $bj$ )  $S'$  where  $cdcl_W-s'^{**} S S'$  and  $cdcl_W-bj^{++} S' T$  and conflicting  $T \neq C\text{-True}$ 
  using IH by blast
then show ?thesis
proof cases
case  $s'$ 
have  $cdcl_W-bj^{++} T V$ 
  using skip by force
moreover have conflicting  $V \neq C\text{-True}$ 
  using skip by auto
ultimately show ?thesis using  $s'$  by auto
next
case ( $bj S'$ ) note  $S-S' = \text{this}(1)$  and  $bj-T = \text{this}(2)$ 
have  $cdcl_W-bj^{++} S' V$ 
  using skip  $bj-T$  by (metis  $\langle U = V \rangle cdcl_W-bj.skip \text{trancp.simps}$ )

moreover have conflicting  $V \neq C\text{-True}$ 
  using skip by auto
ultimately show ?thesis using  $S-S'$  by auto
qed
next
case resolve
then have [simp]:  $U = V$ 
  using full converse-rtrancpE unfolding full-def by fastforce
consider
  ( $s'$ )  $cdcl_W-s'^{**} S T$ 
  | ( $bj$ )  $S'$  where  $cdcl_W-s'^{**} S S'$  and  $cdcl_W-bj^{++} S' T$  and conflicting  $T \neq C\text{-True}$ 
  using IH by blast
then show ?thesis
proof cases
case  $s'$ 
have  $cdcl_W-bj^{++} T V$ 
  using resolve by force
moreover have conflicting  $V \neq C\text{-True}$ 
  using resolve by auto
ultimately show ?thesis using  $s'$  by auto
next
case ( $bj S'$ ) note  $S-S' = \text{this}(1)$  and  $bj-T = \text{this}(2)$ 
have  $cdcl_W-bj^{++} S' V$ 
  using resolve  $bj-T$  by (metis  $\langle U = V \rangle cdcl_W-bj.resolve \text{trancp.simps}$ )
moreover have conflicting  $V \neq C\text{-True}$ 
  using resolve by auto
ultimately show ?thesis using  $S-S'$  by auto
qed
qed
qed
qed

```



```

lemma n-step-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 cdclW-bj++ Sa S'a  $\wedge$  no-step cdclW-bj S'a
      using a2 by (simp add: full1-def)
    then have cdclW-bj Sa (ss S'a Sa cdclW-bj)  $\wedge$  cdclW-bj** (ss S'a Sa cdclW-bj) S'a
      using f3 by auto
    then show cdclW++ Sa S''

```

```

    using a1 n-s by (meson bj other rtrancp-cdclW-bj-full1-cdclp-cdclW-stgy
      rtrancp-cdclW-stgy-rtrancp-cdclW rtrancp-into-trancp2)
qed

lemma trancp-cdclW-s'-trancp-cdclW:
  cdclW-s'++ S S'  $\implies$  cdclW++ S S'
  apply (induct rule: trancp.induct)
  using cdclW-s'-trancp-cdclW apply blast
  by (meson cdclW-s'-trancp-cdclW trancp-trans)

lemma rtrancp-cdclW-s'-rtrancp-cdclW:
  cdclW-s'** S S'  $\implies$  cdclW** S S'
  using rtrancp-unfold[of cdclW-s' S S'] trancp-cdclW-s'-trancp-cdclW[of S S'] by auto

lemma full-cdclW-stgy-iff-full-cdclW-s':
  assumes inv: cdclW-all-struct-inv S
  shows full cdclW-stgy S T  $\longleftrightarrow$  full cdclW-s' S T (is ?S  $\longleftrightarrow$  ?S')
proof
  assume ?S'
  then have cdclW** S T
    using rtrancp-cdclW-s'-rtrancp-cdclW[of S T] unfolding full-def by blast
  then have inv': cdclW-all-struct-inv T
    using rtrancp-cdclW-all-struct-inv-inv inv by blast
  have cdclW-stgy** S T
    using ⟨?S'⟩ unfolding full-def
    using cdclW-s'-is-rtrancp-cdclW-stgy rtrancp-mono[of cdclW-s' cdclW-stgy**] by auto
  then show ?S
    using ⟨?S'⟩ inv' cdclW-stgy-cdclW-s'-connected' unfolding full-def by blast
next
  assume ?S
  then have inv-T: cdclW-all-struct-inv T
    by (metis assms full-def rtrancp-cdclW-all-struct-inv-inv rtrancp-cdclW-stgy-rtrancp-cdclW)
  consider
    (s') cdclW-s'** S T
  | (st) S' where cdclW-s'** S S' and cdclW-bj++ S' T and conflicting T  $\neq$  C-True
  using rtrancp-cdclW-stgy-connected-to-rtrancp-cdclW-s'[of S T] inv ⟨?S⟩
  unfolding full-def cdclW-all-struct-inv-def
  by blast
  then show ?S'
  proof cases
    case s'
    then show ?thesis
      by (metis ⟨full cdclW-stgy S T⟩ inv-T cdclW-all-struct-inv-def cdclW-s'.simps
        cdclW-stgy.conflict' cdclW-then-exists-cdclW-stgy-step full-def
        n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o)
  next
    case (st S')
    have full cdclW-cp T T
      using conflicting-clause-full-cdclW-cp st(3) by blast
    moreover
      have n-s: no-step cdclW-bj T
        by (metis ⟨full cdclW-stgy S T⟩ bj inv-T cdclW-all-struct-inv-def
          cdclW-then-exists-cdclW-stgy-step full-def)
    then have full1 cdclW-bj S' T

```

using $st(2)$ unfolding $full1-def$ by $blast$
 moreover have $no-step\ cdcl_W-cp\ S'$
 using $st(2)$ by $(fastforce\ dest!:\ tranclpD\ simp:\ cdcl_W-cp.simps\ cdcl_W-bj.simps)$
 ultimately have $cdcl_W-s'\ S'\ T$
 using $cdcl_W-s'.bj'[of\ S'\ T\ T]$ by $blast$
 then have $cdcl_W-s'^{**}\ S\ T$
 using $st(1)$ by $auto$
 moreover have $no-step\ cdcl_W-s'\ T$
 using $inv-T$ by $(metis\ \langle full\ cdcl_W-cp\ T\ T \rangle\ \langle full\ cdcl_W-stgy\ S\ T \rangle\ cdcl_W-all-struct-inv-def\ cdcl_W-then-exists-cdcl_W-stgy-step\ full-def\ n-step-cdcl_W-stgy-iff-no-step-cdcl_W-cl-cdcl_W-o)$
 ultimately show $?thesis$
 unfolding $full-def$ by $blast$
 qed
 qed

lemma $conflict-step-cdcl_W-stgy-step$:

assumes
 $conflict\ S\ T$
 $cdcl_W-all-struct-inv\ S$
 shows $\exists T. cdcl_W-stgy\ S\ T$
 proof –
 obtain U where $full\ cdcl_W-cp\ S\ U$
 using $cdcl_W-cp-normalized-element-all-inv\ assms$ by $blast$
 then have $full1\ cdcl_W-cp\ S\ U$
 by $(metis\ cdcl_W-cp.conflict'\ assms(1)\ full-unfold)$
 then show $?thesis$ using $cdcl_W-stgy.conflict'$ by $blast$
 qed

lemma $decide-step-cdcl_W-stgy-step$:

assumes
 $decide\ S\ T$
 $cdcl_W-all-struct-inv\ S$
 shows $\exists T. cdcl_W-stgy\ S\ T$
 proof –
 obtain U where $full\ cdcl_W-cp\ T\ U$
 using $cdcl_W-cp-normalized-element-all-inv$ by $(meson\ assms(1)\ assms(2)\ cdcl_W-all-struct-inv-inv\ cdcl_W-cp-normalized-element-all-inv\ decide\ other)$
 then show $?thesis$
 by $(metis\ assms\ cdcl_W-cp-normalized-element-all-inv\ cdcl_W-stgy.conflict'\ decide\ full-unfold\ other')$
 qed

lemma $rtranclp-cdcl_W-cp-conflicting-C-Clause$:

$cdcl_W-cp^{**}\ S\ T \implies conflicting\ S = C-Clause\ D \implies S = T$
 using $rtranclpD\ tranclpD$ by $fastforce$

inductive $cdcl_W-merge-cp :: 'st \Rightarrow 'st \Rightarrow bool$ **where**

$conflict'[intro]: conflict\ S\ T \implies full\ cdcl_W-bj\ T\ U \implies cdcl_W-merge-cp\ S\ U \mid$
 $propagate'[intro]: propagate^{++}\ S\ S' \implies cdcl_W-merge-cp\ S\ S'$

lemma $cdcl_W-merge-restart-cases[consumes\ 1,\ case-names\ conflict\ propagate]$:

assumes
 $cdcl_W-merge-cp\ S\ U$ **and**
 $\bigwedge T. conflict\ S\ T \implies full\ cdcl_W-bj\ T\ U \implies P$ **and**
 $propagate^{++}\ S\ U \implies P$

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 $S\ T \implies cdcl_W\text{-merge}^{++}\ S\ T$
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^{**} $S\ T \implies cdcl_W^{**}\ S\ T$
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 $S\ T \implies no\text{-step}\ cdcl_W\text{-cp}\ S$
by (*metis rtranclp-unfold cdcl_W-cp-conflicting-not-empty conflicting-clause.exhaust full1-def*
rtranclp-cdcl_W-merge-restart-no-step-cdcl_W-bj tranclpD)

inductive *cdcl_W-s'-without-decide* **where**
conflict'-without-decide[*intro*]: *full1 cdcl_W-cp* $S\ S' \implies cdcl_W\text{-s'-without-decide}\ S\ S' \mid$
bj'-without-decide[*intro*]: *full1 cdcl_W-bj* $S\ S' \implies no\text{-step}\ cdcl_W\text{-cp}\ S \implies full\ cdcl_W\text{-cp}\ S'\ S''$
 $\implies cdcl_W\text{-s'-without-decide}\ S\ S''$

lemma *rtranclp-cdcl_W-s'-without-decide-rtranclp-cdcl_W*:
cdcl_W-s'-without-decide^{**} $S\ T \implies cdcl_W^{**}\ S\ T$
apply (*induction rule*: *rtranclp-induct*)
apply *simp*
by (*meson cdcl_W-s'.simps cdcl_W-s'-tranclp-cdcl_W cdcl_W-s'-without-decide.simps*
rtranclp-tranclp-tranclp tranclp-into-rtranclp)

lemma *rtranclp-cdcl_W-s'-without-decide-rtranclp-cdcl_W-s'*:
cdcl_W-s'-without-decide^{**} $S\ T \implies cdcl_W\text{-s}'^{**}\ S\ T$
proof (*induction rule*: *rtranclp-induct*)
case *base*
then show ?*case* **by** *simp*
next
case (*step* $y\ z$) **note** $a2 = this(2)$ **and** $a1 = this(3)$
have *cdcl_W-s'* $y\ z$
 using $a2$ **by** (*metis (no-types) bj' cdcl_W-s'.conflict' cdcl_W-s'-without-decide.cases*)
then show *cdcl_W-s'*^{**} $S\ z$
 using $a1$ **by** (*meson r-into-rtranclp rtranclp-trans*)
qed

lemma *rtranclp-cdcl_W-merge-cp-is-rtranclp-cdcl_W-s'-without-decide*:
assumes
cdcl_W-merge-cp^{**} $S\ V$
conflicting $S = C\text{-True}$
shows
 $(cdcl_W\text{-s'-without-decide}^{**}\ S\ V)$
 $\vee (\exists T. cdcl_W\text{-s'-without-decide}^{**}\ S\ T \wedge propagate^{++}\ T\ V)$
 $\vee (\exists T\ U. cdcl_W\text{-s'-without-decide}^{**}\ S\ T \wedge full1\ cdcl_W\text{-bj}\ T\ U \wedge propagate^{**}\ U\ V)$
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 rtranclp.rtrancl-refl)
 qed
 qed
 qed

lemma *rtranclp-cdcl_W-s'-without-decide-is-rtranclp-cdcl_W-merge-cp*:

assumes

*cdcl_W-s'-without-decide*** *S V* **and**

confl: *conflicting S = C-True*

shows

(*cdcl_W-merge-cp*** *S V* \wedge *conflicting V = C-True*)

\vee (*cdcl_W-merge-cp*** *S V* \wedge *conflicting V* \neq *C-True* \wedge *no-step cdcl_W-cp V* \wedge *no-step cdcl_W-bj V*)

\vee ($\exists T$. *cdcl_W-merge-cp*** *S T* \wedge *conflict T V*)

using *assms*(1)

proof (*induction*)

case *base*

then show ?*case* **using** *confl* **by** *auto*

next

case (*step U V*) **note** *st = this(1)* **and** *s = this(2)* **and** *IH = this(3)*

from *s* **show** ?*case*

proof (*cases rule: cdcl_W-s'-without-decide.cases*)

case *conflict'-without-decide*

then have *rt: cdcl_W-cp*⁺⁺ *U V* **unfolding** *full1-def* **by** *fast*

then have *conflicting U = C-True*

using *trancplp-cdcl_W-cp-propagate-with-conflict-or-not*[*of U V*]

conflict **by** (*auto dest!*: *trancplpD simp: rtranclp-unfold*)

then have *cdcl_W-merge-cp*** *S U* **using** *IH* **by** *auto*

consider

(*propa*) *propagate*⁺⁺ *U V*

| (*confl'*) *conflict U V*

| (*propa-confl'*) *U' where propagate*⁺⁺ *U U' conflict U' V*

using *trancplp-cdcl_W-cp-propagate-with-conflict-or-not*[*OF rt*] **unfolding** *rtranclp-unfold*

by *fastforce*

then show ?*thesis*

proof *cases*

case *propa*

then have *cdcl_W-merge-cp U V*

by *auto*

moreover have *conflicting V = C-True*

using *propa unfolding trancplp-unfold-end* **by** *auto*

ultimately show ?*thesis* **using** \langle *cdcl_W-merge-cp*** *S U* \rangle **by** *force*

next

case *confl'*

then show ?*thesis* **using** \langle *cdcl_W-merge-cp*** *S U* \rangle **by** *auto*

next

case *propa-confl'* **note** *propa = this(1)* **and** *confl' = this(2)*

then have *cdcl_W-merge-cp U U'* **by** *auto*

then have *cdcl_W-merge-cp*** *S U'* **using** \langle *cdcl_W-merge-cp*** *S U* \rangle **by** *auto*

then show ?*thesis* **using** \langle *cdcl_W-merge-cp*** *S U* \rangle *confl'* **by** *auto*

qed

next

case (*bj'-without-decide U'*) **note** *full-bj = this(1)* **and** *cp = this(3)*

then have *conflicting U* \neq *C-True*

using *full-bj unfolding full1-def* **by** (*fastforce dest!*: *trancplpD simp: cdcl_W-bj.simps*)

with IH **obtain** T **where**
 $S-T$: $cdcl_W\text{-merge-cp}^{**} S T$ **and** $T-U$: $conflict T U$
using $full\text{-bj}$ **unfolding** $full1\text{-def}$ **by** ($blast\ dest$: $trancplD$)
then have $cdcl_W\text{-merge-cp} T U'$
using $cdcl_W\text{-merge-cp.conflict'}$ [$of T U U'$] $full\text{-bj}$ **by** ($simp\ add$: $full\text{-unfold}$)
then have $S-U'$: $cdcl_W\text{-merge-cp}^{**} S U'$ **using** $S-T$ **by** $auto$
consider
 $(n-s) U' = V$
 $| (propa) propagate^{++} U' V$
 $| (confl') conflict U' V$
 $| (propa\text{-}confl') U''$ **where** $propagate^{++} U' U'' conflict U'' V$
using $trancpl\text{-}cdcl_W\text{-cp-propagate-with-conflict-or-not cp}$
unfolding $rtrancpl\text{-unfold full-def}$ **by** $metis$
then show $?thesis$
proof cases
case propa
then have $cdcl_W\text{-merge-cp} U' V$ **by** $auto$
moreover have $conflicting V = C\text{-True}$
using $propa$ **unfolding** $trancpl\text{-unfold-end}$ **by** $auto$
ultimately show $?thesis$ **using** $S-U'$ **by** $force$
next
case confl'
then show $?thesis$ **using** $S-U'$ **by** $auto$
next
case propa-confl' **note** $propa = this(1)$ **and** $confl = this(2)$
have $cdcl_W\text{-merge-cp} U' U''$ **using** $propa$ **by** $auto$
then show $?thesis$ **using** $S-U'$ $confl$ **by** ($meson rtrancpl.rtrancpl\text{-into-rtrancpl}$)
next
case n-s
then show $?thesis$
using $S-U'$ **apply** ($cases conflicting V = C\text{-True}$)
using $full\text{-bj}$ **apply** $simp$
by ($metis cp full\text{-def full-unfold full-bj}$)
qed
qed
qed

lemma $no\text{-step-cdcl}_W\text{-s}'\text{-no-ste-cdcl}_W\text{-merge-cp}$:

assumes
 $cdcl_W\text{-all-struct-inv } S$
 $conflicting S = C\text{-True}$
 $no\text{-step } cdcl_W\text{-s}' S$
shows $no\text{-step } cdcl_W\text{-merge-cp } S$
using $assms$ **apply** ($auto simp$: $cdcl_W\text{-s}'.simps cdcl_W\text{-merge-cp}.simps$)
using $conflict\text{-is-full1-cdcl}_W\text{-cp}$ **apply** $blast$
using $cdcl_W\text{-cp-normalized-element-all-inv cdcl}_W\text{-cp.propagate'}$ **by** ($metis cdcl_W\text{-cp.propagate' full-unfold trancplD}$)

The $no\text{-step decide } S$ is needed, since $cdcl_W\text{-merge-cp}$ is $cdcl_W\text{-s}'$ without $decide$.

lemma $conflicting\text{-true-no-step-cdcl}_W\text{-merge-cp-no-step-s}'\text{-without-decide}$:

assumes
 $confl$: $conflicting S = C\text{-True}$ **and**
 inv : $cdcl_W\text{-M-level-inv } S$ **and**
 $n\text{-s}$: $no\text{-step } cdcl_W\text{-merge-cp } S$
shows $no\text{-step } cdcl_W\text{-s}'\text{-without-decide } S$

proof (*rule ccontr*)
assume \neg *no-step cdcl_W-s'-without-decide S*
then obtain *T* **where**
cdcl_W: cdcl_W-s'-without-decide S T
by *auto*
then have *inv-T: cdcl_W-M-level-inv T*
using *rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W[of S T]*
rtrancpl-cdcl_W-consistent-inv inv **by** *blast*
from *cdcl_W* **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' trancpl-cdcl_W-cp-propagate-with-conflict-or-not[of S T]*
unfolding *full1-def* **by** (*metis full1-def local.conflict'-without-decide rtrancpl-unfold*
trancpl-unfold-begin)
moreover
then obtain *T'* **where** *full cdcl_W-bj T T'*
using *cdcl_W-bj-exists-normal-form inv-T* **by** *blast*
ultimately show *False* **using** *cdcl_W-merge-cp.conflict' n-s* **by** *meson*
next
case (*bj'-without-decide S'*)
then show *?thesis*
using *confl unfolding full1-def* **by** (*fastforce simp: cdcl_W-bj.simps dest: trancplD*)
qed
qed

lemma *conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp:*

assumes
inv: cdcl_W-all-struct-inv S **and**
n-s: no-step cdcl_W-s'-without-decide S
shows *no-step cdcl_W-merge-cp S*
proof (*rule ccontr*)
assume \neg *?thesis*
then obtain *T* **where** *cdcl_W-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-cdcl_W-cp* **by** *blast*
next
case *propagate'*
moreover
have *cdcl_W-all-struct-inv T*
using *inv* **by** (*meson local.propagate' rtrancpl-cdcl_W-all-struct-inv-inv*
rtrancpl-propagate-is-rtrancpl-cdcl_W trancpl-into-rtrancpl)
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 *trancpl-full-full1I[of cdcl_W-cp S T U] cdcl_W-cp.propagate'*
trancpl-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 = C-True 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)+

lemma *conflicting-true-full-cdcl_W-merge-cp-iff-full-cdcl_W-s'-without-decode*:
assumes
confl: conflicting S = C-True and
inv: cdcl_W-all-struct-inv S
shows
full cdcl_W-merge-cp S V \longleftrightarrow *full cdcl_W-s'-without-decode S V* (**is** *?fw* \longleftrightarrow *?s'*)

proof
assume *?fw*
then have *st: cdcl_W-merge-cp** S V and n-s: no-step cdcl_W-merge-cp V*
unfolding *full-def* **by** *blast+*
have *inv-V: cdcl_W-all-struct-inv V*
using *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W[of S V] <?fw>* **unfolding** *full-def*
by (*simp add: inv rtranclp-cdcl_W-all-struct-inv-inv*)
consider
*(s') cdcl_W-s'-without-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 tranclp-mono[of propagate cdcl_W-cp] cdcl_W-cp.propagate'* **unfolding** *full1-def*
by *blast*
then have *cdcl_W-s'-without-decode T V*
using *conflict'-without-decode* **by** *blast*
then show *?thesis using s' by auto*
next
case *bj* **note** *s' = this(1) and bj = this(2) and propa = this(3)*
have *no-step cdcl_W-cp V*
using *no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp n-s inv-V*
unfolding *cdcl_W-all-struct-inv-def* **by** *blast*
then have *full cdcl_W-cp U V*
using *propa rtranclp-mono[of propagate cdcl_W-cp] cdcl_W-cp.propagate'* **unfolding** *full-def*
by *blast*

```

moreover have no-step cdclW-cp T
  using bj unfolding full1-def by (fastforce dest!: tranclpD simp:cdclW-bj.simps)
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 = C-True)
  case False
  { fix ss :: 'st
    have ff1: ∀ s sa. ¬ cdclW-s' s sa ∨ full1 cdclW-cp s sa
      ∨ (∃ sb. decide s sb ∧ no-step cdclW-cp s ∧ full cdclW-cp sb sa)
      ∨ (∃ sb. full1 cdclW-bj s sb ∧ no-step cdclW-cp s ∧ full cdclW-cp sb sa)
      by (metis cdclW-s'.cases)
    have ff2: (∀ p s sa. ¬ full1 p (s::'st) sa ∨ p++ s sa ∧ no-step p sa)
      ∧ (∀ p s sa. (¬ p++ (s::'st) sa ∨ (∃ s. p sa s)) ∨ full1 p s sa)
      by (meson full1-def)
    obtain ssa :: ('st ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st ⇒ 'st where
      ff3: ∀ p s sa. ¬ p++ s sa ∨ p s (ssa p s sa) ∧ p** (ssa p s sa) sa
      by (metis (no-types) tranclpD)
    then have a3: ¬ cdclW-cp++ V ss
      using False by (metis conflicting-clause-full-cdclW-cp full-def)
    have ∧s. ¬ cdclW-bj++ V s
      using ff3 False by (metis confl st
        conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj)
    then have ¬ cdclW-s'-without-decide V ss
      using ff1 a3 ff2 by (metis cdclW-s'-without-decide.cases)
  }
then show ?thesis
  by fastforce
next
  case True
  then show ?thesis
    using conflicting-true-no-step-cdclW-merge-cp-no-step-s'-without-decide n-s inv-V
    unfolding cdclW-all-struct-inv-def by blast
  qed
ultimately show ?s' unfolding full-def by blast
next
assume s': ?s'
then have st: cdclW-s'-without-decide** S V and n-s: no-step cdclW-s'-without-decide V
  unfolding full-def by auto
then have cdclW** S V
  using rtranclp-cdclW-s'-without-decide-rtranclp-cdclW st by blast
then have inv-V: cdclW-all-struct-inv V using inv rtranclp-cdclW-all-struct-inv-inv by blast
then have n-s-cp-V: no-step cdclW-cp V
  using cdclW-cp-normalized-element-all-inv[of V] full-fullI[of cdclW-cp V] n-s
  conflict'-without-decide conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp
  no-step-cdclW-merge-cp-no-step-cdclW-cp
  unfolding cdclW-all-struct-inv-def by presburger
have n-s-bj: no-step cdclW-bj V
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain W where W: cdclW-bj V W by blast
  have cdclW-all-struct-inv W
    using W cdclW.simps cdclW-all-struct-inv-inv inv-V by blast

```

then obtain W' **where** $\text{full1 } \text{cdcl}_W\text{-bj } V \ W'$
using $\text{cdcl}_W\text{-bj-exists-normal-form}[of \ W] \ \text{full-fullI}[of \ \text{cdcl}_W\text{-bj } V \ W] \ W$
unfolding $\text{cdcl}_W\text{-all-struct-inv-def}$
by blast
moreover
then have $\text{cdcl}_W^{++} \ V \ W'$
using $\text{trancpl-mono}[of \ \text{cdcl}_W\text{-bj } \text{cdcl}_W] \ \text{cdcl}_W.\text{other } \text{cdcl}_W\text{-o.bj}$ **unfolding** full1-def **by** blast
then have $\text{cdcl}_W\text{-all-struct-inv } W'$
by $(\text{meson } \text{inv-} V \ \text{rtrancpl-cdcl}_W\text{-all-struct-inv-inv } \text{trancpl-into-rtrancpl})$
then obtain X **where** $\text{full } \text{cdcl}_W\text{-cp } W' \ X$
using $\text{cdcl}_W\text{-cp-normalized-element-all-inv}$ **by** blast
ultimately show False
using $\text{bj'-without-decide } n\text{-s-cp-} V \ n\text{-s}$ **by** blast
qed
from s' **consider**
 $(\text{cp-true}) \ \text{cdcl}_W\text{-merge-cp}^{**} \ S \ V$ **and** $\text{conflicting } V = C\text{-True}$
 $| \ (\text{cp-false}) \ \text{cdcl}_W\text{-merge-cp}^{**} \ S \ V$ **and** $\text{conflicting } V \neq C\text{-True}$ **and** $\text{no-step } \text{cdcl}_W\text{-cp } V$ **and**
 $\text{no-step } \text{cdcl}_W\text{-bj } V$
 $| \ (\text{cp-conf}) \ T$ **where** $\text{cdcl}_W\text{-merge-cp}^{**} \ S \ T$ $\text{conflict } T \ V$
using $\text{rtrancpl-cdcl}_W\text{-s'-without-decide-is-rtrancpl-cdcl}_W\text{-merge-cp}[of \ S \ V] \ \text{confl}$
unfolding full-def **by** blast
then have $\text{cdcl}_W\text{-merge-cp}^{**} \ S \ V$
proof cases
case cp-conf **note** $S\text{-}T = \text{this}(1)$ **and** $\text{conf-}V = \text{this}(2)$
have $\text{full } \text{cdcl}_W\text{-bj } V \ V$
using $\text{conf-}V \ n\text{-s-bj}$ **unfolding** full-def **by** fast
then have $\text{cdcl}_W\text{-merge-cp } T \ V$
using $\text{cdcl}_W\text{-merge-cp.conflict' } \text{conf-}V$ **by** auto
then show $?thesis$ **using** $S\text{-}T$ **by** auto
qed fast+
moreover
then have $\text{cdcl}_W^{**} \ S \ V$ **using** $\text{rtrancpl-cdcl}_W\text{-merge-cp-rtrancpl-cdcl}_W$ **by** blast
then have $\text{cdcl}_W\text{-all-struct-inv } V$
using $\text{inv } \text{rtrancpl-cdcl}_W\text{-all-struct-inv-inv}$ **by** blast
then have $\text{no-step } \text{cdcl}_W\text{-merge-cp } V$
using $\text{conflicting-true-no-step-s'-without-decide-no-step-cdcl}_W\text{-merge-cp } s'$
unfolding full-def **by** blast
ultimately show $?fw$ **unfolding** full-def **by** auto
qed

lemma $\text{conflicting-true-full1-cdcl}_W\text{-merge-cp-iff-full1-cdcl}_W\text{-s'-without-decode}$:
assumes
 $\text{confl: } \text{conflicting } S = C\text{-True}$ **and**
 $\text{inv: } \text{cdcl}_W\text{-all-struct-inv } S$
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 $\text{confl } \text{conflicting-true-full-cdcl}_W\text{-merge-cp-iff-full-cdcl}_W\text{-s'-without-decide } \text{inv}$
by blast
then show $?thesis$ **unfolding** $\text{full-unfold full1-def}$
by $(\text{metis } (\text{mono-tags}) \ \text{trancpl-unfold-begin})$
qed

lemma $\text{conflicting-true-full1-cdcl}_W\text{-merge-cp-imp-full1-cdcl}_W\text{-s'-without-decode}$:

```

assumes
  fw: full1 cdclW-merge-cp S V and
  inv: cdclW-all-struct-inv S
shows
  full1 cdclW-s'-without-decide S V
proof –
  have conflicting S = C-True
    using fw unfolding full1-def by (auto dest!: tranclpD simp: cdclW-merge-cp.simps)
  then show ?thesis
    using conflicting-true-full1-cdclW-merge-cp-iff-full1-cdclW-s'-without-decode fw inv by blast
qed

inductive cdclW-merge-stgy where
  fw-s-cp[intro]: full1 cdclW-merge-cp S T  $\implies$  cdclW-merge-stgy S T |
  fw-s-decide[intro]: decide S T  $\implies$  no-step cdclW-merge-cp S  $\implies$  full cdclW-merge-cp T U
     $\implies$  cdclW-merge-stgy S U

lemma cdclW-merge-stgy-tranclp-cdclW-merge:
  assumes fw: cdclW-merge-stgy S T
  shows cdclW-merge++ S T
proof –
  { fix S T
    assume full1 cdclW-merge-cp S T
    then have cdclW-merge++ S T
      using tranclp-mono[of cdclW-merge-cp cdclW-merge++] cdclW-merge-cp-tranclp-cdclW-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: cdclW-merge-stgy** S T
  shows cdclW-merge** S T
  using fw cdclW-merge-stgy-tranclp-cdclW-merge rtranclp-mono[of cdclW-merge-stgy cdclW-merge++]
  unfolding tranclp-rtranclp-rtranclp by blast

lemma cdclW-merge-stgy-rtranclp-cdclW:
  cdclW-merge-stgy S T  $\implies$  cdclW** S T
  apply (induction rule: cdclW-merge-stgy.induct)
    using rtranclp-cdclW-merge-cp-rtranclp-cdclW unfolding full1-def
    apply (simp add: tranclp-into-rtranclp)
  using rtranclp-cdclW-merge-cp-rtranclp-cdclW cdclW-o.decide cdclW-other unfolding full-def
  by (meson r-into-rtranclp rtranclp-trans)

lemma rtranclp-cdclW-merge-stgy-rtranclp-cdclW:
  cdclW-merge-stgy** S T  $\implies$  cdclW** S T
  using rtranclp-mono[of cdclW-merge-stgy cdclW**] cdclW-merge-stgy-rtranclp-cdclW by auto

lemma cdclW-merge-stgy-cases[consumes 1, case-names fw-s-cp fw-s-decide]:
  assumes

```

$cdcl_W\text{-merge-stgy } S \ U$
 $full1 \ cdcl_W\text{-merge-cp } S \ U \implies P$
 $\bigwedge T. \text{ decide } S \ T \implies \text{no-step } cdcl_W\text{-merge-cp } S \implies full \ cdcl_W\text{-merge-cp } T \ U \implies P$
shows P
using *assms* **by** (*auto simp: cdcl_W-merge-stgy.simps*)

inductive $cdcl_W\text{-s'-w} :: 'st \Rightarrow 'st \Rightarrow bool$ **where**
 $\text{conflict': } full1 \ cdcl_W\text{-s'-without-decide } S \ S' \implies cdcl_W\text{-s'-w } S \ S' \mid$
 $\text{decide': } \text{decide } S \ S' \implies \text{no-step } cdcl_W\text{-s'-without-decide } S \implies full \ cdcl_W\text{-s'-without-decide } S' \ S''$
 $\implies cdcl_W\text{-s'-w } S \ S''$

lemma $cdcl_W\text{-s'-w-rtrancp-cdcl}_W$:
 $cdcl_W\text{-s'-w } S \ T \implies cdcl_W^{**} \ S \ T$
apply (*induction rule: cdcl_W-s'-w.induct*)
using $rtrancp\text{-cdcl}_W\text{-s'-without-decide-rtrancp-cdcl}_W$ **unfolding** $full1\text{-def}$
apply (*simp add: trancp-into-rtrancp*)
using $rtrancp\text{-cdcl}_W\text{-s'-without-decide-rtrancp-cdcl}_W$ **unfolding** $full\text{-def}$
by (*meson decide other rtrancp-into-trancp2 trancp-into-rtrancp*)

lemma $rtrancp\text{-cdcl}_W\text{-s'-w-rtrancp-cdcl}_W$:
 $cdcl_W\text{-s'-w}^{**} \ S \ T \implies cdcl_W^{**} \ S \ T$
using $rtrancp\text{-mono}[of \ cdcl_W\text{-s'-w } cdcl_W^{**}] \ cdcl_W\text{-s'-w-rtrancp-cdcl}_W$ **by** *auto*

lemma $\text{no-step-cdcl}_W\text{-cp-no-step-cdcl}_W\text{-s'-without-decide}$:
assumes $\text{no-step } cdcl_W\text{-cp } S$ **and** $\text{conflicting } S = C\text{-True}$ **and** $\text{inv: } cdcl_W\text{-M-level-inv } S$
shows $\text{no-step } cdcl_W\text{-s'-without-decide } S$
by (*metis assms cdcl_W-cp.conflict' cdcl_W-cp.propagate' cdcl_W-merge-restart-cases trancpD*
 $\text{conflicting-true-no-step-cdcl}_W\text{-merge-cp-no-step-s'-without-decide}$)

lemma $\text{no-step-cdcl}_W\text{-cp-no-step-cdcl}_W\text{-merge-restart}$:
assumes $\text{no-step } cdcl_W\text{-cp } S$ **and** $\text{conflicting } S = C\text{-True}$
shows $\text{no-step } cdcl_W\text{-merge-cp } S$
by (*metis assms(1) cdcl_W-cp.conflict' cdcl_W-cp.propagate' cdcl_W-merge-restart-cases trancpD*)

lemma $\text{after-cdcl}_W\text{-s'-without-decide-no-step-cdcl}_W\text{-cp}$:
assumes $cdcl_W\text{-s'-without-decide } S \ T$
shows $\text{no-step } cdcl_W\text{-cp } T$
using *assms* **by** (*induction rule: cdcl_W-s'-without-decide.induct*) (*auto simp: full1-def full-def*)

lemma $\text{no-step-cdcl}_W\text{-s'-without-decide-no-step-cdcl}_W\text{-cp}$:
 $cdcl_W\text{-all-struct-inv } S \implies \text{no-step } cdcl_W\text{-s'-without-decide } S \implies \text{no-step } cdcl_W\text{-cp } S$
by (*simp add: conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp*
 $\text{no-step-cdcl}_W\text{-merge-cp-no-step-cdcl}_W\text{-cp } cdcl_W\text{-all-struct-inv-def}$)

lemma $\text{after-cdcl}_W\text{-s'-w-no-step-cdcl}_W\text{-cp}$:
assumes $cdcl_W\text{-s'-w } S \ T$ **and** $cdcl_W\text{-all-struct-inv } S$
shows $\text{no-step } cdcl_W\text{-cp } T$
using *assms*
proof (*induction rule: cdcl_W-s'-w.induct*)
case conflict'
then show $?case$
by (*auto simp: full1-def trancp-unfold-end after-cdcl_W-s'-without-decide-no-step-cdcl_W-cp*)

next
case ($\text{decide' } S \ T \ U$)
moreover
then have $cdcl_W^{**} \ S \ U$

using *rtrancpl-cdcl_W-s'-without-decide-rtrancpl-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 rtrancpl-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 *rtrancpl-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: rtrancpl-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step T U*)
moreover have *cdcl_W-all-struct-inv T*
using *rtrancpl-cdcl_W-s'-w-rtrancpl-cdcl_W[of S U] assms(2) rtrancpl-cdcl_W-all-struct-inv-inv*
rtrancpl-cdcl_W-s'-w-rtrancpl-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 *rtrancpl-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: rtrancpl-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step T U*)
moreover have *cdcl_W-all-struct-inv T*
using *rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W[of S U] assms(2) rtrancpl-cdcl_W-all-struct-inv-inv*
rtrancpl-cdcl_W-s'-w-rtrancpl-cdcl_W step.hyps(1)
by (*meson rtrancpl-cdcl_W-merge-stgy-rtrancpl-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 rtrancpl-cdcl_W-all-struct-inv-inv
rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W trancpl.intros(1) trancpl-into-rtrancpl)
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 $rtrancpl\text{-}mono[of\ cdcl_W\text{-}bj\ cdcl_W]\ cdcl_W.other\ cdcl_W\text{-}o.bj\ trancpl\text{-}into\text{-}rtrancpl[of\ cdcl_W\text{-}bj]$
unfolding $full1\text{-}def$ **by** $(metis\ (full\text{-}types)\ predicate2D\ predicate2I)$
then have $cdcl_W\text{-}all\text{-}struct\text{-}inv\ T'$
using $inv\ rtrancpl\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ **by** $blast$
then obtain U **where** $full\ cdcl_W\text{-}cp\ T'\ U$
using $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv$ **by** $blast$
moreover have $no\text{-}step\ cdcl_W\text{-}cp\ S$
using $S\text{-}T$ **by** $(auto\ simp:\ cdcl_W\text{-}bj.simps)$
ultimately show $False$
using $assms\ cdcl_W\text{-}s'\text{-}without\text{-}decide.intros(2)[of\ S\ T'\ U]$ **by** $fast$
qed

lemma $cdcl_W\text{-}s'\text{-}w\text{-}no\text{-}step\text{-}cdcl_W\text{-}bj$:
assumes $cdcl_W\text{-}s'\text{-}w\ S\ T$ **and** $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $no\text{-}step\ cdcl_W\text{-}bj\ T$
using $assms$ **apply** $induction$
using $rtrancpl\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}rtrancpl\text{-}cdcl_W\ rtrancpl\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$
 $no\text{-}step\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}no\text{-}step\text{-}cdcl_W\text{-}bj$ **unfolding** $full1\text{-}def$
apply $(meson\ trancpl\text{-}into\text{-}rtrancpl)$
using $rtrancpl\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}rtrancpl\text{-}cdcl_W\ rtrancpl\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$
 $no\text{-}step\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}no\text{-}step\text{-}cdcl_W\text{-}bj$ **unfolding** $full\text{-}def$
by $(meson\ cdcl_W\text{-}merge\text{-}restart\text{-}cdcl_W\ fw\text{-}r\text{-}decide)$

lemma $rtrancpl\text{-}cdcl_W\text{-}s'\text{-}w\text{-}no\text{-}step\text{-}cdcl_W\text{-}bj\text{-}or\text{-}eq$:
assumes $cdcl_W\text{-}s'\text{-}w^{**} S\ T$ **and** $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $S = T \vee no\text{-}step\ cdcl_W\text{-}bj\ T$
using $assms$ **apply** $induction$
apply $simp$
using $rtrancpl\text{-}cdcl_W\text{-}s'\text{-}w\text{-}rtrancpl\text{-}cdcl_W\ rtrancpl\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$
 $cdcl_W\text{-}s'\text{-}w\text{-}no\text{-}step\text{-}cdcl_W\text{-}bj$ **by** $meson$

lemma $rtrancpl\text{-}cdcl_W\text{-}s'\text{-}no\text{-}step\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}decomp\text{-}into\text{-}cdcl_W\text{-}merge$:
assumes
 $cdcl_W\text{-}s'^{**} R\ V$ **and**
 $conflicting\ R = C\text{-}True$ **and**
 $inv:\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ R$
shows $(cdcl_W\text{-}merge\text{-}stgy^{**} R\ V \wedge conflicting\ V = C\text{-}True)$
 $\vee (cdcl_W\text{-}merge\text{-}stgy^{**} R\ V \wedge conflicting\ V \neq C\text{-}True \wedge no\text{-}step\ cdcl_W\text{-}bj\ V)$
 $\vee (\exists S\ T\ U. cdcl_W\text{-}merge\text{-}stgy^{**} R\ S \wedge no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S \wedge decide\ S\ T$
 $\wedge cdcl_W\text{-}merge\text{-}cp^{**} T\ U \wedge conflict\ U\ V)$
 $\vee (\exists S\ T. cdcl_W\text{-}merge\text{-}stgy^{**} R\ S \wedge no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S \wedge decide\ S\ T$
 $\wedge cdcl_W\text{-}merge\text{-}cp^{**} T\ V$
 $\wedge conflicting\ V = C\text{-}True)$
 $\vee (cdcl_W\text{-}merge\text{-}cp^{**} R\ V \wedge conflicting\ V = C\text{-}True)$
 $\vee (\exists U. cdcl_W\text{-}merge\text{-}cp^{**} R\ U \wedge conflict\ U\ V)$
using $assms(1,2)$

proof $induction$

case $base$

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

next

case $(step\ V\ W)$ **note** $st = this(1)$ **and** $s' = this(2)$ **and** $IH = this(3)[OF\ this(4)]$ **and**
 $n\text{-}s\text{-}R = this(4)$

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 = C-True
  | (cp) cdclW-merge-cp** R V
  | (cp-conf) U where cdclW-merge-cp** R U and conflict U V
  using IH by meson
then show ?thesis
proof cases
next
  case s'
  then have R = V
  by (metis full1-def inv local.conflict' tranclp-unfold-begin
    rtranclp-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq)
  consider
    (V-W) V = W
  | (propa) propagate++ V W and conflicting W = C-True
  | (propa-conf) V' where propagate** V V' and conflict V' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V W] conflict'
  unfolding full-unfold full1-def by blast
then show ?thesis
proof cases
  case V-W
  then show ?thesis using ⟨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)
  then show ?thesis by fast
next
  case dec note T-V = this(4)
  consider
    (propa) propagate++ V W and conflicting W = C-True
  | (propa-conf) V' where propagate** V V' and conflict V' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V W] conflict'
  unfolding full1-def by blast
then show ?thesis
proof cases
  case propa
  then show ?thesis
  by (meson T-V cdclW-merge-cp.propagate' dec rtranclp.rtrancl-into-rtrancl)

```



```

next
  case propa-conf
  then have cdclW-merge-cp** T V'
    using T-V by (metis rtrancp-unfold cdclW-merge-cp.propagate' rtrancp.simps)
  then show ?thesis using dec propa-conf(2) by metis
qed
next
case cp
consider
  (propa) propagate++ V W and conflicting W = C-True
  | (propa-conf) V' where propagate** V V' and conflict V' W
  using trancp-cdclW-cp-propagate-with-conflict-or-not[of V W] conflict'
  unfolding full1-def by blast
then show ?thesis
proof cases
  case propa
  then show ?thesis by (meson cdclW-merge-cp.propagate' cp rtrancp.rtrancp-into-rtrancp)
next
  case propa-conf
  then show ?thesis
    using propa-conf(2) by (metis rtrancp-unfold cdclW-merge-cp.propagate'
      cp rtrancp.rtrancp-into-rtrancp)
qed
next
case cp-conf
then show ?thesis using conflict' unfolding full1-def by (fastforce dest!: trancpD)
qed
next
case (decide' V')
then have conf-V: conflicting V = C-True
  by auto
consider
  (s') cdclW-merge-stgy** R V
  | (dec-conf) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T U and conflict U V
  | (dec) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and decide S T
    and cdclW-merge-cp** T V and conflicting V = C-True
  | (cp) cdclW-merge-cp** R V
  | (cp-conf) U where cdclW-merge-cp** R U and conflict U V
  using IH by meson
then show ?thesis
proof cases
  case s'
  have conf-V': conflicting V' = C-True using decide'(1) by auto
  have full: full1 cdclW-cp V' W ∨ (V' = W ∧ no-step cdclW-cp W)
    using decide'(3) unfolding full-unfold by blast
  consider
    (V'-W) V' = W
    | (propa) propagate++ V' W and conflicting W = C-True
    | (propa-conf) V'' where propagate** V' V'' and conflict V'' W
  using trancp-cdclW-cp-propagate-with-conflict-or-not[of V W] decide'
  by (metis ⟨full1 cdclW-cp V' W ∨ V' = W ∧ no-step cdclW-cp W⟩ full1-def
    trancp-cdclW-cp-propagate-with-conflict-or-not)
then show ?thesis
proof cases

```

```

    case V'-W
  then show ?thesis
    using conf1-V' local.decide'(1,2) s' conf-V
    no-step-cdclW-cp-no-step-cdclW-merge-restart by auto
next
  case propa
  then show ?thesis using local.decide'(1,2) s' by (metis cdclW-merge-cp.simps conf-V
    no-step-cdclW-cp-no-step-cdclW-merge-restart r-into-rtranclp)
next
  case propa-conf1
  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-conf1(2) s' conf-V
    no-step-cdclW-cp-no-step-cdclW-merge-restart
    by metis
qed
next
  case (dec) note s' = this(1) and dec = this(2) and cp = this(3) and ns-cp-T = this(4)
  have full cdclW-merge-cp T V
    unfolding full-def by (simp add: conf-V local.decide'(2)
      no-step-cdclW-cp-no-step-cdclW-merge-restart ns-cp-T)
  moreover have no-step cdclW-merge-cp V
    by (simp add: conf-V local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart)
  moreover have no-step cdclW-merge-cp S
    by (metis dec)
  ultimately have cdclW-merge-stgy S V
    using cp by blast
  then have cdclW-merge-stgy** R V using s' by auto
  consider
    (V'-W) V' = W
    | (propa) propagate++ V' W and conflicting W = C-True
    | (propa-conf1) V'' where propagate** V' V'' and conflict V'' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
  unfolding full-unfold full1-def by blast
  then show ?thesis
  proof cases
    case V'-W
    moreover have conflicting V' = C-True
      using decide'(1) by auto
    ultimately show ?thesis
      using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩ by blast
  next
    case propa
    moreover then have cdclW-merge-cp V' W
      by auto
    ultimately show ?thesis
      using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩
      by (meson r-into-rtranclp)
  next
    case propa-conf1
    moreover then have cdclW-merge-cp** V' V''
      by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
    ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
      ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)

```

```

qed
next
case cp
have no-step cdclW-merge-cp V
  using conf-V local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart by blast
then have full cdclW-merge-cp R V
  unfolding full-def using cp by fast
then have cdclW-merge-stgy** R V
  unfolding full-unfold by auto
have full1 cdclW-cp V' W  $\vee$  (V' = W  $\wedge$  no-step cdclW-cp W)
  using decide'(3) unfolding full-unfold by blast

consider
  (V'-W) V' = W
| (propa) propagate++ V' W and conflicting W = C-True
| (propa-confl) V'' where propagate** V' V'' and conflict V'' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
  unfolding full-unfold full1-def by blast
then show ?thesis

proof cases
case V'-W
moreover have conflicting V' = C-True
  using decide'(1) by auto
ultimately show ?thesis
  using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩ by blast
next
case propa
moreover then have cdclW-merge-cp V' W
  by auto
ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
  ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)
next
case propa-confl
moreover then have cdclW-merge-cp** V' V''
  by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
  ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)
qed
next
case (dec-confl)
show ?thesis using conf-V dec-confl(5) by auto
next
case cp-confl
then show ?thesis using decide' by fastforce
qed
next
case (bj' V')
then have  $\neg$ no-step cdclW-bj V
  by (auto dest: tranclpD simp: full1-def)
then consider
  (s') cdclW-merge-stgy** R V and conflicting V = C-True
| (dec-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  $cdcl_W\text{-merge-cp}^{**} T V$  and  $conflicting V = C\text{-True}$ 
|  $(cp) cdcl_W\text{-merge-cp}^{**} R V$  and  $conflicting V = C\text{-True}$ 
|  $(cp\text{-confl}) U$  where  $cdcl_W\text{-merge-cp}^{**} R U$  and  $conflict U V$ 
using  $IH$  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:  $cdcl_W\text{-bj.simps}$ )
  then show ?thesis by fast
next
  case dec note - =  $this(5)$ 
  then have False
    using  $bj'(1)$  unfolding full1-def by (force dest!: tranclpD simp:  $cdcl_W\text{-bj.simps}$ )
  then show ?thesis by fast
next
  case dec-confl
  then have  $cdcl_W\text{-merge-cp} U V'$ 
    using  $bj' cdcl_W\text{-merge-cp.intros}(1)[of U V V']$  by (simp add: full-unfold)
  then have  $cdcl_W\text{-merge-cp}^{**} T V'$ 
    using dec-confl(4) by simp
  consider
    ( $V' - W$ )  $V' = W$ 
  |  $(propa) propagate^{++} V' W$  and  $conflicting W = C\text{-True}$ 
  |  $(propa\text{-confl}) V''$  where  $propagate^{**} V' V''$  and  $conflict V'' W$ 
  using tranclp- $cdcl_W\text{-cp-propagate-with-conflict-or-not}[of V' W]$   $bj'(3)$ 
  unfolding full-unfold full1-def by blast
  then show ?thesis
  proof cases
    case  $V' - W$ 
    then have no-step  $cdcl_W\text{-cp} V'$ 
      using  $bj'(3)$  unfolding full-def by auto
    then have no-step  $cdcl_W\text{-merge-cp} V'$ 
      by (metis  $cdcl_W\text{-cp.propagate}' cdcl_W\text{-merge-cp.cases tranclpD}$ 
        no-step- $cdcl_W\text{-cp-no-conflict-no-propagate}(1)$ )
    then have full1  $cdcl_W\text{-merge-cp} T V'$ 
      unfolding full1-def using  $\langle cdcl_W\text{-merge-cp} U V' \rangle$  dec-confl(4) by auto
    then have full  $cdcl_W\text{-merge-cp} T V'$ 
      by (simp add: full-unfold)
    then have  $cdcl_W\text{-merge-stgy} S V'$ 
      using dec-confl(3)  $cdcl_W\text{-merge-stgy.fw-s-decide}$  (no-step  $cdcl_W\text{-merge-cp} S$ ) by blast
    then have  $cdcl_W\text{-merge-stgy}^{**} R V'$ 
      using  $\langle cdcl_W\text{-merge-stgy}^{**} R S \rangle$  by auto
  show ?thesis
  proof cases
    assume  $conflicting W = C\text{-True}$ 
    then show ?thesis using  $\langle cdcl_W\text{-merge-stgy}^{**} R V' \rangle \langle V' = W \rangle$  by auto
  next
    assume  $conflicting W \neq C\text{-True}$ 
    then show ?thesis
      using  $\langle cdcl_W\text{-merge-stgy}^{**} R V' \rangle \langle V' = W \rangle$  by (metis  $\langle cdcl_W\text{-merge-cp} U V' \rangle$ 
        conflicting-not-true-rtranclp- $cdcl_W\text{-merge-cp-no-step-cdcl_W-bj}$  dec-confl(5)
        r-into-rtranclp conflictE)
  qed
next

```

```

    case propa
    moreover then have cdclW-merge-cp V' W
    by auto
    ultimately show ?thesis using decide' by (meson ⟨cdclW-merge-cp** T V'⟩ dec-confl(1-3)
    rtranclp.rtrancl-into-rtrancl)
  next
    case propa-confl
    moreover then have cdclW-merge-cp** V' V''
    by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
    ultimately show ?thesis by (meson ⟨cdclW-merge-cp** T V'⟩ dec-confl(1-3) rtranclp-trans)
  qed
next
  case cp note - = this(2)
  then show ?thesis using bj'(1) ⟨¬ no-step cdclW-bj V⟩
    conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj by auto
next
  case cp-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 = C-True
  | (propa-confl) V'' where propagate** V' V'' and conflict V'' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] bj'
  unfolding full-unfold full1-def by blast
  then show ?thesis

proof cases
  case V'-W
  show ?thesis
  proof cases
    assume conflicting V' = C-True
    then show ?thesis
    using V'-W ⟨cdclW-merge-cp U V'⟩ cp-confl(1) by force
  next
    assume confl: conflicting V' ≠ C-True
    then have no-step cdclW-merge-stgy V'
    by (auto simp: cdclW-merge-stgy.simps full1-def full-def cdclW-merge-cp.simps
    dest!: tranclpD)
    have no-step cdclW-merge-cp V'
    using confl by (auto simp: full1-def full-def cdclW-merge-cp.simps
    dest!: tranclpD)
    moreover have cdclW-merge-cp U W
    using V'-W ⟨cdclW-merge-cp U V'⟩ by blast
    ultimately have full1 cdclW-merge-cp R V'
    using cp-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 ⟨no-step cdclW-merge-cp V'⟩ by (auto simp: cdclW-merge-stgy.simps
    full1-def dest!: tranclpD)
    ultimately have cdclW-merge-stgy** R V' by auto
    show ?thesis by (metis V'-W ⟨cdclW-merge-cp U V'⟩ ⟨cdclW-merge-stgy** R V'⟩
    conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj cp-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  $\langle \text{cdcl}_W\text{-merge-cp } U \ V' \rangle \text{ cp-conf}(1)$  by force
    next
      case propa-conf
      moreover then have cdclW-merge-cp**  $V' V''$ 
        by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
      ultimately show ?thesis
        using  $\langle \text{cdcl}_W\text{-merge-cp } U \ V' \rangle \text{ cp-conf}(1)$  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
    cdclW-s'**  $T \ U$  and
    n-s-S: no-step cdclW-cp  $S$  and
    no-step cdclW-cp  $U$ 
  shows cdclW-s'**  $S \ U$ 
  using assms(2,4)

```

proof *induction*

```

  case (step  $U \ V$ ) note  $st = \text{this}(1)$  and  $s' = \text{this}(2)$  and  $IH = \text{this}(3)$  and  $n-s = \text{this}(4)$ 

```

consider

```

  ( $TU$ )  $T = U$ 
  | ( $s'-st$ )  $T'$  where cdclW-s'  $T \ T'$  and cdclW-s'**  $T' \ U$ 
  using st[unfolded rtranclp-unfold] by (auto dest!: tranclpD)

```

then show ?*case*

proof *cases*

case TU

then show ?*thesis*

proof –

assume $a1: T = U$

then have $f2: \text{cdcl}_W\text{-s}' \ T \ V$

using s' by force

obtain $ss :: 'st$ where

$\text{cdcl}_W\text{-s^{'**}} \ S \ T \vee \text{cdcl}_W\text{-cp} \ T \ ss$

using $a1 \text{ step.IH}$ by blast

then show ?*thesis*

using $f2$ by (metis (*full-types*) *cdcl_W-s'.decide'* *cdcl_W-s'E dec full1-is-full n-s-S*
rtranclp-unfold tranclp-unfold-end)

qed

next

case ($s'-st \ T'$) note $s'-T' = \text{this}(1)$ and $st = \text{this}(2)$

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 cdclW-s' S T
      using dec cdclW-s'.decide' n-s-S by (simp add: full-unfold)
    then show ?thesis using decide' s'-T' by auto
  next
    case bj'
    then have False
      using dec unfolding full1-def by (fastforce dest!: tranclpD simp: cdclW-bj.simps)
    then show ?thesis by fast
  qed
  then show ?thesis using s' st by auto
qed
next
case base
then have full cdclW-cp T T
  by (simp add: full-unfold)
then show ?case
  using cdclW-s'.simps dec n-s-S by auto
qed

lemma rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s':
  assumes
    cdclW-merge-stgy** R V and
    inv: cdclW-all-struct-inv R
  shows cdclW-s'** R V
  using assms(1)
proof induction
  case base
  then show ?case by simp
next
case (step S T) note st = this(1) and fw = this(2) and IH = this(3)
have cdclW-all-struct-inv S
  using inv rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW st by blast
from fw show ?case
proof (cases rule: cdclW-merge-stgy-cases)
  case fw-s-cp
  then show ?thesis
  proof -
    assume a1: full1 cdclW-merge-cp S T
    obtain ss :: ('st ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st where
      f2:  $\bigwedge p\ s\ sa\ pa\ sb\ sc\ sd\ pb\ se\ sf. (\neg full1\ p\ (s::'st)\ sa \vee p^{++}\ s\ sa) \wedge (\neg pa\ (sb::'st)\ sc \vee \neg full1\ pa\ sd\ sb) \wedge (\neg pb^{++}\ se\ sf \vee pb\ sf\ (ss\ pb\ sf) \vee full1\ pb\ se\ sf)$ 
      by (metis (no-types) full1-def)
    then have f3: cdclW-merge-cp++ S T
      using a1 by auto
    obtain ssa :: ('st ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st ⇒ 'st where
      f4:  $\bigwedge p\ s\ sa. \neg p^{++}\ s\ sa \vee p\ s\ (ssa\ p\ s\ sa)$ 
      by (meson tranclp-unfold-begin)
    then have f5:  $\bigwedge s. \neg full1\ cdcl_W\text{-merge-cp}\ s\ S$ 
      using f3 f2 by (metis (full-types))
    have  $\bigwedge s. \neg full\ cdcl_W\text{-merge-cp}\ s\ S$ 

```

```

    using f4 f3 by (meson full-def)
  then have S = R
    using f5 by (metis (no-types) cdclW-merge-stgy.simps rtrancp-unfold st
      trancp-unfold-end)
  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' = C-True
  by auto
ultimately have full cdclW-s'-without-decide S' T
  by (meson ⟨cdclW-all-struct-inv S⟩ cdclW-merge-restart-cdclW fw-r-decide
    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)
then show ?thesis using IH by auto
qed
qed

```

lemma *rtrancp-cdcl_W-merge-stgy-distinct-mset-clauses:*

```

  assumes invR: cdclW-all-struct-inv R and
    st: cdclW-merge-stgy** R S and
    dist: distinct-mset (clauses R) and
    R: trail R = []
  shows distinct-mset (clauses S)
  using rtrancp-cdclW-stgy-distinct-mset-clauses[OF invR - dist R]
  invR st rtrancp-mono[of cdclW-s' cdclW-stgy**] cdclW-s'-is-rtrancp-cdclW-stgy
  by (auto dest!: cdclW-s'-is-rtrancp-cdclW-stgy rtrancp-cdclW-merge-stgy-rtrancp-cdclW-s')

```

lemma *no-step-cdcl_W-s'-no-step-cdcl_W-merge-stgy:*

```

  assumes
    inv: cdclW-all-struct-inv R and s': no-step cdclW-s' R
  shows no-step cdclW-merge-stgy R

```

proof –

```

{ fix ss :: 'st
  obtain ssa :: 'st ⇒ 'st ⇒ 'st where
    ff1: ⋀ s sa. ¬ cdclW-merge-stgy s sa ∨ full1 cdclW-merge-cp s sa ∨ decide s (ssa s sa)
    using cdclW-merge-stgy.cases by maura
  obtain ssb :: ('st ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st ⇒ 'st where
    ff2: ⋀ p s sa. ¬ p++ s sa ∨ p s (ssb p s sa)
    by (meson trancp-unfold-begin)
  obtain ssc :: 'st ⇒ 'st where
    ff3: ⋀ s sa sb. (¬ cdclW-all-struct-inv s ∨ ¬ cdclW-cp s sa ∨ cdclW-s' s (ssc s))
      ∧ (¬ cdclW-all-struct-inv s ∨ ¬ cdclW-o s sb ∨ cdclW-s' s (ssc s))
    using n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o by maura

```



```

then have ff4:  $\bigwedge s. \neg \text{cdcl}_W\text{-o } R \ s$ 
  using  $s' \text{ inv}$  by blast
have ff5:  $\bigwedge s. \neg \text{cdcl}_W\text{-cp}^{++} \ R \ s$ 
  using ff3 ff2  $s'$  by (metis inv)
have  $\bigwedge s. \neg \text{cdcl}_W\text{-bj}^{++} \ R \ s$ 
  using ff4 ff2 by (metis bj)
then have  $\bigwedge s. \neg \text{cdcl}_W\text{-s'-without-decide } R \ s$ 
  using ff5 by (simp add:  $\text{cdcl}_W\text{-s'-without-decide.simps full1-def}$ )
then have  $\neg \text{cdcl}_W\text{-s'-without-decide}^{++} \ R \ ss$ 
  using ff2 by blast
then have  $\neg \text{cdcl}_W\text{-merge-stgy } R \ ss$ 
  using ff4 ff1 by (metis (full-types) decide full1-def inv
    conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode) }
then show ?thesis
  by fastforce
qed

```

```

lemma wf-cdclW-merge-cp:
  wf{(T, S).  $\text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \ T$ }
  using wf-tranclp-cdclW-merge by (rule wf-subset) (auto simp:  $\text{cdcl}_W\text{-merge-cp-tranclp-cdcl}_W\text{-merge}$ )

```

```

lemma wf-cdclW-merge-stgy:
  wf{(T, S).  $\text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-stgy } S \ T$ }
  using wf-tranclp-cdclW-merge by (rule wf-subset)
  (auto simp add:  $\text{cdcl}_W\text{-merge-stgy-tranclp-cdcl}_W\text{-merge}$ )

```

```

lemma cdclW-merge-cp-obtain-normal-form:
  assumes inv:  $\text{cdcl}_W\text{-all-struct-inv } R$ 
  obtains S where full  $\text{cdcl}_W\text{-merge-cp } R \ S$ 
proof -
  obtain S where full  $(\lambda S \ T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \ T) \ R \ S$ 
    using wf-exists-normal-form-full[OF wf-cdclW-merge-cp] by blast
  then have
    st:  $(\lambda S \ T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \ T)^{**} \ R \ S$  and
    n-s: no-step  $(\lambda S \ T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \ T) \ S$ 
  unfolding full-def by blast+
  have  $\text{cdcl}_W\text{-merge-cp}^{**} \ R \ S$ 
    using st by induction auto
  moreover
    have  $\text{cdcl}_W\text{-all-struct-inv } S$ 
      using st inv
      apply (induction rule: rtranclp-induct)
      apply simp
      by (meson r-into-rtranclp rtranclp-cdclW-all-struct-inv-inv
        rtranclp-cdclW-merge-cp-rtranclp-cdclW)
    then have no-step  $\text{cdcl}_W\text{-merge-cp } S$ 
      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:  $\text{cdcl}_W\text{-all-struct-inv } R$  and
    confl:  $\text{conflicting } R = \text{C-True}$  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 blast
qed
qed

```

```

lemma rtranclp-cdclW-merge-cp-no-step-cdclW-bj:
  assumes conflicting R = C-True and cdclW-merge-cp** R S
  shows no-step cdclW-bj S
  using assms conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj by blast

```

```

lemma rtranclp-cdclW-merge-stgy-no-step-cdclW-bj:
  assumes confl: conflicting R = C-True and cdclW-merge-stgy** R S
  shows no-step cdclW-bj S
  using assms(2)

```

```

proof induction
  case base
  then show ?case
    using confl by (auto simp: cdclW-bj.simps)[]
next
case (step S T) note st = this(1) and fw = this(2) and IH = this(3)
have confl-S: conflicting S = C-True
  using fw apply cases
  by (auto simp: full1-def cdclW-merge-cp.simps dest!: tranclpD)
from fw show ?case
  proof cases
    case fw-s-cp
    then show ?thesis
      using rtranclp-cdclW-merge-cp-no-step-cdclW-bj confl-S
      by (simp add: full1-def tranclp-into-rtranclp)
  next

```

```

    case (fw-s-decide S')
    moreover then have conflicting S' = C-True by auto
    ultimately show ?thesis
      using conflicting-not-true-rtrancp-cdclW-merge-cp-no-step-cdclW-bj
      unfolding full-def by fast
qed
qed

lemma full-cdclW-s'-full-cdclW-merge-restart:
  assumes
    conflicting R = C-True and
    inv: cdclW-all-struct-inv R
  shows full cdclW-s' R V  $\longleftrightarrow$  full cdclW-merge-stgy R V (is ?s'  $\longleftrightarrow$  ?fw)
proof
  assume ?s'
  then have cdclW-s'^** R V unfolding full-def by blast
  have cdclW-all-struct-inv V
    using ⟨cdclW-s'^** R V⟩ inv rtrancp-cdclW-all-struct-inv-inv rtrancp-cdclW-s'-rtrancp-cdclW
    by blast
  then have n-s: no-step cdclW-merge-stgy V
    using no-step-cdclW-s'-no-step-cdclW-merge-stgy by (meson ⟨full cdclW-s' R V⟩ full-def)
  have n-s-bj: no-step cdclW-bj V
    by (metis ⟨cdclW-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ bj full-def
        n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o)
  have n-s-cp: no-step cdclW-merge-cp V
  proof -
    { fix ss :: 'st
      obtain ssa :: 'st  $\Rightarrow$  'st where
        ff1:  $\forall s. \neg \text{cdcl}_W\text{-all-struct-inv } s \vee \text{cdcl}_W\text{-s'-without-decide } s \text{ (ssa } s) \vee \text{no-step cdcl}_W\text{-merge-cp } s$ 
        using conflicting-true-no-step-s'-without-decide-no-step-cdclW-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) ⟨cdclW-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ cdclW-s'.sims
            cdclW-s'-without-decide.cases) }
      then show ?thesis
        by blast
    }
  qed
  consider
    (fw-no-confl) cdclW-merge-stgy** R V and conflicting V = C-True
  | (fw-confl) cdclW-merge-stgy** R V and conflicting V  $\neq$  C-True and no-step cdclW-bj V
  | (fw-dec-confl) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T U and conflict U V
  | (fw-dec-no-confl) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T V and conflicting V = C-True
  | (cp-no-confl) cdclW-merge-cp** R V and conflicting V = C-True
  | (cp-confl) U where cdclW-merge-cp** R U and conflict U V
  using rtrancp-cdclW-s'-no-step-cdclW-s'-without-decide-decomp-into-cdclW-merge[OF
    ⟨cdclW-s'^** R V⟩ assms] by auto
  then show ?fw
  proof cases
    case fw-no-confl
    then show ?thesis using n-s unfolding full-def by blast
  end
end

```

```

next
  case fw-confl
  then show ?thesis using n-s unfolding full-def by blast
next
  case fw-dec-confl
  have cdclW-merge-cp U V
    using n-s-bj by (metis cdclW-merge-cp.simps full-unfold fw-dec-confl(5))
  then have full1 cdclW-merge-cp T V
    unfolding full1-def by (metis fw-dec-confl(4) n-s-cp tranclp-unfold-end)
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case fw-dec-no-confl
  then have full cdclW-merge-cp T V
    using n-s-cp unfolding full-def by blast
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case cp-no-confl
  then have full cdclW-merge-cp R V
    by (simp add: full-def n-s-cp)
  then have R = V ∨ cdclW-merge-stgy++ R V
    by (metis (no-types) full-unfold fw-s-cp rtranclp-unfold tranclp-unfold-end)
  then show ?thesis
    by (simp add: full-def n-s rtranclp-unfold)
next
  case cp-confl
  have full cdclW-bj V V
    using n-s-bj unfolding full-def by blast
  then have full1 cdclW-merge-cp R V
    unfolding full1-def by (meson cdclW-merge-cp.conflict' cp-confl(1,2) n-s-cp
      rtranclp-into-tranclp1)
  then show ?thesis using n-s unfolding full-def by auto
qed
next
assume ?fw
then have cdclW** R V using rtranclp-mono[of cdclW-merge-stgy cdclW**]
  cdclW-merge-stgy-rtranclp-cdclW unfolding full-def by auto
then have inv': cdclW-all-struct-inv V using inv rtranclp-cdclW-all-struct-inv-inv by blast
have cdclW-s'** R V
  using ⟨?fw⟩ by (simp add: full-def inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s')
moreover have no-step cdclW-s' V
proof cases
  assume conflicting V = C-True
  then show ?thesis
    by (metis inv' ⟨full cdclW-merge-stgy R V⟩ full-def
      no-step-cdclW-merge-stgy-no-step-cdclW-s')
next
  assume confl-V: conflicting V ≠ C-True
  then have no-step cdclW-bj V
    using rtranclp-cdclW-merge-stgy-no-step-cdclW-bj by (meson ⟨full cdclW-merge-stgy R V⟩
      assms(1) full-def)
  then show ?thesis using confl-V by (fastforce simp: cdclW-s'.simps full1-def cdclW-cp.simps
    dest!: tranclpD)
qed

```

ultimately show ?s' unfolding full-def by blast
qed

lemma full-cdcl_W-stgy-full-cdcl_W-merge:

assumes

conflicting $R = C\text{-True}$ and

inv: cdcl_W-all-struct-inv R

shows full cdcl_W-stgy $R \ V \longleftrightarrow$ full cdcl_W-merge-stgy $R \ V$

by (simp add: assms(1) full-cdcl_W-s'-full-cdcl_W-merge-restart full-cdcl_W-stgy-iff-full-cdcl_W-s'
inv)

lemma full-cdcl_W-merge-stgy-final-state-conclusive':

fixes $S' :: 'st$

assumes full: full cdcl_W-merge-stgy (init-state N) S'

and no-d: distinct-mset-mset N

shows (conflicting $S' = C\text{-Clause } \{\#\} \wedge$ unsatisfiable (set-mset N))

\vee (conflicting $S' = C\text{-True} \wedge$ trail $S' \models_{asm} N \wedge$ satisfiable (set-mset N))

proof –

have cdcl_W-all-struct-inv (init-state N)

using no-d unfolding cdcl_W-all-struct-inv-def by auto

moreover have conflicting (init-state N) = $C\text{-True}$

by auto

ultimately show ?thesis

by (simp add: full full-cdcl_W-stgy-final-state-conclusive-from-init-state
full-cdcl_W-stgy-full-cdcl_W-merge no-d)

qed

end

19.6 Adding Restarts

locale cdcl_W-ops-restart =

cdcl_W-ops trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail

add-init-cls

add-learned-cls remove-cls update-backtrack-lvl update-conflicting init-state

restart-state

for

trail :: 'st \Rightarrow ('v::linorder, nat, 'v clause) marked-lits and

init-clss :: 'st \Rightarrow 'v clauses and

learned-clss :: 'st \Rightarrow 'v clauses and

backtrack-lvl :: 'st \Rightarrow nat and

conflicting :: 'st \Rightarrow 'v clause conflicting-clause and

cons-trail :: ('v, nat, 'v clause) marked-lit \Rightarrow 'st \Rightarrow 'st and

tl-trail :: 'st \Rightarrow 'st and

add-init-cls :: 'v clause \Rightarrow 'st \Rightarrow 'st and

add-learned-cls remove-cls :: 'v clause \Rightarrow 'st \Rightarrow 'st and

update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st and

update-conflicting :: 'v clause conflicting-clause \Rightarrow 'st \Rightarrow 'st and

init-state :: 'v::linorder clauses \Rightarrow 'st and

restart-state :: 'st \Rightarrow 'st +

fixes $f :: nat \Rightarrow nat$

assumes f : unbounded f

begin

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

inductive *cdcl_W-merge-with-restart* **where**

restart-step:

(*cdcl_W-merge-stgy* $\widehat{\sim}$ (*card* (*set-mset* (*learned-clss* *T*)) - *card* (*set-mset* (*learned-clss* *S*)))) *S T*
 \implies *card* (*set-mset* (*learned-clss* *T*)) - *card* (*set-mset* (*learned-clss* *S*)) > *f n*
 \implies *restart T U* \implies *cdcl_W-merge-with-restart* (*S*, *n*) (*U*, *Suc n*) |

restart-full: *full1 cdcl_W-merge-stgy S T* \implies *cdcl_W-merge-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma *cdcl_W-merge-with-restart S T* \implies *cdcl_W-merge-restart*** (*fst S*) (*fst T*)

by (*induction rule*: *cdcl_W-merge-with-restart.induct*)

(*auto dest!*: *relpowp-imp-rtranclp cdcl_W-merge-stgy-tranclp-cdcl_W-merge tranclp-into-rtranclp*
rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-merge rtranclp-cdcl_W-merge-tranclp-cdcl_W-merge-restart
fw-r-rf cdcl_W-rf.restart
simp: *full1-def*)

lemma *cdcl_W-merge-with-restart-rtranclp-cdcl_W*:

cdcl_W-merge-with-restart S T \implies *cdcl_W*** (*fst S*) (*fst T*)

by (*induction rule*: *cdcl_W-merge-with-restart.induct*)

(*auto dest!*: *relpowp-imp-rtranclp rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W cdcl_W.rf*
cdcl_W-rf.restart tranclp-into-rtranclp simp: *full1-def*)

lemma *cdcl_W-merge-with-restart-increasing-number*:

cdcl_W-merge-with-restart S T \implies *snd T* = 1 + *snd S*

by (*induction rule*: *cdcl_W-merge-with-restart.induct*) *auto*

lemma *full1 cdcl_W-merge-stgy S T* \implies *cdcl_W-merge-with-restart* (*S*, *n*) (*T*, *Suc n*)

using *restart-full* **by** *blast*

lemma *cdcl_W-all-struct-inv-learned-clss-bound*:

assumes *inv*: *cdcl_W-all-struct-inv S*

shows *set-mset* (*learned-clss S*) \subseteq *build-all-simple-clss* (*atms-of-msu* (*init-clss S*))

proof

fix *C*

assume *C*: *C* \in *set-mset* (*learned-clss S*)

have *distinct-mset C*

using *C inv unfolding cdcl_W-all-struct-inv-def distinct-cdcl_W-state-def distinct-mset-set-def*

by *auto*

moreover **have** \neg *tautology C*

using *C inv unfolding cdcl_W-all-struct-inv-def cdcl_W-learned-clause-def* **by** *auto*

moreover

have *atms-of C* \subseteq *atms-of-msu* (*learned-clss S*)

using *C* **by** *auto*

then **have** *atms-of C* \subseteq *atms-of-msu* (*init-clss S*)

using *inv unfolding cdcl_W-all-struct-inv-def no-strange-atm-def* **by** *force*

moreover **have** *finite* (*atms-of-msu* (*init-clss S*))

using *inv unfolding cdcl_W-all-struct-inv-def* **by** *auto*

ultimately **show** *C* \in *build-all-simple-clss* (*atms-of-msu* (*init-clss S*))

using *distinct-mset-not-tautology-implies-in-build-all-simple-clss build-all-simple-clss-mono*

by *blast*

qed

lemma *cdcl_W-merge-with-restart-init-clss*:

cdcl_W-merge-with-restart S T \implies *cdcl_W-M-level-inv* (*fst S*) \implies

init-clss (*fst S*) = *init-clss* (*fst T*)
using *cdcl_W-merge-with-restart-rtrancpl-cdcl_W rtrancpl-cdcl_W-init-clss* **by** *blast*

lemma

wf {(*T*, *S*). *cdcl_W-all-struct-inv* (*fst S*) \wedge *cdcl_W-merge-with-restart* *S T*}

proof (*rule ccontr*)

assume \neg *?thesis*

then obtain *g* **where**

g: $\bigwedge i$. *cdcl_W-merge-with-restart* (*g i*) (*g* (*Suc i*)) **and**

inv: $\bigwedge i$. *cdcl_W-all-struct-inv* (*fst* (*g i*))

unfolding *wf-iff-no-infinite-down-chain* **by** *fast*

{ **fix** *i*

have *init-clss* (*fst* (*g i*)) = *init-clss* (*fst* (*g 0*))

apply (*induction i*)

apply *simp*

using *g inv unfolding cdcl_W-all-struct-inv-def* **by** (*metis cdcl_W-merge-with-restart-init-clss*)

} **note** *init-g* = *this*

let *?S* = *g 0*

have *finite* (*atms-of-msu* (*init-clss* (*fst ?S*)))

using *inv unfolding cdcl_W-all-struct-inv-def* **by** *auto*

have *snd-g*: $\bigwedge i$. *snd* (*g i*) = *i* + *snd* (*g 0*)

apply (*induct-tac i*)

apply *simp*

by (*metis Suc-eq-plus1-left add-Suc cdcl_W-merge-with-restart-increasing-number g*)

then have *snd-g-0*: $\bigwedge i$. *i* > 0 \implies *snd* (*g i*) = *i* + *snd* (*g 0*)

by *blast*

have *unbounded-f-g*: *unbounded* (λi . *f* (*snd* (*g i*)))

using *f unfolding bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g*

not-bounded-nat-exists-larger not-le ordered-cancel-comm-monoid-diff-class.le-iff-add)

obtain *k* **where**

f-g-k: *f* (*snd* (*g k*)) > *card* (*build-all-simple-clss* (*atms-of-msu* (*init-clss* (*fst ?S*)))) **and**

k > *card* (*build-all-simple-clss* (*atms-of-msu* (*init-clss* (*fst ?S*))))

using *not-bounded-nat-exists-larger[OF unbounded-f-g]* **by** *blast*

The following does not hold anymore with the non-strict version of cardinality in the definition.

{ **fix** *i*

assume *no-step cdcl_W-merge-stgy* (*fst* (*g i*))

with *g[of i]*

have *False*

proof (*induction rule: cdcl_W-merge-with-restart.induct*)

case (*restart-step T S n*) **note** *H* = *this*(1) **and** *c* = *this*(2) **and** *n-s* = *this*(4)

obtain *S'* **where** *cdcl_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: trancplD*)

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**

cdcl_W-merge-stgy: (*cdcl_W-merge-stgy* \rightsquigarrow *m*) (*fst* (*g k*)) *T*

```

  using  $g[of\ k]\ H[of\ Suc\ k]$  by (force simp:  $cdcl_W$ -merge-with-restart.simps full1-def)
have  $cdcl_W$ -merge-stgy** (fst (g k)) T
  using  $cdcl_W$ -merge-stgy relpowp-imp-rtrancp by metis
then have  $cdcl_W$ -all-struct-inv T
  using inv[of k] rtrancp- $cdcl_W$ -all-struct-inv-inv rtrancp- $cdcl_W$ -merge-stgy-rtrancp- $cdcl_W$ 
  by blast
moreover have card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k))))
  > card (build-all-simple-clss (atms-of-msu (init-clss (fst ?S))))
  unfolding m[symmetric] using  $\langle m > f\ (snd\ (g\ k))\ \rangle\ f-g-k$  by linarith
then have card (set-mset (learned-clss T))
  > card (build-all-simple-clss (atms-of-msu (init-clss (fst ?S))))
  by linarith
moreover
  have init-clss (fst (g k)) = init-clss T
    using  $\langle cdcl_W$ -merge-stgy** (fst (g k)) T  $\rangle$  rtrancp- $cdcl_W$ -merge-stgy-rtrancp- $cdcl_W$ 
    rtrancp- $cdcl_W$ -init-clss inv unfolding  $cdcl_W$ -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  $cdcl_W$ -all-struct-inv-learned-clss-bound by (metis Suc-leI card-mono not-less-eq-eq
    build-all-simple-clss-finite)
qed

```

lemma $cdcl_W$ -merge-with-restart-distinct-mset-clauses:

```

  assumes invR:  $cdcl_W$ -all-struct-inv (fst R) and
  st:  $cdcl_W$ -merge-with-restart R S and
  dist: distinct-mset (clauses (fst R)) and
  R: trail (fst R) = []
  shows distinct-mset (clauses (fst S))
  using assms(2,1,3,4)
proof (induction)
  case (restart-full S T)
  then show ?case using rtrancp- $cdcl_W$ -merge-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: trancp-into-rtrancp)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T)
    using rtrancp- $cdcl_W$ -merge-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: relpowp-imp-rtrancp)
  then show ?case using  $\langle restart\ T\ U \rangle$  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:
  ( $cdcl_W$ -stgy  $\sim$  (card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)))) S T  $\implies$ 
  card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)) > f n  $\implies$ 
  restart T U  $\implies$ 
   $cdcl_W$ -with-restart (S, n) (U, Suc n) |
restart-full: full1  $cdcl_W$ -stgy S T  $\implies cdcl_W$ -with-restart (S, n) (T, Suc n)

```

lemma $cdcl_W$ -with-restart-rtrancp- $cdcl_W$:

```

 $cdcl_W$ -with-restart S T  $\implies cdcl_W$ ** (fst S) (fst T)
apply (induction rule:  $cdcl_W$ -with-restart.induct)
by (auto dest!: relpowp-imp-rtrancp trancp-into-rtrancp fw-r-rf)

```


cdcl_W-rf.restart rtrancp-cdcl_W-stgy-rtrancp-cdcl_W cdcl_W-merge-restart-cdcl_W
simp: full1-def)

lemma *cdcl_W-with-restart-increasing-number:*
cdcl_W-with-restart S T \implies snd T = 1 + snd S
by (*induction rule: cdcl_W-with-restart.induct*) *auto*

lemma *full1 cdcl_W-stgy S T \implies cdcl_W-with-restart (S, n) (T, Suc n)*
using *restart-full* **by** *blast*

lemma *cdcl_W-with-restart-init-clss:*
cdcl_W-with-restart S T \implies cdcl_W-M-level-inv (fst S) \implies init-clss (fst S) = init-clss (fst T)
using *cdcl_W-with-restart-rtrancp-cdcl_W rtrancp-cdcl_W-init-clss* **by** *blast*

lemma

wf {(T, S). cdcl_W-all-struct-inv (fst S) \wedge cdcl_W-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 *init-clss (fst (g i)) = init-clss (fst (g 0))*

apply (*induction i*)

apply *simp*

using *g inv unfolding cdcl_W-all-struct-inv-def* **by** (*metis cdcl_W-with-restart-init-clss*)

} note *init-g = this*

let *?S = g 0*

have *finite (atms-of-msu (init-clss (fst ?S)))*

using *inv unfolding cdcl_W-all-struct-inv-def* **by** *auto*

have *snd-g: $\bigwedge i. \text{snd } (g\ i) = i + \text{snd } (g\ 0)$*

apply (*induct-tac i*)

apply *simp*

by (*metis Suc-eq-plus1-left add-Suc cdcl_W-with-restart-increasing-number g*)

then have *snd-g-0: $\bigwedge i. i > 0 \implies \text{snd } (g\ i) = i + \text{snd } (g\ 0)$*

by *blast*

have *unbounded-f-g: unbounded ($\lambda i. f\ (\text{snd } (g\ i))$)*

using *f unfolding bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g*

not-bounded-nat-exists-larger not-le ordered-cancel-comm-monoid-diff-class.le-iff-add)

obtain k where

f-g-k: $f\ (\text{snd } (g\ k)) > \text{card } (\text{build-all-simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$ and

k > card (build-all-simple-clss (atms-of-msu (init-clss (fst ?S))))

using *not-bounded-nat-exists-larger[OF unbounded-f-g]* **by** *blast*

The following does not hold anymore with the non-strict version of cardinality in the definition.

{ fix i

assume *no-step cdcl_W-stgy (fst (g i))*

with *g[of i]*

have *False*

proof (*induction rule: cdcl_W-with-restart.induct*)

case (*restart-step T S n*) **note** *H = this(1)* **and** *c = this(2)* **and** *n-s = this(4)*

obtain *S' where cdcl_W-stgy S S'*

using *H c* **by** (*metis gr-implies-not0 relpoup-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 relpoup-imp-rtranclp by metis
then have cdclW-all-struct-inv T
  using inv[of k] rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-stgy-rtranclp-cdclW by blast
moreover have card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k))))
  > card (build-all-simple-clss (atms-of-msu (init-clss (fst ?S))))
  unfolding m[symmetric] using m > f (snd (g k)) f-g-k by linarith
then have card (set-mset (learned-clss T))
  > card (build-all-simple-clss (atms-of-msu (init-clss (fst ?S))))
  by linarith
moreover
  have init-clss (fst (g k)) = init-clss T
    using cdclW-stgy** (fst (g k)) T rtranclp-cdclW-stgy-rtranclp-cdclW rtranclp-cdclW-init-clss
    inv unfolding cdclW-all-struct-inv-def
    by blast
  then have init-clss (fst ?S) = init-clss T
    using init-g[of k] by auto
ultimately show False
  using cdclW-all-struct-inv-learned-clss-bound by (metis Suc-leI card-mono not-less-eq-eq
    build-all-simple-clss-finite)
qed

```

```

lemma cdclW-with-restart-distinct-mset-clauses:
  assumes invR: cdclW-all-struct-inv (fst R) and
  st: cdclW-with-restart R S and
  dist: distinct-mset (clauses (fst R)) and
  R: trail (fst R) = []
  shows distinct-mset (clauses (fst S))
  using assms(2,1,3,4)
proof (induction)
  case (restart-full S T)
  then show ?case using rtranclp-cdclW-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: tranclp-into-rtranclp)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T) using rtranclp-cdclW-stgy-distinct-mset-clauses[of S T]
    unfolding full1-def by (auto dest: relpoup-imp-rtranclp)
  then show ?case using (restart T U) by (metis clauses-restart distinct-mset-union fstI
    mset-le-exists-conv restart.cases state-eq-clauses)
qed
end

```

locale *luby-sequence* =

```

fixes ur :: nat
assumes ur > 0
begin

lemma exists-luby-decomp:
  fixes i :: nat
  shows  $\exists k :: \text{nat}. (2^{k-1} \leq i \wedge i < 2^k - 1) \vee i = 2^k - 1$ 
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
           $(2^{k-1} \leq n \wedge n < 2^k - 1 \vee n = 2^k - 1)$  diff-self-eq-0
          dual-order.trans le-SucI le-imp-less-Suc numeral-2-eq-2 one-le-numeral
          one-le-power zero-less-numeral zero-less-power)
      next
      case end-interv
      then show ?thesis apply – apply (rule exI[of - k]) by auto
      next
      case pow2
      then show ?thesis apply – apply (rule exI[of - k+1]) by auto
    qed
qed

```

Luby sequences are defined by:

- $2^k - 1$, if $i = (2 :: 'a)^k - (1 :: 'a)$
- *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  $2^{k-1} - 1$ 
      else luby-sequence-core  $(i - 2^{k-1} + 1)$ )
by auto
termination
proof (relation less-than, goal-cases)
  case 1
  then show ?case by auto
next

```

```

case (2 i)
let ?k = (SOME k. 2 ^ (k - 1) ≤ i ∧ i < 2 ^ k - 1)
have 2 ^ (?k - 1) ≤ i ∧ i < 2 ^ ?k - 1
  apply (rule someI-ex)
  using 2 exists-luby-decomp by blast
then show ?case

proof -
  have ∀ n na. ¬ (1::nat) ≤ n ∨ 1 ≤ n ^ na
    by (meson one-le-power)
  then have f1: (1::nat) ≤ 2 ^ (?k - 1)
    using one-le-numeral by blast
  have f2: i - 2 ^ (?k - 1) + 2 ^ (?k - 1) = i
    using 2 ^ (?k - 1) ≤ i ∧ i < 2 ^ ?k - 1 le-add-diff-inverse2 by blast
  have f3: 2 ^ ?k - 1 ≠ Suc 0
    using f1 2 ^ (?k - 1) ≤ i ∧ i < 2 ^ ?k - 1 by linarith
  have 2 ^ ?k - (1::nat) ≠ 0
    using 2 ^ (?k - 1) ≤ i ∧ i < 2 ^ ?k - 1 gr-implies-not0 by blast
  then have f4: 2 ^ ?k ≠ (1::nat)
    by linarith
  have f5: ∀ n na. if na = 0 then (n::nat) ^ na = 1 else n ^ na = n * n ^ (na - 1)
    by (simp add: power-eq-if)
  then have ?k ≠ 0
    using f4 by meson
  then have 2 ^ (?k - 1) ≠ Suc 0
    using f5 f3 by presburger
  then have Suc 0 < 2 ^ (?k - 1)
    using f1 by linarith
  then show ?thesis
    using f2 less-than-iff by presburger
qed
qed

```

declare luby-sequence-core.simps[simp del]

lemma two-pover-n-eq-two-power-n'-eq:

assumes $H: (2::nat) ^ (k::nat) - 1 = 2 ^ k' - 1$
 shows $k' = k$

proof -

have $(2::nat) ^ (k::nat) = 2 ^ k'$

using H by (metis One-nat-def Suc-pred zero-less-numeral zero-less-power)

then show ?thesis by simp

qed

lemma luby-sequence-core-two-power-minus-one:

$luby-sequence-core (2^k - 1) = 2^{(k-1)}$ (is ?L = ?K)

proof -

have decomp: $\exists ka. 2 ^ k - 1 = 2 ^ ka - 1$

by auto

have ?L = $2^{((SOME k'. (2::nat) ^ k - 1 = 2 ^ k' - 1) - 1)}$

apply (subst luby-sequence-core.simps, subst decomp)

by simp

moreover have $(SOME k'. (2::nat) ^ k - 1 = 2 ^ k' - 1) = k$

apply (rule some-equality)

apply simp

using *two-pover-n-eq-two-power-n'-eq* by *blast*
 ultimately show *?thesis* by *presburger*
 qed

lemma *different-luby-decomposition-false*:

assumes
 $H: 2 \wedge (k - \text{Suc } 0) \leq i$ and
 $k': i < 2 \wedge k' - \text{Suc } 0$ and
 $k-k': k > k'$

shows *False*

proof -

have $2 \wedge k' - \text{Suc } 0 < 2 \wedge (k - \text{Suc } 0)$

using *k-k' less-eq-Suc-le* by *auto*

then show *?thesis*

using *H k'* by *linarith*

qed

lemma *luby-sequence-core-not-two-power-minus-one*:

assumes
 $k-i: 2 \wedge (k - 1) \leq i$ and
 $i-k: i < 2 \wedge k - 1$

shows *luby-sequence-core i = luby-sequence-core (i - 2 \wedge (k - 1) + 1)*

proof -

have $H: \neg (\exists ka. i = 2 \wedge ka - 1)$

proof (rule *ccontr*)

assume $\neg ?thesis$

then obtain $k': nat$ where $k': i = 2 \wedge k' - 1$ by *blast*

have $(2::nat) \wedge k' - 1 < 2 \wedge k - 1$

using *i-k unfolding k'*.

then have $(2::nat) \wedge k' < 2 \wedge k$

by *linarith*

then have $k' < k$

by *simp*

have $2 \wedge (k - 1) \leq 2 \wedge k' - (1::nat)$

using *k-i unfolding k'*.

then have $(2::nat) \wedge (k-1) < 2 \wedge k'$

by (*metis Suc-diff-1 not-le not-less-eq zero-less-numeral zero-less-power*)

then have $k-1 < k'$

by *simp*

show *False* using $\langle k' < k \rangle \langle k-1 < k' \rangle$ by *linarith*

qed

have $\bigwedge k k'. 2 \wedge (k - \text{Suc } 0) \leq i \implies i < 2 \wedge k - \text{Suc } 0 \implies 2 \wedge (k' - \text{Suc } 0) \leq i \implies$
 $i < 2 \wedge k' - \text{Suc } 0 \implies k = k'$

by (*meson different-luby-decomposition-false linorder-neqE-nat*)

then have $k: (\text{SOME } k. 2 \wedge (k - \text{Suc } 0) \leq i \wedge i < 2 \wedge k - \text{Suc } 0) = k$

using *k-i i-k* by *auto*

show *?thesis*

apply (*subst luby-sequence-core.simps[of i], subst H*)

by (*simp add: k*)

qed

lemma *unbounded-luby-sequence-core: unbounded luby-sequence-core*

unfolding *bounded-def*

proof

```

assume  $\exists b. \forall n. \text{ luby-sequence-core } n \leq b$ 
then obtain  $b$  where  $b: \bigwedge n. \text{ luby-sequence-core } n \leq b$ 
  by metis
have  $\text{ luby-sequence-core } (2^{b+1} - 1) = 2^b$ 
  using luby-sequence-core-two-power-minus-one[of b+1] by simp
moreover have  $(2::\text{nat})^b > b$ 
  by (induction b) auto
ultimately show False using  $b[\text{of } 2^{b+1} - 1]$  by linarith
qed

```

abbreviation *luby-sequence* :: *nat* \Rightarrow *nat* **where**
luby-sequence $n \equiv ur * \text{ luby-sequence-core } n$

lemma *bounded-luby-sequence: unbounded luby-sequence*
using *bounded-const-product[of ur] luby-sequence-axioms*
luby-sequence-def unbounded-luby-sequence-core **by** *blast*

lemma *luby-sequence-core-0: luby-sequence-core 0 = 1*
proof –
have $0: (0::\text{nat}) = 2^0 - 1$
by *auto*
show *?thesis*
by (*subst 0, subst luby-sequence-core-two-power-minus-one*) *simp*
qed

lemma *luby-sequence-core n \geq 1*
proof (*induction n rule: nat-less-induct-case*)
case 0
then show *?case* **by** (*simp add: luby-sequence-core-0*)
next
case (*Suc n*) **note** *IH = this*

consider
 (*interv*) k **where** $2^k - 1 \leq \text{Suc } n$ **and** $\text{Suc } n < 2^{k+1} - 1$
 | (*pow2*) k **where** $\text{Suc } n = 2^k - 1$
using *exists-luby-decomp[of Suc n]* **by** *auto*

then show *?case*
proof *cases*
case *pow2*
show *?thesis*
using *luby-sequence-core-two-power-minus-one pow2* **by** *auto*
next
case *interv*
have $n: \text{Suc } n - 2^k + 1 < \text{Suc } n$
by (*metis Suc-1 Suc-eq-plus1 add.commute add-diff-cancel-left' add-less-mono1 gr0I*
interv(1) interv(2) le-add-diff-inverse2 less-Suc-eq not-le power-0 power-one-right
power-strict-increasing-iff)
show *?thesis*
apply (*subst luby-sequence-core-not-two-power-minus-one[OF interv]*)
using *IH n* **by** *auto*
qed
qed
end

```

locale luby-sequence-restart =
  luby-sequence ur +
  cdclW-ops trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail
  add-init-clss
  add-learned-clss remove-clss update-backtrack-lvl update-conflicting init-state
  restart-state
for
  ur :: nat and
  trail :: 'st ⇒ ('v::linorder, nat, 'v clause) marked-lits and
  init-clss :: 'st ⇒ 'v clauses and
  learned-clss :: 'st ⇒ 'v clauses and
  backtrack-lvl :: 'st ⇒ nat and
  conflicting :: 'st ⇒ 'v clause conflicting-clause and
  cons-trail :: ('v, nat, 'v clause) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-init-clss :: 'v clause ⇒ 'st ⇒ 'st and
  add-learned-clss remove-clss :: 'v clause ⇒ 'st ⇒ 'st and
  update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
  update-conflicting :: 'v clause conflicting-clause ⇒ 'st ⇒ 'st and

  init-state :: 'v::linorder clauses ⇒ 'st and
  restart-state :: 'st ⇒ 'st
begin

sublocale cdclW-ops-restart - - - - - luby-sequence
  apply unfold-locales
  using bounded-luby-sequence by blast

end

end
theory CDCL-W-Incremental
imports CDCL-W-Termination
begin

```

20 Incremental SAT solving

```

context cdclW-ops
begin

```

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl_W-all-struct-inv*

```

definition cdclW-stgy-invariant where
  cdclW-stgy-invariant  $S \longleftrightarrow$ 
    conflict-is-false-with-level  $S$ 
    ∧ no-clause-is-false  $S$ 
    ∧ no-smaller-confl  $S$ 
    ∧ 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 standard
apply (rule cdclW-stgy-ex-lit-of-max-level[of S])
using assms unfolding cdclW-stgy-invariant-def cdclW-all-struct-inv-def apply auto[7]
apply standard
using cdclW cdclW-stgy-not-non-negated-init-clss apply blast
apply standard
apply (rule cdclW-stgy-no-smaller-conflict-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 *rtranclp-cdcl_W-stgy-cdcl_W-stgy-invariant*:

```

assumes
  cdclW: cdclW-stgy** S T and
  inv-s: cdclW-stgy-invariant S and
  inv: cdclW-all-struct-inv S
shows
  cdclW-stgy-invariant T
using assms apply (induction)
apply simp
using cdclW-stgy-cdclW-stgy-invariant rtranclp-cdclW-all-struct-inv-inv
rtranclp-cdclW-stgy-rtranclp-cdclW by blast

```

abbreviation *decr-bt-lvl* **where**

decr-bt-lvl S \equiv *update-backtrack-lvl (backtrack-lvl S - 1) S*

When we add a new clause, we reduce the trail until we get to the first literal included in *C*. Then we can mark the conflict.

fun *cut-trail-wrt-clause* **where**

```

cut-trail-wrt-clause C [] S = S |
cut-trail-wrt-clause C (Marked L - # M) S =
  (if  $-L \in \# C$  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* :: '*v* literal multiset \Rightarrow '*st* \Rightarrow '*st* **where**

```

add-new-clause-and-update C S =
  (if trail S  $\models_{as}$  CNot C
   then update-conflicting (C-Clause C) (add-init-cls C (cut-trail-wrt-clause C (trail S) S))
   else add-init-cls C S)

```

thm *cut-trail-wrt-clause.induct*

lemma *init-clss-cut-trail-wrt-clause[simp]*:

```

init-clss (cut-trail-wrt-clause C M S) = init-clss S
by (induction rule: cut-trail-wrt-clause.induct) auto

```

lemma *learned-clss-cut-trail-wrt-clause[simp]*:

```

learned-clss (cut-trail-wrt-clause C M S) = learned-clss S
by (induction rule: cut-trail-wrt-clause.induct) auto

```

lemma *conflicting-clss-cut-trail-wrt-clause[simp]*:

```

conflicting (cut-trail-wrt-clause C M S) = conflicting S
by (induction rule: cut-trail-wrt-clause.induct) auto

```



```

lemma trail-cut-trail-wrt-clause:
   $\exists M. \text{ trail } S = M @ \text{ trail } (\text{cut-trail-wrt-clause } C (\text{trail } S) S)$ 
proof (induction trail S arbitrary:S rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note  $IH = \text{this}(1)[\text{of } \text{decr-bt-lvl } (\text{tl-trail } S)]$  and  $M = \text{this}(2)[\text{symmetric}]$ 
  then show ?case using Cons-eq-appendI by fastforce+
next
  case (proped L l M) note  $IH = \text{this}(1)[\text{of } (\text{tl-trail } S)]$  and  $M = \text{this}(2)[\text{symmetric}]$ 
  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
     $M: \text{ trail } T = M @ \text{ trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)$ 
    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
     $\text{backtrack-lvl } T = \text{length } (\text{get-all-levels-of-marked } (\text{trail } T))$ 
  shows
     $\text{backtrack-lvl } (\text{cut-trail-wrt-clause } C (\text{trail } T) T) =$ 
     $\text{length } (\text{get-all-levels-of-marked } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))$ 
    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  $IH = \text{this}(1)[\text{of } \text{decr-bt-lvl } (\text{tl-trail } T)]$  and  $M = \text{this}(2)[\text{symmetric}]$ 
    and  $\text{bt} = \text{this}(3)$ 
  then show ?case by auto
next
  case (proped L l M) note  $IH = \text{this}(1)[\text{of } \text{tl-trail } T]$  and  $M = \text{this}(2)[\text{symmetric}]$  and  $\text{bt} = \text{this}(3)$ 
  then show ?case by auto
qed

lemma cut-trail-wrt-clause-get-all-levels-of-marked:
  assumes  $\text{get-all-levels-of-marked } (\text{trail } T) = \text{rev } [\text{Suc } 0..<$ 
     $\text{Suc } (\text{length } (\text{get-all-levels-of-marked } (\text{trail } T)))]$ 
  shows
     $\text{get-all-levels-of-marked } (\text{trail } ((\text{cut-trail-wrt-clause } C (\text{trail } T) T))) = \text{rev } [\text{Suc } 0..<$ 
     $\text{Suc } (\text{length } (\text{get-all-levels-of-marked } (\text{trail } ((\text{cut-trail-wrt-clause } C (\text{trail } T) T)))))]$ 
    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  $IH = this(1)[of\ decr\text{-}bt\text{-}lvl\ (tl\text{-}trail\ T)]$  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  $IH = this(1)[of\ tl\text{-}trail\ T]$  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  $IH = this(1)[of\ decr\text{-}bt\text{-}lvl\ (tl\text{-}trail\ T)]$  and  $M = this(2)[symmetric]$ 
and  $bt = this(3)$ 

  then show ?case apply (cases count  $C$  ( $-L$ ) = 0)
  apply (auto simp: true-annots-true-cls)
  by (smt CNot-def One-nat-def count-single diff-Suc-1 in-CNot-uminus less-numeral-extra(4)
    marked.prems marked-lit.sel(1) mem-Collect-eq true-annot-def true-annot-lit-of-notin-skip
    true-annots-def true-clss-def zero-less-diff)
next
case (proped  $L$   $l$   $M$ ) note  $IH = this(1)[of\ tl\text{-}trail\ T]$  and  $M = this(2)[symmetric]$  and  $bt = this(3)$ 
then show ?case

  apply (cases count  $C$  ( $-L$ ) = 0)
  apply (auto simp: true-annots-true-cls)
  by (smt CNot-def One-nat-def count-single diff-Suc-1 in-CNot-uminus less-numeral-extra(4)
    proped.prems marked-lit.sel(2) mem-Collect-eq true-annot-def true-annot-lit-of-notin-skip
    true-annots-def true-clss-def zero-less-diff)
qed

lemma cut-trail-wrt-clause-hd-trail-in-or-empty-trail:
  ( $(\forall L \in \#C. -L \notin \text{ lits-of } (trail\ T)) \wedge trail\ (cut\text{-}trail\text{-}wrt\text{-}clause\ C\ (trail\ T)\ T) = []$ 
     $\vee (-lit\text{-}of\ (hd\ (trail\ (cut\text{-}trail\text{-}wrt\text{-}clause\ C\ (trail\ T)\ T)))) \in \#C$ 
     $\wedge length\ (trail\ (cut\text{-}trail\text{-}wrt\text{-}clause\ C\ (trail\ T)\ T)) \geq 1$ )
  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  $IH = this(1)[of\ decr\text{-}bt\text{-}lvl\ (tl\text{-}trail\ T)]$  and  $M = this(2)[symmetric]$ 
then show ?case by simp force
next
case (proped  $L$   $l$   $M$ ) note  $IH = this(1)[of\ tl\text{-}trail\ T]$  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-conf!
 $trail\ S \models_{asm} init-clss\ S \implies distinct-mset\ C \implies conflicting\ S = C-True \implies$
 $trail\ S \models_{as} CNot\ C \implies$
 $full\ cdcl_W-stgy$
 $(update-conflicting\ (C-Clause\ C)\ (add-init-clss\ C\ (cut-trail-wrt-clause\ C\ (trail\ S)\ S)))\ T \implies$
 $incremental-cdcl_W\ S\ T \mid$
add-no-conf!
 $trail\ S \models_{asm} init-clss\ S \implies distinct-mset\ C \implies conflicting\ S = C-True \implies$
 $\neg trail\ S \models_{as} CNot\ C \implies$
 $full\ cdcl_W-stgy\ (add-init-clss\ C\ S)\ T \implies$
 $incremental-cdcl_W\ S\ T$

inductive *add-learned-clss* :: 'st \Rightarrow 'v clauses \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**
add-learned-clss-nil: *add-learned-clss* *S* {#} *S* |
add-learned-clss-plus:
 $add-learned-clss\ S\ A\ T \implies add-learned-clss\ S\ (\{ \#x\# \} + A)\ (add-learned-clss\ x\ T)$
declare *add-learned-clss.intros*[intro]

lemma *Ex-add-learned-clss*:
 $\exists T. add-learned-clss\ S\ A\ T$
by (*induction* *A* *arbitrary*: *S* *rule*: *multiset-induct*) (*auto simp*: *union-commute*[of - {#-#}])

lemma *add-learned-clss-trail*:
assumes *add-learned-clss* *S* *U* *T* **and** *no-dup* (*trail* *S*)
shows *trail* *T* = *trail* *S*
using *assms* **by** (*induction* *rule*: *add-learned-clss.induct*) (*simp-all* *add*: *ac-simps*)

lemma *add-learned-clss-learned-clss*:
assumes *add-learned-clss* *S* *U* *T* **and** *no-dup* (*trail* *S*)
shows *learned-clss* *T* = *U* + *learned-clss* *S*
using *assms* **by** (*induction* *rule*: *add-learned-clss.induct*)
(*auto simp*: *ac-simps* *dest*: *add-learned-clss-trail*)

lemma *add-learned-clss-init-clss*:
assumes *add-learned-clss* *S* *U* *T* **and** *no-dup* (*trail* *S*)
shows *init-clss* *T* = *init-clss* *S*
using *assms* **by** (*induction* *rule*: *add-learned-clss.induct*)
(*auto simp*: *ac-simps* *dest*: *add-learned-clss-trail*)

lemma *add-learned-clss-conflicting*:
assumes *add-learned-clss* *S* *U* *T* **and** *no-dup* (*trail* *S*)
shows *conflicting* *T* = *conflicting* *S*
using *assms* **by** (*induction* *rule*: *add-learned-clss.induct*)
(*auto simp*: *ac-simps* *dest*: *add-learned-clss-trail*)

lemma *add-learned-clss-backtrack-lvl*:
assumes *add-learned-clss* *S* *U* *T* **and** *no-dup* (*trail* *S*)
shows *backtrack-lvl* *T* = *backtrack-lvl* *S*
using *assms* **by** (*induction* *rule*: *add-learned-clss.induct*)
(*auto simp*: *ac-simps* *dest*: *add-learned-clss-trail*)

lemma *add-learned-clss-init-state-mempty*[*dest*!]:
 $add-learned-clss\ (init-state\ N)\ \{\#\}\ T \implies T = init-state\ N$
by (*cases* *rule*: *add-learned-clss.cases*) (*auto simp*: *add-learned-clss.cases*)

For multiset larger than 1 element, there is no way to know in which order the clauses are added.

But contrary to a definition *fold-mset*, there is an element.

lemma *add-learned-clss-init-state-single*[*dest!*]:

add-learned-clss (*init-state* *N*) {#*C*#} *T* $\implies T = \text{add-learned-clss } C \text{ (init-state } N)$

by (*induction* {#*C*#} *T* *rule*: *add-learned-clss.induct*)

(*auto simp*: *add-learned-clss.cases ac-simps union-is-single split: split-if-asm*)

thm *rtrancpl-cdcl_W-stgy-no-smaller-confl-inv cdcl_W-stgy-final-state-conclusive*

lemma *cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv*:

assumes

inv-T: *cdcl_W-all-struct-inv* *T* **and**

tr-T-N[*simp*]: *trail* *T* $\models_{asm} N$ **and**

tr-C[*simp*]: *trail* *T* $\models_{as} C \text{Not } C$ **and**

[*simp*]: *distinct-mset* *C*

shows *cdcl_W-all-struct-inv* (*add-new-clause-and-update* *C* *T*) (**is** *cdcl_W-all-struct-inv* ?*T'*)

proof –

let ?*T* = *update-conflicting* (*C-Clause* *C*) (*add-init-clss* *C* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*))

obtain *M* **where**

M: *trail* *T* = *M* @ *trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)

using *trail-cut-trail-wrt-clause*[*of* *T* *C*] **by** *blast*

have *H*[*dest*]: $\bigwedge x. x \in \text{lits-of } (\text{trail } (\text{cut-trail-wrt-clause } C \text{ (trail } T) \text{ } T)) \implies$

$x \in \text{lits-of } (\text{trail } T)$

using *inv-T* *arg-cong*[*OF* *M*, *of* *lits-of*] **by** *auto*

have *H'*[*dest*]: $\bigwedge x. x \in \text{set } (\text{trail } (\text{cut-trail-wrt-clause } C \text{ (trail } T) \text{ } T)) \implies x \in \text{set } (\text{trail } T)$

using *inv-T* *arg-cong*[*OF* *M*, *of* *set*] **by** *auto*

have *H-proped*: $\bigwedge x. x \in \text{set } (\text{get-all-mark-of-propagated } (\text{trail } (\text{cut-trail-wrt-clause } C \text{ (trail } T) \text{ } T))) \implies x \in \text{set } (\text{get-all-mark-of-propagated } (\text{trail } T))$

using *inv-T* *arg-cong*[*OF* *M*, *of* *get-all-mark-of-propagated*] **by** *auto*

have [*simp*]: *no-strange-atm* ?*T*

using *inv-T* **unfolding** *cdcl_W-all-struct-inv-def no-strange-atm-def add-new-clause-and-update-def*

cdcl_W-M-level-inv-def

by (*auto dest!*: *H* *H'*)

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* *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* *C* (*trail* *T*) *T*))

by *auto*

have *consistent-interp* (*lits-of* (*M* @ *trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)))

using *M-lev* **unfolding** *cdcl_W-M-level-inv-def* **unfolding** *M[symmetric]* **by** *auto*

then have [*simp*]: *consistent-interp* (*lits-of* (*trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)))

unfolding *consistent-interp-def* **by** *auto*

have [*simp*]: *cdcl_W-M-level-inv* ?*T*

unfolding *cdcl_W-M-level-inv-def* **apply** (*auto dest*: *H* *H'*)

simp: *M-lev* *cdcl_W-M-level-inv-def cut-trail-wrt-clause-backtrack-lvl-length-marked*)

using *M-lev* *cut-trail-wrt-clause-get-all-levels-of-marked*[*of* *T* *C*]

by (*auto simp*: *cdcl_W-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** *cdcl_W-all-struct-inv-def* **by** *auto*

```

have distinct-cdclW-state T
  using inv-T unfolding cdclW-all-struct-inv-def by auto
then have [simp]: distinct-cdclW-state ?T
  unfolding distinct-cdclW-state-def by auto

have cdclW-conflicting T
  using inv-T unfolding cdclW-all-struct-inv-def by auto
have trail ?T  $\models_{as}$  CNot C
  by (simp add: cut-trail-wrt-clause-CNot-trail)
then have [simp]: cdclW-conflicting ?T
  unfolding cdclW-conflicting-def apply simp
  by (metis M  $\langle$ cdclW-conflicting T $\rangle$  append-assoc cdclW-conflicting-decomp(2))

have decomp-T: all-decomposition-implies-m (init-clss T) (get-all-marked-decomposition (trail T))
  using inv-T unfolding cdclW-all-struct-inv-def by auto
have all-decomposition-implies-m (init-clss ?T)
  (get-all-marked-decomposition (trail ?T))
  unfolding all-decomposition-implies-def
proof clarify
  fix a b
  assume (a, b)  $\in$  set (get-all-marked-decomposition (trail ?T))
  from in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend[OF this]
  obtain b' where
    (a, b' @ b)  $\in$  set (get-all-marked-decomposition (trail T))
    using M by simp metis
  then have ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ' set a  $\cup$  set-mset (init-clss ?T)
     $\models_{ps}$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ' set (b @ b')
    using decomp-T unfolding all-decomposition-implies-def

    apply auto
    by (metis (no-types, lifting) case-prodD set-append sup commute true-clss-clss-insert-l)

  then show ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ' set a  $\cup$  set-mset (init-clss ?T)
     $\models_{ps}$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ' set b
    by (auto simp: image-Un)
qed

have [simp]: cdclW-learned-clause ?T
  using inv-T unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def
  by (auto dest!: H-proped simp: clauses-def)
show ?thesis
  using  $\langle$ all-decomposition-implies-m (init-clss ?T)
  (get-all-marked-decomposition (trail ?T)) $\rangle$ 
  unfolding cdclW-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
qed

lemma cdclW-all-struct-inv-add-new-clause-and-update-cdclW-stgy-inv:
  assumes
    inv-s: cdclW-stgy-invariant T and
    inv: cdclW-all-struct-inv T and
    tr-T-N[simp]: trail T  $\models_{asm}$  N and
    tr-C[simp]: trail T  $\models_{as}$  CNot C and
    [simp]: distinct-mset C
  shows cdclW-stgy-invariant (add-new-clause-and-update C T) (is cdclW-stgy-invariant ?T')
proof -

```

```

have  $cdcl_W$ -all-struct-inv ? $T'$ 
  using  $cdcl_W$ -all-struct-inv-add-new-clause-and-update- $cdcl_W$ -all-struct-inv assms by blast
then have
   $no\_dup\_cut\_T[simp]$ :  $no\_dup$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ )) and
   $n\_d[simp]$ :  $no\_dup$  ( $trail$   $T$ )
  using  $cdcl_W$ - $M$ -level-inv-decomp(2)  $cdcl_W$ -all-struct-inv-def inv
   $n\_dup\_no\_dup\_trail\_cut\_trail\_wrt\_clause$  by blast+
then have  $trail$  ( $add\_new\_clause\_and\_update$   $C$   $T$ )  $\models_{as}$   $CNot$   $C$ 
  by ( $simp$  add:  $add\_new\_clause\_and\_update\_def$   $cut\_trail\_wrt\_clause$ - $CNot$ - $trail$ 
     $cdcl_W$ - $M$ -level-inv-def  $cdcl_W$ -all-struct-inv-def)
obtain  $MT$  where
   $MT$ :  $trail$   $T = MT$  @  $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ )
  using  $trail\_cut\_trail\_wrt\_clause$  by blast
consider
  ( $false$ )  $\forall L \in \#C. - L \notin lits\_of$  ( $trail$   $T$ ) and  $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ) = []
  | ( $not\_false$ ) -  $lit\_of$  ( $hd$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ )))  $\in \#C$  and
     $1 \leq length$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ))
  using  $cut\_trail\_wrt\_clause$ - $hd$ - $trail$ -in-or-empty-trail[of  $C$   $T$ ] by auto
then show ?thesis
proof cases
  case  $false$  note  $C = this(1)$  and  $empty\_tr = this(2)$ 
  then have [ $simp$ ]:  $C = \{\#\}$ 
    by ( $simp$  add:  $in\_CNot$ -implies- $uminus(2)$  multiset-eqI)
  show ?thesis
    using  $empty\_tr$  unfolding  $cdcl_W$ -stgy-invariant-def  $no\_smaller\_confl\_def$ 
     $cdcl_W$ -all-struct-inv-def by ( $auto$   $simp$ :  $add\_new\_clause\_and\_update\_def$ )
next
  case  $not\_false$  note  $C = this(1)$  and  $l = this(2)$ 
  let ? $L = - lit\_of$  ( $hd$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ )))
  have  $get\_all\_levels\_of\_marked$  ( $trail$  ( $add\_new\_clause\_and\_update$   $C$   $T$ )) =
     $rev$  [ $1..<1 + length$  ( $get\_all\_levels\_of\_marked$  ( $trail$  ( $add\_new\_clause\_and\_update$   $C$   $T$ )))]
    using  $\langle cdcl_W$ -all-struct-inv ? $T' \rangle$  unfolding  $cdcl_W$ -all-struct-inv-def  $cdcl_W$ - $M$ -level-inv-def
    by blast
moreover
  have  $backtrack\_lvl$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ) =
     $length$  ( $get\_all\_levels\_of\_marked$  ( $trail$  ( $add\_new\_clause\_and\_update$   $C$   $T$ )))
    using  $\langle cdcl_W$ -all-struct-inv ? $T' \rangle$  unfolding  $cdcl_W$ -all-struct-inv-def  $cdcl_W$ - $M$ -level-inv-def
    by ( $auto$   $simp$ :  $add\_new\_clause\_and\_update\_def$ )
moreover
  have  $no\_dup$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ))
    using  $\langle cdcl_W$ -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  $atm\_of$  ? $L \notin atm\_of$  '  $lits\_of$  ( $tl$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ )))
    apply ( $cases$   $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ))
    apply ( $auto$ )
    using  $Marked$ - $Propagated$ -in-iff-in-lits-of  $defined$ -lit-map by blast

ultimately have  $L$ :  $get\_level$  ( $- ?L$ ) ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ))
  =  $length$  ( $get\_all\_levels\_of\_marked$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ )))
  using  $get\_level$ - $get\_rev$ - $level$ - $get\_all\_levels\_of\_marked$ [OF
     $\langle atm\_of$  ? $L \notin atm\_of$  '  $lits\_of$  ( $tl$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ))),
     $of$  [ $hd$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ))]]
    apply ( $cases$   $trail$  ( $add\_init\_cls$   $C$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ ));
     $cases$   $hd$  ( $trail$  ( $cut\_trail\_wrt\_clause$   $C$  ( $trail$   $T$ )  $T$ )))

```

```

using  $l$  by (auto split: split-if-asm
  simp: rev-swap[symmetric] add-new-clause-and-update-def
  simp del:)

have  $L'$ : length (get-all-levels-of-marked (trail (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ )))
  = backtrack-lvl (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ )
using <cdclW-all-struct-inv ? $T'$ > unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
by (auto simp: add-new-clause-and-update-def)

have [simp]: no-smaller-confl (update-conflicting ( $C$ -Clause  $C$ )
  (add-init-cls  $C$  (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ )))
unfolding no-smaller-confl-def
proof (clarify, goal-cases)
  case (1  $M$   $K$   $i$   $M'$   $D$ )
  then consider
    ( $DC$ )  $D = C$ 
  | ( $D-T$ )  $D \in \#$  clauses  $T$ 
  by (auto simp: clauses-def split: split-if-asm)
then show False
proof cases
  case  $D-T$ 
  have no-smaller-confl  $T$ 
    using inv-s unfolding cdclW-stgy-invariant-def by auto
  have ( $MT @ M'$ ) @ Marked  $K$   $i$  #  $M =$  trail  $T$ 
    using  $MT$  1(1) by auto
  thus False using  $D-T$  <no-smaller-confl  $T$ > 1(3) unfolding no-smaller-confl-def by blast
next
  case  $DC$  note -[simp] = this
  then have atm-of ( $-?L$ )  $\in$  atm-of ' (lits-of  $M$ )
    using 1(3)  $C$  in-CNot-implies-uminus(2) by blast
  moreover
    have lit-of (hd ( $M' @$  Marked  $K$   $i$  # [])) =  $-?L$ 
      using  $l$  1(1)[symmetric] inv
      by (cases trail (add-init-cls  $C$  (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ )))
      (auto dest!: arg-cong[of - # - - hd] simp: hd-append cdclW-all-struct-inv-def
        cdclW-M-level-inv-def)
    from arg-cong[OF this, of atm-of]
    have atm-of ( $-?L$ )  $\in$  atm-of ' (lits-of ( $M' @$  Marked  $K$   $i$  # []))
      by (cases ( $M' @$  Marked  $K$   $i$  # [])) auto
    moreover have no-dup (trail (cut-trail-wrt-clause  $C$  (trail  $T$ )  $T$ ))
      using <cdclW-all-struct-inv ? $T'$ > unfolding cdclW-all-struct-inv-def
      cdclW-M-level-inv-def by (auto simp: add-new-clause-and-update-def)
    ultimately show False
      unfolding 1(1)[symmetric, simplified]
      apply auto
      using Marked-Propagated-in-iff-in-lits-of defined-lit-map apply blast
      by (metis IntI Marked-Propagated-in-iff-in-lits-of defined-lit-map empty-iff)
  qed
qed
show ?thesis using  $L$   $L'$   $C$ 
  unfolding cdclW-stgy-invariant-def
  unfolding cdclW-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
qed
qed

```

lemma *full-cdcl_W-stgy-inv-normal-form*:

assumes

full: *full cdcl_W-stgy S T* **and**

inv-s: *cdcl_W-stgy-invariant S* **and**

inv: *cdcl_W-all-struct-inv S*

shows *conflicting T = C-Clause {#} ∧ unsatisfiable (set-mset (init-clss S))*

∨ *conflicting T = C-True ∧ trail T ⊨_{asm} init-clss S ∧ satisfiable (set-mset (init-clss S))*

proof –

have *no-step cdcl_W-stgy T*

using *full unfolding full-def by blast*

moreover have *cdcl_W-all-struct-inv T* **and** *inv-s: cdcl_W-stgy-invariant T*

apply (*metis cdcl_W-ops.rtranclp-cdcl_W-stgy-rtranclp-cdcl_W cdcl_W-ops-axioms full full-def inv rtranclp-cdcl_W-all-struct-inv-inv*)

by (*metis full full-def inv inv-s rtranclp-cdcl_W-stgy-cdcl_W-stgy-invariant*)

ultimately have *conflicting T = C-Clause {#} ∧ unsatisfiable (set-mset (init-clss T))*

∨ *conflicting T = C-True ∧ trail T ⊨_{asm} init-clss T*

using *cdcl_W-stgy-final-state-conclusive[of T] full*

unfolding *cdcl_W-all-struct-inv-def cdcl_W-stgy-invariant-def full-def by fast*

moreover have *consistent-interp (lits-of (trail T))*

using *⟨cdcl_W-all-struct-inv T⟩ unfolding cdcl_W-all-struct-inv-def cdcl_W-M-level-inv-def by auto*

moreover have *init-clss S = init-clss T*

using *inv unfolding cdcl_W-all-struct-inv-def*

by (*metis rtranclp-cdcl_W-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-cdcl_W S T* **and**

inv: *cdcl_W-all-struct-inv S* **and**

s-inv: *cdcl_W-stgy-invariant S*

shows

cdcl_W-all-struct-inv T **and**

cdcl_W-stgy-invariant T

using *inc*

proof (*induction*)

case (*add-confl C T*)

let *?T = (update-conflicting (C-Clause C) (add-init-cls C (cut-trail-wrt-clause C (trail S) S)))*

have *cdcl_W-all-struct-inv ?T* **and** *inv-s-T: cdcl_W-stgy-invariant ?T*

using *add-confl.hyps(1,2,4) add-new-clause-and-update-def*

cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv inv **apply** *auto[1]*

using *add-confl.hyps(1,2,4) add-new-clause-and-update-def*

cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-stgy-inv inv s-inv **by** *auto*

case 1 show *?case*

by (*metis add-confl.hyps(1,2,4,5) add-new-clause-and-update-def*

cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv

rtranclp-cdcl_W-all-struct-inv-inv rtranclp-cdcl_W-stgy-rtranclp-cdcl_W full-def inv)

case 2 show *?case*

by (*metis inv-s-T add-confl.hyps(1,2,4,5) add-new-clause-and-update-def*

cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv full-def inv

rtranclp-cdcl_W-stgy-cdcl_W-stgy-invariant)

next


```

case (add-no-confl C T)
case 1
have cdclW-all-struct-inv (add-init-cls C S)
  using inv ⟨distinct-mset C⟩ unfolding cdclW-all-struct-inv-def no-strange-atm-def
  cdclW-M-level-inv-def distinct-cdclW-state-def cdclW-conflicting-def cdclW-learned-clause-def
  by (auto simp: all-decomposition-implies-insert-single clauses-def)
then show ?case
  using add-no-confl(5) unfolding full-def by (auto intro: rtranclp-cdclW-stgy-cdclW-all-struct-inv)
case 2 have cdclW-stgy-invariant (add-init-cls C S)
  using s-inv ⟨¬ trail S ⊨as CNot C⟩ inv unfolding cdclW-stgy-invariant-def no-smaller-confl-def
  eq-commute[of - trail -] cdclW-M-level-inv-def cdclW-all-struct-inv-def
  by (auto simp: true-annots-true-cls-def-iff-negation-in-model clauses-def split: split-if-asm)
then show ?case
  by (metis ⟨cdclW-all-struct-inv (add-init-cls C S)⟩ add-no-confl.hyps(5) full-def
      rtranclp-cdclW-stgy-cdclW-stgy-invariant)
qed

```

lemma *rtranclp-incremental-cdcl_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 apply induction
  using inv apply simp
  using s-inv apply simp
using incremental-cdclW-inv by blast+

```

lemma *incremental-conclusive-state*:

```

assumes
  inc: incremental-cdclW S T and
  inv: cdclW-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows conflicting T = C-Clause {#} ∧ unsatisfiable (set-mset (init-cls T))
  ∨ conflicting T = C-True ∧ trail T ⊨asm init-cls T ∧ satisfiable (set-mset (init-cls T))
using inc apply induction

```

```

apply (metis Nitpick.rtranclp-unfold add-confl full-cdclW-stgy-inv-normal-form full-def
  incremental-cdclW-inv(1) incremental-cdclW-inv(2) inv s-inv)
by (metis (full-types) rtranclp-unfold add-no-confl full-cdclW-stgy-inv-normal-form
  full-def incremental-cdclW-inv(1) incremental-cdclW-inv(2) inv s-inv)

```

lemma *tranclp-incremental-correct*:

```

assumes
  inc: incremental-cdclW++ S T and
  inv: cdclW-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows conflicting T = C-Clause {#} ∧ unsatisfiable (set-mset (init-cls T))
  ∨ conflicting T = C-True ∧ trail T ⊨asm init-cls T ∧ satisfiable (set-mset (init-cls 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)

```

lemma *blocked-induction-with-marked*:

assumes

n-d: *no-dup* ($L \# M$) **and**

nil: $P []$ **and**

append: $\bigwedge M L M'. P M \implies \text{is-marked } L \implies \forall m \in \text{set } M'. \neg \text{is-marked } m \implies \text{no-dup } (L \# M' @ M) \implies$

$P (L \# M' @ M)$ **and**

L: *is-marked* *L*

shows

$P (L \# M)$

using *n-d L*

proof (*induction card* $\{L' \in \text{set } M. \text{is-marked } L'\}$ *arbitrary*: $L M$)

case 0 **note** $n = \text{this}(1)$ **and** $n\text{-d} = \text{this}(2)$ **and** $L = \text{this}(3)$

then have $\forall m \in \text{set } M. \neg \text{is-marked } m$ **by** *auto*

then show ?*case* **using** *append*[*of* $[] L M$] *L nil n-d* **by** *auto*

next

case (*Suc n*) **note** $IH = \text{this}(1)$ **and** $n = \text{this}(2)$ **and** $n\text{-d} = \text{this}(3)$ **and** $L = \text{this}(4)$

have $\exists L' \in \text{set } M. \text{is-marked } L'$

proof (*rule ccontr*)

assume $\neg ?thesis$

then have $H: \{L' \in \text{set } M. \text{is-marked } L'\} = \{\}$

by *auto*

show *False* **using** *n unfolding H* **by** *auto*

qed

then obtain $L' M' M''$ **where**

$M: M = M' @ L' \# M''$ **and**

$L': \text{is-marked } L'$ **and**

$nm: \forall m \in \text{set } M'. \neg \text{is-marked } m$

by (*auto elim!*: *split-list-first-propE*)

have $\text{Suc } n = \text{card } \{L' \in \text{set } M. \text{is-marked } L'\}$

using *n* .

moreover have $\{L' \in \text{set } M. \text{is-marked } L'\} = \{L'\} \cup \{L' \in \text{set } M''. \text{is-marked } L'\}$

using *nm L' n-d unfolding M* **by** *auto*

moreover have $L' \notin \{L' \in \text{set } M''. \text{is-marked } L'\}$

using *n-d unfolding M* **by** *auto*

ultimately have $n = \text{card } \{L'' \in \text{set } M''. \text{is-marked } L''\}$

using *n L' by auto*

then have $P (L' \# M'')$ **using** *IH L' n-d M* **by** *auto*

then show ?*case* **using** *append*[*of* $L' \# M'' L M$] *nm L n-d unfolding M* **by** *blast*

qed

lemma *trail-bloc-induction*:

assumes

n-d: *no-dup* *M* **and**

nil: $P []$ **and**

append: $\bigwedge M L M'. P M \implies \text{is-marked } L \implies \forall m \in \text{set } M'. \neg \text{is-marked } m \implies \text{no-dup } (L \# M' @ M) \implies$

$P (L \# M' @ M)$ **and**

append-nm: $\bigwedge M' M''. P M' \implies M = M'' @ M' \implies \forall m \in \text{set } M''. \neg \text{is-marked } m \implies P M$

shows

$P M$

proof (*cases* $\{L' \in \text{set } M. \text{is-marked } L'\} = \{\}$)

case *True*

then show ?*thesis* **using** *append-nm*[*of* $[] M$] *nil* **by** *auto*

```

next
case False
then have  $\exists L' \in \text{set } M. \text{is-marked } L'$ 
  by auto
then obtain  $L' M' M''$  where
   $M: M = M' @ L' \# M''$  and
   $L': \text{is-marked } L'$  and
   $nm: \forall m \in \text{set } M'. \neg \text{is-marked } m$ 
  by (auto elim!: split-list-first-propE)
have  $P (L' \# M'')$ 
  apply (rule blocked-induction-with-marked)
  using n-d unfolding M apply simp
  using nil apply simp
  using append apply simp
  using L' by auto
then show ?thesis
  using append-nm[of - M'] nm unfolding M by simp
qed

inductive Tcons :: ('v, nat, 'v clause) marked-lits  $\Rightarrow$  ('v, nat, 'v clause) marked-lits  $\Rightarrow$  bool
  for M :: ('v, nat, 'v clause) marked-lits where
    Tcons M [] |
    Tcons M M'  $\Rightarrow M = M'' @ M' \Rightarrow (\forall m \in \text{set } M''. \neg \text{is-marked } m) \Rightarrow \text{Tcons } M (M'' @ M') |$ 
    Tcons M M'  $\Rightarrow \text{is-marked } L \Rightarrow M = M''' @ L \# M'' @ M' \Rightarrow (\forall m \in \text{set } M''. \neg \text{is-marked } m) \Rightarrow$ 
    Tcons M (L # M'' @ M')

lemma Tcons-same-end: Tcons M M'  $\Rightarrow \exists M''. M = M'' @ M'$ 
  by (induction rule: Tcons.induct) auto

end

end

```

21 2-Watched-Literal

```

theory CDCL-Two-Watched-Literals
imports CDCL-WNOT
begin

```

21.1 Datastructure and Access Functions

Only the 2-watched literals have to be verified here: the backtrack level and the trail that appear in the state are not related to the 2-watched algorithm.

```

datatype 'v twl-clause =
  TWL-Clause (watched: 'v clause) (unwatched: 'v clause)

```

```

abbreviation raw-clause :: 'v twl-clause  $\Rightarrow$  'v clause where
  raw-clause C  $\equiv$  watched C + unwatched C

```

```

datatype ('v, 'vl, 'mark) twl-state =
  TWL-State (trail: ('v, 'vl, 'mark) marked-lits) (init-clss: 'v twl-clause multiset)
    (learned-clss: 'v twl-clause multiset) (backtrack-lvl: 'vl)
    (conflicting: 'v clause conflicting-clause)

```

abbreviation *raw-init-clss* **where**

raw-init-clss $S \equiv \text{image-mset } \text{raw-clause } (\text{init-clss } S)$

abbreviation *raw-learned-clss* **where**

raw-learned-clss $S \equiv \text{image-mset } \text{raw-clause } (\text{learned-clss } S)$

abbreviation *clauses* **where**

clauses $S \equiv \text{init-clss } S + \text{learned-clss } S$

abbreviation *raw-clauses* **where**

raw-clauses $S \equiv \text{image-mset } \text{raw-clause } (\text{clauses } S)$

definition

candidates-propagate $:: ('v, 'lv, 'mark) \text{ twl-state} \Rightarrow ('v \text{ literal} \times 'v \text{ clause}) \text{ set}$

where

candidates-propagate $S =$

$\{(L, \text{raw-clause } C) \mid L \in C.$

$C \in \# \text{ clauses } S \wedge \text{watched } C - \text{mset-set } (\text{uminus } ' \text{ lits-of } (\text{trail } S)) = \{\#L\# \} \wedge$

$\text{undefined-lit } (\text{trail } S) L\}$

definition *candidates-conflict* $:: ('v, 'lv, 'mark) \text{ twl-state} \Rightarrow 'v \text{ clause set}$ **where**

candidates-conflict $S =$

$\{\text{raw-clause } C \mid C. C \in \# \text{ clauses } S \wedge \text{watched } C \subseteq \# \text{ mset-set } (\text{uminus } ' \text{ lits-of } (\text{trail } S))\}$

primrec (*nonexhaustive*) *index* $:: 'a \text{ list} \Rightarrow 'a \Rightarrow \text{nat}$ **where**

index $(a \# l) c = (\text{if } a = c \text{ then } 0 \text{ else } 1 + \text{index } l c)$

lemma *index-nth*:

$a \in \text{set } l \implies l ! (\text{index } l a) = a$

by (*induction* l) *auto*

21.2 Invariants

We need the following property about updates: if there is a literal L with $-L$ in the trail, and L is not watched, then it stays unwatched; i.e., while updating with *rewatch* it does not get swap with a watched literal L' such that $-L'$ is in the trail.

primrec *watched-decided-most-recently* $:: ('v, 'lv, 'mark) \text{ marked-lit list} \Rightarrow 'v \text{ twl-clause} \Rightarrow \text{bool}$

where

watched-decided-most-recently $M (TWL\text{-Clause } W UW) \longleftrightarrow$

$(\forall L' \in \# W. \forall L \in \# UW.$

$-L' \in \text{lits-of } M \longrightarrow -L \in \text{lits-of } M \longrightarrow L \notin \# W \longrightarrow$

$\text{index } (\text{map lit-of } M) (-L') \leq \text{index } (\text{map lit-of } M) (-L))$

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-tw-clcs* $:: ('v, 'lv, 'mark) \text{ marked-lit list} \Rightarrow 'v \text{ twl-clause} \Rightarrow \text{bool}$ **where**

wf-tw-clcs $M (TWL\text{-Clause } W UW) \longleftrightarrow$

$\text{distinct-mset } W \wedge \text{size } W \leq 2 \wedge (\text{size } W < 2 \longrightarrow \text{set-mset } UW \subseteq \text{set-mset } W) \wedge$

$(\forall L \in \# W. -L \in \text{lits-of } M \longrightarrow (\forall L' \in \# UW. L' \notin \# W \longrightarrow -L' \in \text{lits-of } M)) \wedge$

watched-decided-most-recently $M (TWL\text{-Clause } W UW)$

lemma $-L \in \text{lits-of } M \implies \{i. \text{map lit-of } M ! i = -L\} \neq \{\}$

unfolding *set-map-lit-of-lits-of* [*symmetric*] *set-conv-nth*

by (*smt Collect-empty-eq mem-Collect-eq*)

lemma *size-mset-2*: $\text{size } x1 = 2 \longleftrightarrow (\exists a b. x1 = \{\#a, b\# \})$

by (*metis* (*no-types*, *hide-lams*) *Suc-eq-plus1 one-add-one size-1-singleton-mset size-Diff-singleton size-Suc-Diff1 size-eq-Suc-imp-eq-union size-single union-single-eq-diff union-single-eq-member*)

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

lemma *wf-twl-cla-annotation-indepndant*:
assumes *M*: *map lit-of* *M* = *map lit-of* *M'*
shows *wf-twl-cla* *M* (*TWL-Clause* *W UW*) \longleftrightarrow *wf-twl-cla* *M'* (*TWL-Clause* *W UW*)

proof –

have *lits-of* *M* = *lits-of* *M'*
using *arg-cong*[*OF* *M*, *of set*] **by** (*simp add: lits-of-def*)
then show *?thesis*
by (*simp add: lits-of-def* *M*)

qed

lemma *wf-twl-cla-wf-twl-cla-tl*:
assumes *wf*: *wf-twl-cla* *M* *C* **and** *n-d*: *no-dup* *M*
shows *wf-twl-cla* (*tl* *M*) *C*

proof (*cases* *M*)

case *Nil*
then show *?thesis* **using** *wf*
by (*cases* *C*) (*simp add: wf-twl-cla.simps*[*of tl -*])

next

case (*Cons* *l* *M'*) **note** *M* = *this*(1)
obtain *W UW* **where** *C*: *C* = *TWL-Clause* *W UW*
by (*cases* *C*)
{ **fix** *L L'*
assume
LW: *L* $\in \#$ *W* **and**
LM: $\neg L \in \text{lits-of } M'$ **and**
L'UW: *L'* $\in \#$ *UW* **and**
count *W* *L'* = 0
then have
L'M: $\neg L' \in \text{lits-of } M$
using *wf* **by** (*auto simp: C M*)
have *watched-decided-most-recently* *M* *C*
using *wf* **by** (*auto simp: C*)
then have
index (*map lit-of* *M*) ($\neg L$) \leq *index* (*map lit-of* *M*) ($\neg L'$)
using *LM L'M L'UW LW* $\langle \text{count } W L' = 0 \rangle$
by (*metis* (*no-types*, *lifting*) *C M bspec-mset insert-iff less-not-refl2 lits-of-cons watched-decided-most-recently.simps*)
then have $\neg L' \in \text{lits-of } M'$
using $\langle \text{count } W L' = 0 \rangle$ *LW L'M* **by** (*auto simp: C M split: split-if-asm*)
}

moreover

{
fix *L' L*
assume
L' $\in \#$ *W* **and**
L $\in \#$ *UW* **and**
L'M: $\neg L' \in \text{lits-of } M'$ **and**
 $\neg L \in \text{lits-of } M'$ **and**

```

    L ∉# W
  moreover
    have lit-of l ≠ - L'
    using n-d unfolding M
      by (metis (no-types) L'M M Marked-Propagated-in-iff-in-lits-of defined-lit-map
        distinct.simps(2) list.simps(9) set-map)
    moreover have watched-decided-most-recently M C
      using wf by (auto simp: C)
    ultimately have index (map lit-of M') (- L') ≤ index (map lit-of M') (- L)
      by (fastforce simp: M C split: split-if-asm)
  }
  moreover have distinct-mset W and size W ≤ 2 and (size W < 2 ⟶ set-mset UW ⊆ set-mset
W)
    using wf by (auto simp: C M)
  ultimately show ?thesis by (auto simp add: M C)
qed

```

definition *wf-twl-state* :: ('v, 'wl, 'mark) *twl-state* \Rightarrow bool **where**
wf-twl-state S $\longleftrightarrow (\forall C \in \# \text{ clauses } S. \text{wf-twl-cl} (\text{trail } S) C) \wedge \text{no-dup} (\text{trail } S)$

lemma *wf-candidates-propagate-sound*:

assumes *wf*: *wf-twl-state* S **and**
cand: (L, C) \in *candidates-propagate* S
shows *trail* S \models_{as} CNot (mset-set (set-mset C - {L})) \wedge *undefined-lit* (*trail* S) L

proof

def M \equiv *trail* S
def N \equiv *init-clss* S
def U \equiv *learned-clss* S

note MNU-defs [*simp*] = M-def N-def U-def

obtain Cw **where** cw:

C = *raw-clause* Cw
Cw $\in \#$ N + U
watched Cw - mset-set (uminus ' *lits-of* M) = {#L#}
undefined-lit M L
using *cand* **unfolding** *candidates-propagate-def* MNU-defs **by** *blast*

obtain W UW **where** cw-eq: Cw = *TWL-Clause* W UW
by (*case-tac* Cw, *blast*)

have l-w: L $\in \#$ W

by (metis *Multiset.diff-le-self* cw(3) cw-eq mset-leD multi-member-last *twl-clause.sel*(1))

have wf-c: *wf-twl-cl* M Cw

using wf (Cw $\in \#$ N + U) **unfolding** *wf-twl-state-def* **by** *simp*

have w-nw:

distinct-mset W
size W < 2 \implies set-mset UW \subseteq set-mset W
 $\bigwedge L L'. L \in \# W \implies -L \in \text{lits-of } M \implies L' \in \# UW \implies L' \notin \# W \implies -L' \in \text{lits-of } M$
using wf-c **unfolding** cw-eq **by** *auto*

have $\forall L' \in \text{set-mset } C - \{L\}. -L' \in \text{lits-of } M$

proof (*cases* size W < 2)

```

case True
moreover have size W ≠ 0
  using cw(3) cw-eq by auto
ultimately have size W = 1
  by linarith
then have w: W = {#L#}
  by (metis (no-types, lifting) Multiset.diff-le-self cw(3) cw-eq single-not-empty
    size-1-singleton-mset subset-mset.add-diff-inverse union-is-single twl-clause.sel(1))
from True have set-mset UW ⊆ set-mset W
  using w-nw(2) by blast
then show ?thesis
  using w cw(1) cw-eq by auto
next
case sz2: False
show ?thesis
proof
  fix L'
  assume l': L' ∈ set-mset C - {L}
  have ex-la: ∃ La. La ≠ L ∧ La ∈# W
  proof (cases W)
    case empty
    thus ?thesis
      using l-w by auto
  next
    case lb: (add W' Lb)
    show ?thesis
    proof (cases W')
      case empty
      thus ?thesis
        using lb sz2 by simp
    next
      case lc: (add W'' Lc)
      thus ?thesis
        by (metis add-gr-0 count-union distinct-mset-single-add lb union-single-eq-member
          w-nw(1))
    qed
  qed
then obtain La where la: La ≠ L La ∈# W
  by blast
then have La ∈# mset-set (uminus ' lits-of M)
  using cw(3)[unfolded cw-eq, simplified, folded M-def]
  by (metis count-diff count-single diff-zero not-gr0)
then have nla: -La ∈ lits-of M
  by auto
then show -L' ∈ lits-of M

proof -
  have f1: L' ∈ set-mset C
    using l' by blast
  have f2: L' ∉ {L}
    using l' by fastforce
  have ∧l L. - (l::'a literal) ∈ L ∨ l ∉ uminus ' L
    by force
  then have ∧l. - l ∈ lits-of M ∨ count {#L#} l = count (C - UW) l
    by (metis (no-types) add-diff-cancel-right' count-diff count-mset-set(3) cw(1) cw(3))

```

```

      cw-eq diff-zero twl-clause.sel(2))
  then show ?thesis
    by (smt comm-monoid-add-class.add-0 cw(1) cw-eq diff-union-cancelR ex-la f1 f2 insertCI
        less-numeral-extra(3) mem-set-mset-iff plus-multiset.rep-eq single.rep-eq
        twl-clause.sel(1) twl-clause.sel(2) w-nw(3))
  qed
qed
qed
then show trail S  $\models_{as}$  CNot (mset-set (set-mset C - {L}))
  unfolding true-annots-def by auto

show undefined-lit (trail S) L
  using cw(4) M-def by blast
qed

lemma wf-candidates-propagate-complete:
  assumes wf: wf-twl-state S and
    c-mem: C  $\in \#$  raw-clauses S and
    l-mem: L  $\in \#$  C and
    unsat: trail S  $\models_{as}$  CNot (mset-set (set-mset C - {L})) and
    undef: undefined-lit (trail S) L
  shows (L, C)  $\in$  candidates-propagate S
proof -
  def M  $\equiv$  trail S
  def N  $\equiv$  init-clss S
  def U  $\equiv$  learned-clss S

  note MNU-defs [simp] = M-def N-def U-def

  obtain Cw where cw: C = raw-clause Cw Cw  $\in \#$  N + U
    using c-mem by force

  obtain W UW where cw-eq: Cw = TWL-Clause W UW
    by (case-tac Cw, blast)

  have wf-c: wf-twl-cls M Cw
    using wf cw(2) unfolding wf-twl-state-def by simp

  have w-nw:
    distinct-mset W
    size W < 2  $\implies$  set-mset UW  $\subseteq$  set-mset W
     $\bigwedge L L'. L \in \# W \implies -L \in \text{lits-of } M \implies L' \in \# UW \implies L' \notin \# W \implies -L' \in \text{lits-of } M$ 
    using wf-c unfolding cw-eq by auto

  have unit-set: set-mset (W - mset-set (uminus ' lits-of M)) = {L}
proof
  show set-mset (W - mset-set (uminus ' lits-of M))  $\subseteq$  {L}
proof
  fix L'
  assume l': L'  $\in$  set-mset (W - mset-set (uminus ' lits-of M))
  hence l'-mem-w: L'  $\in$  set-mset W
    by auto
  have L'  $\notin$  uminus ' lits-of M
    using distinct-mem-diff-mset[OF w-nw(1) l'] by simp
  then have  $\neg M \models_a \{\# - L' \# \}$ 

```



```

    using image-iff by fastforce
  moreover have  $L' \in \# C$ 
    using cw(1) cw-eq l'-mem-w by auto
  ultimately have  $L' = L$ 
    unfolding M-def by (metis unsat[unfolded CNot-def true-annots-def, simplified])
  then show  $L' \in \{L\}$ 
    by simp
qed
next
show  $\{L\} \subseteq \text{set-mset } (W - \text{mset-set } (\text{uminus } \text{' lits-of } M))$ 
proof clarify
  have  $L \in \# W$ 
  proof (cases W)
    case empty
    thus ?thesis
      using w-nw(2) cw(1) cw-eq l-mem by auto
  next
    case (add W' La)
    thus ?thesis
      proof (cases La = L)
        case True
        thus ?thesis
          using add by simp
      next
        case False
        have  $-La \in \text{lits-of } M$ 
          using False add cw(1) cw-eq unsat[unfolded CNot-def true-annots-def, simplified]
          by fastforce
        then show ?thesis
          by (metis M-def Marked-Propagated-in-iff-in-lits-of add add.left-neutral count-union
            cw(1) cw-eq grOI l-mem twl-clause.sel(1) twl-clause.sel(2) undef union-single-eq-member
            w-nw(3))
      qed
    qed
  moreover have  $L \notin \# \text{mset-set } (\text{uminus } \text{' lits-of } M)$ 
    using Marked-Propagated-in-iff-in-lits-of undef by auto
  ultimately show  $L \in \text{set-mset } (W - \text{mset-set } (\text{uminus } \text{' lits-of } M))$ 
    by auto
  qed
qed
have unit:  $W - \text{mset-set } (\text{uminus } \text{' lits-of } M) = \{\#L\# \}$ 
  by (metis distinct-mset-minus distinct-mset-set-mset-ident distinct-mset-singleton
    set-mset-single unit-set w-nw(1))

show ?thesis
  unfolding candidates-propagate-def using unit undef cw cw-eq by fastforce
qed

lemma wf-candidates-conflict-sound:
  assumes wf: wf-twl-state S and
    cand:  $C \in \text{candidates-conflict } S$ 
  shows  $\text{trail } S \models_{\text{as}} \text{CNot } C \wedge C \in \# \text{image-mset raw-clause } (\text{clauses } S)$ 
proof
  def M  $\equiv \text{trail } S$ 
  def N  $\equiv \text{init-clss } S$ 

```

```

def U ≡ learned-clss S

note MNU-defs [simp] = M-def N-def U-def

obtain Cw where cw:
  C = raw-clause Cw
  Cw ∈# N + U
  watched Cw ⊆# mset-set (uminus ' lits-of (trail S))
  using cand[unfolding candidates-conflict-def, simplified] by auto

obtain W UW where cw-eq: Cw = TWL-Clause W UW
  by (case-tac Cw, blast)

have wf-c: wf-twl-clss M Cw
  using wf cw(2) unfolding wf-twl-state-def by simp

have w-nw:
  distinct-mset W
  size W < 2 ⇒ set-mset UW ⊆ set-mset W
  ∧ L L'. L ∈# W ⇒ -L ∈ lits-of M ⇒ L' ∈# UW ⇒ L' ∉# W ⇒ -L' ∈ lits-of M
  using wf-c unfolding cw-eq by auto

have ∀ L ∈# C. -L ∈ lits-of M
proof (cases W = {#})
  case True
  then have C = {#}
    using cw(1) cw-eq w-nw(2) by auto
  then show ?thesis
    by simp
next
  case False
  then obtain La where la: La ∈# W
    using multiset-eq-iff by force
  show ?thesis
  proof
    fix L
    assume l: L ∈# C
    show -L ∈ lits-of M
    proof (cases L ∈# W)
      case True
      thus ?thesis
        using cw(3) cw-eq by fastforce
    next
      case False
      thus ?thesis
        by (smt M-def l add-diff-cancel-left' count-diff cw(1) cw(3) la cw-eq
            diff-zero elem-mset-set finite-imageI finite-lits-of-def grOI imageE mset-leD
            uminus-of-uminus-id twl-clause.sel(1) twl-clause.sel(2) w-nw(3))
    qed
  qed
qed
then show trail S ⊨as CNot C
  unfolding CNot-def true-annots-def by auto

show C ∈# image-mset raw-clause (clauses S)

```

```

    using cw by auto
qed

lemma wf-candidates-conflict-complete:
  assumes wf: wf-twl-state S and
    c-mem: C ∈# raw-clauses S and
    unsat: trail S ⊨as CNot C
  shows C ∈ candidates-conflict S
proof -
  def M ≡ trail S
  def N ≡ init-clss S
  def U ≡ learned-clss S

  note MNU-defs [simp] = M-def N-def U-def

  obtain Cw where cw: C = raw-clause Cw Cw ∈# N + U
    using c-mem by force

  obtain W UW where cw-eq: Cw = TWL-Clause W UW
    by (case-tac Cw, blast)

  have wf-c: wf-twl-clss M Cw
    using wf cw(2) unfolding wf-twl-state-def by simp

  have w-nw:
    distinct-mset W
    size W < 2 ⟹ set-mset UW ⊆ set-mset W
    ∧ L L'. L ∈# W ⟹ -L ∈ lits-of M ⟹ L' ∈# UW ⟹ L' ∉# W ⟹ -L' ∈ lits-of M
    using wf-c unfolding cw-eq by auto

  have ∧L. L ∈# C ⟹ -L ∈ lits-of M
    unfolding M-def using unsat[unfolded CNot-def true-annots-def, simplified] by blast
  then have set-mset C ⊆ uminus ' lits-of M
    by (metis imageI mem-set-mset-iff subsetI uminus-of-uminus-id)
  then have set-mset W ⊆ uminus ' lits-of M
    using cw(1) cw-eq by auto
  then have subset: W ⊆# mset-set (uminus ' lits-of M)
    by (simp add: w-nw(1))

  have W = watched Cw
    using cw-eq twl-clause.sel(1) by simp
  then show ?thesis
    using MNU-defs cw(1) cw(2) subset candidates-conflict-def by blast
qed

typedef 'v wf-twl = {S::('v, nat, 'v clause) twl-state. wf-twl-state S}
morphisms rough-state-of-twl twl-of-rough-state
proof -
  have TWL-State ([::('v, nat, 'v clause) marked-lits)
    {#} {#} 0 C-True ∈ {S::('v, nat, 'v clause) twl-state. wf-twl-state S}
    by (auto simp: wf-twl-state-def)
  then show ?thesis by auto
qed

lemma [code abstype]:

```

twl-of-rough-state (*rough-state-of-twl* *S*) = *S*
by (*fact CDCL-Two-Watched-Literals.wf-twl.rough-state-of-twl-inverse*)

lemma *wf-twl-state-rough-state-of-twl[simp]*: *wf-twl-state* (*rough-state-of-twl* *S*)
using *rough-state-of-twl* **by** *auto*

abbreviation *candidates-conflict-twl* :: '*v* *wf-twl* \Rightarrow '*v* literal multiset set **where**
candidates-conflict-twl *S* \equiv *candidates-conflict* (*rough-state-of-twl* *S*)

abbreviation *candidates-propagate-twl* :: '*v* *wf-twl* \Rightarrow ('*v* literal \times '*v* clause) set **where**
candidates-propagate-twl *S* \equiv *candidates-propagate* (*rough-state-of-twl* *S*)

abbreviation *trail-twl* :: '*a* *wf-twl* \Rightarrow ('*a*, nat, '*a* literal multiset) marked-lit list **where**
trail-twl *S* \equiv *trail* (*rough-state-of-twl* *S*)

abbreviation *clauses-twl* :: '*a* *wf-twl* \Rightarrow '*a* literal multiset multiset **where**
clauses-twl *S* \equiv *raw-clauses* (*rough-state-of-twl* *S*)

abbreviation *init-clss-twl* :: '*a* *wf-twl* \Rightarrow '*a* literal multiset multiset **where**
init-clss-twl *S* \equiv *raw-init-clss* (*rough-state-of-twl* *S*)

abbreviation *learned-clss-twl* :: '*a* *wf-twl* \Rightarrow '*a* literal multiset multiset **where**
learned-clss-twl *S* \equiv *raw-learned-clss* (*rough-state-of-twl* *S*)

abbreviation *backtrack-lvl-twl* **where**
backtrack-lvl-twl *S* \equiv *backtrack-lvl* (*rough-state-of-twl* *S*)

abbreviation *conflicting-twl* **where**
conflicting-twl *S* \equiv *conflicting* (*rough-state-of-twl* *S*)

lemma *wf-candidates-twl-conflict-complete*:
assumes
c-mem: *C* $\in \#$ *clauses-twl* *S* **and**
unsat: *trail-twl* *S* \models_{as} *CNot* *C*
shows *C* \in *candidates-conflict-twl* *S*
using *c-mem* *unsat* *wf-candidates-conflict-complete* *wf-twl-state-rough-state-of-twl* **by** *blast*

21.3 Abstract 2-WL

locale *abstract-twl* =

fixes

watch :: ('*v*, nat, '*v* clause) *twl-state* \Rightarrow '*v* clause \Rightarrow '*v* *twl-clause* **and**
rewatch :: ('*v*, nat, '*v* literal multiset) marked-lit \Rightarrow ('*v*, nat, '*v* clause) *twl-state* \Rightarrow
'*v* *twl-clause* \Rightarrow '*v* *twl-clause* **and**
linearize :: '*v* clauses \Rightarrow '*v* clause list **and**
restart-learned :: ('*v*, nat, '*v* clause) *twl-state* \Rightarrow '*v* *twl-clause* multiset

assumes

clause-watch: *no-dup*(*trail* *S*) \implies *raw-clause* (*watch* *S* *C*) = *C* **and**
wf-watch: *no-dup* (*trail* *S*) \implies *wf-twl-cl*s (*trail* *S*) (*watch* *S* *C*) **and**
clause-rewatch: *raw-clause* (*rewatch* *L* *S* *C'*) = *raw-clause* *C'* **and**
wf-rewatch:
no-dup (*trail* *S*) \implies *undefined-lit* (*trail* *S*) (*lit-of* *L*) \implies *wf-twl-cl*s (*trail* *S*) *C'* \implies
*wf-twl-cl*s (*L* $\#$ *trail* *S*) (*rewatch* *L* *S* *C'*)

and

linearize: *mset* (*linearize* *N*) = *N* **and**
restart-learned: *restart-learned* *S* $\subseteq \#$ *learned-clss* *S*

begin

lemma *linearize-mempty[simp]*: $\text{linearize } \{\#\} = []$
using *linearize mset-zero-iff* **by** *blast*

definition

$\text{cons-trail} :: ('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow$
 $('v, \text{nat}, 'v \text{ clause}) \text{ twl-state}$

where

$\text{cons-trail } L \ S =$
 $\text{TWL-State } (L \ \# \ \text{trail } S) \ (\text{image-mset } (\text{rewatch } L \ S) \ (\text{init-clss } S))$
 $(\text{image-mset } (\text{rewatch } L \ S) \ (\text{learned-clss } S)) \ (\text{backtrack-lvl } S) \ (\text{conflicting } S)$

definition

$\text{add-init-cls} :: 'v \text{ clause} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow$
 $('v, \text{nat}, 'v \text{ clause}) \text{ twl-state}$

where

$\text{add-init-cls } C \ S =$
 $\text{TWL-State } (\text{trail } S) \ (\{\# \text{watch } S \ C \# \} + \text{init-clss } S) \ (\text{learned-clss } S) \ (\text{backtrack-lvl } S)$
 $(\text{conflicting } S)$

definition

$\text{add-learned-cls} :: 'v \text{ clause} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow$
 $('v, \text{nat}, 'v \text{ clause}) \text{ twl-state}$

where

$\text{add-learned-cls } C \ S =$
 $\text{TWL-State } (\text{trail } S) \ (\text{init-clss } S) \ (\{\# \text{watch } S \ C \# \} + \text{learned-clss } S) \ (\text{backtrack-lvl } S)$
 $(\text{conflicting } S)$

definition

$\text{remove-cls} :: 'v \text{ clause} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state}$

where

$\text{remove-cls } C \ S =$
 $\text{TWL-State } (\text{trail } S) \ (\text{filter-mset } (\lambda D. \text{raw-clause } D \neq C) \ (\text{init-clss } S))$
 $(\text{filter-mset } (\lambda D. \text{raw-clause } D \neq C) \ (\text{learned-clss } S)) \ (\text{backtrack-lvl } S)$
 $(\text{conflicting } S)$

definition $\text{init-state} :: 'v \text{ clauses} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ twl-state}$ **where**

$\text{init-state } N = \text{fold add-init-cls } (\text{linearize } N) \ (\text{TWL-State } [] \ \{\#\} \ \{\#\} \ 0 \ C\text{-True})$

lemma *unchanged-fold-add-init-cls*:

$\text{trail } (\text{fold add-init-cls } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = M$
 $\text{learned-clss } (\text{fold add-init-cls } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = U$
 $\text{backtrack-lvl } (\text{fold add-init-cls } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = k$
 $\text{conflicting } (\text{fold add-init-cls } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = C$
by $(\text{induct } Cs \ \text{arbitrary: } N) \ (\text{auto simp: add-init-cls-def})$

lemma *unchanged-init-state[simp]*:

$\text{trail } (\text{init-state } N) = []$
 $\text{learned-clss } (\text{init-state } N) = \{\#\}$
 $\text{backtrack-lvl } (\text{init-state } N) = 0$
 $\text{conflicting } (\text{init-state } N) = C\text{-True}$
unfolding *init-state-def* **by** $(\text{rule unchanged-fold-add-init-cls}) +$

lemma *clauses-init-fold-add-init*:

$no\text{-}dup\ M \implies$
 $image\text{-}mset\ raw\text{-}clause\ (init\text{-}class\ (fold\ add\text{-}init\text{-}cls\ Cs\ (TWL\text{-}State\ M\ N\ U\ k\ C))) =$
 $mset\ Cs + image\text{-}mset\ raw\text{-}clause\ N$
by (induct Cs arbitrary: N) (auto simp: add.assoc add-init-cls-def clause-watch)

lemma $init\text{-}class\text{-}init\text{-}state[simp]$: $image\text{-}mset\ raw\text{-}clause\ (init\text{-}class\ (init\text{-}state\ N)) = N$
unfolding $init\text{-}state\text{-}def$ **by** (simp add: clauses-init-fold-add-init linearize)

definition $update\text{-}backtrack\text{-}lvl$ **where**
 $update\text{-}backtrack\text{-}lvl\ k\ S =$
 $TWL\text{-}State\ (trail\ S)\ (init\text{-}class\ S)\ (learned\text{-}class\ S)\ k\ (conflicting\ S)$

definition $update\text{-}conflicting$ **where**
 $update\text{-}conflicting\ C\ S = TWL\text{-}State\ (trail\ S)\ (init\text{-}class\ S)\ (learned\text{-}class\ S)\ (backtrack\text{-}lvl\ S)\ C$

definition $tl\text{-}trail$ **where**
 $tl\text{-}trail\ S =$
 $TWL\text{-}State\ (tl\ (trail\ S))\ (init\text{-}class\ S)\ (learned\text{-}class\ S)\ (backtrack\text{-}lvl\ S)\ (conflicting\ S)$

definition $restart'$ **where**
 $restart'\ S = TWL\text{-}State\ []\ (init\text{-}class\ S)\ (restart\text{-}learned\ S)\ 0\ C\text{-}True$
end

21.4 Instantiation of the previous locale

definition $pull :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $pull\ p\ xs = filter\ p\ xs\ @\ filter\ (Not\ o\ p)\ xs$

lemma $set\text{-}pull[simp]$: $set\ (pull\ p\ xs) = set\ xs$
unfolding $pull\text{-}def$ **by** auto

lemma $mset\text{-}pull[simp]$: $mset\ (pull\ p\ xs) = mset\ xs$
by (simp add: pull-def mset-filter-compl)

lemma $mset\text{-}take\text{-}pull\text{-}sorted\text{-}list\text{-}of\text{-}set\text{-}subseq$:
 $mset\ (take\ n\ (pull\ p\ (sorted\text{-}list\text{-}of\text{-}set\ (set\text{-}mset\ A)))) \subseteq\# A$
by (metis $mset\text{-}pull\ mset\text{-}set\text{-}mset\text{-}subseq\ mset\text{-}sorted\text{-}list\text{-}of\text{-}set\ mset\text{-}take\text{-}subseq$
 $subset\text{-}mset.dual\text{-}order.trans$)

definition $watch\text{-}nat :: (nat, nat, nat\ clause)\ twl\text{-}state \Rightarrow nat\ clause \Rightarrow nat\ twl\text{-}clause$ **where**
 $watch\text{-}nat\ S\ C =$
 $(let$
 $\quad C' = remdups\ (sorted\text{-}list\text{-}of\text{-}set\ (set\text{-}mset\ C));$
 $\quad negation\text{-}not\text{-}assigned = filter\ (\lambda L. \neg L \in\# lits\text{-}of\ (trail\ S))\ C';$
 $\quad negation\text{-}assigned\text{-}sorted\text{-}by\text{-}trail = filter\ (\lambda L. L \in\# C')\ (map\ (\lambda L. \neg lit\text{-}of\ L)\ (trail\ S));$
 $\quad W = take\ 2\ (negation\text{-}not\text{-}assigned\ @\ negation\text{-}assigned\text{-}sorted\text{-}by\text{-}trail);$
 $\quad UW = sorted\text{-}list\text{-}of\text{-}multiset\ (C - mset\ W)$
 $\quad in\ TWL\text{-}Clause\ (mset\ W)\ (mset\ UW))$

lemma $list\text{-}cases2$:
fixes $l :: 'a\ list$
assumes
 $l = [] \implies P$ **and**
 $\bigwedge x. l = [x] \implies P$ **and**
 $\bigwedge x\ y\ xs. l = x \# y \# xs \implies P$
shows P

by (metis assms list.collapse)

lemma *filter-in-list-prop-verifiedD*:

assumes $[L \leftarrow P \ . \ Q \ L] = l$
shows $\forall x \in \text{set } l. x \in \text{set } P \wedge Q \ x$
using *assms* **by** *auto*

lemma *no-dup-filter-diff*:

assumes *n-d*: *no-dup* *M* **and** *H*: $[L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) \ M. L \in \# \ C] = l$
shows *distinct* *l*
unfolding $H[\text{symmetric}]$
apply (rule *distinct-filter*)
using *n-d* **by** (induction *M*) *auto*

lemma *watch-nat-lists-disjointD*:

assumes
 $l: [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \ . \ - \ L \notin \text{lits-of } (\text{trail } S)] = l$ **and**
 $l': [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) \ . \ L \in \# \ C] = l'$
shows $\forall x \in \text{set } l. \forall y \in \text{set } l'. x \neq y$
by (auto simp: $l[\text{symmetric}] \ l'[\text{symmetric}] \ \text{lits-of-def}$)

lemma *watch-nat-list-cases* [*consumes 1*, *case-names nil-nil nil-single nil-other single-nil single-other other*]:

fixes *C* :: '*v*::linorder literal multiset **and** *S* :: ('*v*, '*a*, '*b*) twl-state

defines

$xs \equiv [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \ . \ - \ L \notin \text{lits-of } (\text{trail } S)]$ **and**
 $ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) \ . \ L \in \# \ C]$

assumes *n-d*: *no-dup* (*trail* *S*) **and**

nil-nil: $xs = [] \implies ys = [] \implies P$ **and**

nil-single:

$\bigwedge a. xs = [] \implies ys = [a] \implies a \in \# \ C \implies P$ **and**

nil-other: $\bigwedge a \ b \ ys'. xs = [] \implies ys = a \ \# \ b \ \# \ ys' \implies a \neq b \implies P$ **and**

single-nil: $\bigwedge a. xs = [a] \implies ys = [] \implies P$ **and**

single-other: $\bigwedge a \ b \ ys'. xs = [a] \implies ys = b \ \# \ ys' \implies a \neq b \implies P$ **and**

other: $\bigwedge a \ b \ xs'. xs = a \ \# \ b \ \# \ xs' \implies a \neq b \implies P$

shows *P*

proof –

note *xs-def*[*simp*] **and** *ys-def*[*simp*]

have *dist*: *distinct* $[L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \ . \ - \ L \notin \text{lits-of } (\text{trail } S)]$

by *auto*

then have *H*: $\bigwedge a \ xs. [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \ . \ - \ L \notin \text{lits-of } (\text{trail } S)]$
 $\neq a \ \# \ a \ \# \ xs$

by *force*

show *?thesis*

apply (cases $[L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) \ . \ - \ L \notin \text{lits-of } (\text{trail } S)]$

rule: *list-cases2*;

cases $[L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) \ . \ L \in \# \ C]$ rule: *list-cases2*)

using *nil-nil* **apply** *simp*

using *nil-single* **apply** (force dest: *filter-in-list-prop-verifiedD*)

using *nil-other*

apply (auto dest: *filter-in-list-prop-verifiedD* *watch-nat-lists-disjointD*

no-dup-filter-diff[*OF* *n-d*] *simp*: *H*)[]

using *single-nil* **apply** *simp*

using *single-other*

apply (auto dest: *filter-in-list-prop-verifiedD* *watch-nat-lists-disjointD*

$\text{no-dup-filter-diff}[OF\ n-d]\ \text{simp: } H)\square$
using *single-other* **apply** (*auto dest: filter-in-list-prop-verifiedD watch-nat-lists-disjointD*
 $\text{no-dup-filter-diff}[OF\ n-d]\ \text{simp: } H)\square$
using *other xs-def ys-def* **by** (*metis H*)
qed

lemma *watch-nat-lists-set-union:*

fixes $C :: 'v::\text{linorder literal multiset}$ **and** $S :: ('v, 'a, 'b)\ \text{twl-state}$
defines
 $xs \equiv [L \leftarrow \text{remdups}(\text{sorted-list-of-set}(\text{set-mset } C)) \ . \ -\ L \notin \text{lits-of}(\text{trail } S)]$ **and**
 $ys \equiv [L \leftarrow \text{map}(\lambda L. \ -\ \text{lit-of } L)(\text{trail } S) \ . \ L \in\# C]$
assumes $n-d: \text{no-dup}(\text{trail } S)$
shows $\text{set-mset } C = \text{set } xs \cup \text{set } ys$
using $n-d$ **unfolding** *xs-def ys-def* **by** (*auto simp: lits-of-def uminus-lit-swap*)

definition

rewatch-nat ::
 $(\text{nat}, \text{nat}, \text{nat literal multiset})\ \text{marked-lit} \Rightarrow (\text{nat}, \text{nat}, \text{nat clause})\ \text{twl-state} \Rightarrow$
 $\text{nat twl-clause} \Rightarrow \text{nat twl-clause}$

where

$\text{rewatch-nat } L\ S\ C =$
 $(\text{if } -\ \text{lit-of } L \in\# \text{watched } C \text{ then}$
 $\text{case filter } (\lambda L'.\ L' \notin\# \text{watched } C \wedge -\ L' \notin \text{lits-of}(L \# \text{trail } S))$
 $(\text{sorted-list-of-multiset}(\text{unwatched } C))\ \text{of}$
 $\square \Rightarrow C$
 $| L' \# - \Rightarrow$
 $\text{TWL-Clause}(\text{watched } C - \{\# -\ \text{lit-of } L\# \} + \{\# L'\# \})(\text{unwatched } C - \{\# L'\# \} + \{\# -\ \text{lit-of}$
 $L\#\})$
 else
 $C)$

lemma *mset-intersection-inclusion:* $A + (B - A) = B \longleftrightarrow A \subseteq\# B$

apply (*rule iffI*)
apply (*metis mset-le-add-left*)
by (*auto simp: ac-simps multiset-eq-iff subseteq-mset-def*)

lemma *clause-watch-nat:*

assumes $\text{no-dup}(\text{trail } S)$
shows $\text{raw-clause}(\text{watch-nat } S\ C) = C$
using *assms*
apply (*cases rule: watch-nat-list-cases[OF assms(1), of C]*)
by (*auto dest: filter-in-list-prop-verifiedD simp: watch-nat-def Let-def*
 $\text{mset-intersection-inclusion subseteq-mset-def}$)

lemma *distinct-pull[simp]:* $\text{distinct}(\text{pull } p\ xs) = \text{distinct } xs$

unfolding *pull-def* **by** (*induct xs*) *auto*

lemma *falsified-watched-imp-unwatched-falsified:*

assumes
 $\text{watched: } L \in \text{set}(\text{take } n(\text{pull}(\text{Not} \circ \text{fls})(\text{sorted-list-of-set}(\text{set-mset } C))))$ **and**
 $\text{falsified: fls } L$ **and**
 $\text{not-watched: } L' \notin \text{set}(\text{take } n(\text{pull}(\text{Not} \circ \text{fls})(\text{sorted-list-of-set}(\text{set-mset } C))))$ **and**
 $\text{unwatched: } L' \in\# C - \text{mset}(\text{take } n(\text{pull}(\text{Not} \circ \text{fls})(\text{sorted-list-of-set}(\text{set-mset } C))))$
shows $\text{fls } L'$

proof —


```

let ?Ls = sorted-list-of-set (set-mset C)
let ?W = take n (pull (Not ∘ fls) ?Ls)

have n > length (filter (Not ∘ fls) ?Ls)
  using watched falsified
  unfolding pull-def comp-def
  apply auto
  using in-set-takeD apply fastforce
  by (metis grOI length-greater-0-conv length-pos-if-in-set take-0 zero-less-diff)
then have  $\bigwedge L. L \in \text{set } ?Ls \implies \neg \text{fls } L \implies L \in \text{set } ?W$ 
  unfolding pull-def by auto
then show ?thesis
  by (metis Multiset.diff-le-self finite-set-mset mem-set-mset-iff mset-leD not-watched
        sorted-list-of-set unwatched)
qed

lemma set-mset-is-single-in-mset-is-single:
  set-mset C = {a}  $\implies x \in \# C \implies x = a$ 
  by fastforce

lemma index-uminus-index-map-uminus:
   $-a \in \text{set } L \implies \text{index } L (-a) = \text{index } (\text{map } \text{uminus } L) (a::'a \text{ literal})$ 
  by (induction L) auto

lemma index-filter:
   $a \in \text{set } L \implies b \in \text{set } L \implies P a \implies P b \implies$ 
   $\text{index } L a \leq \text{index } L b \iff \text{index } (\text{filter } P L) a \leq \text{index } (\text{filter } P L) b$ 
  by (induction L) auto

lemma wf-watch-nat: no-dup (trail S)  $\implies$  wf-twl-cls (trail S) (watch-nat S C)
  apply (simp only: watch-nat-def Let-def partition-filter-conv case-prod-beta fst-conv snd-conv)
  unfolding wf-twl-cls.simps
  apply (intro conjI)
proof goal-cases
  case 1
  then show ?case
    by (cases rule: watch-nat-list-cases[of S C]) (auto dest: filter-in-list-prop-verifiedD
        simp: distinct-mset-add-single)
next
  case 2
  then show ?case by simp
next
  case 3
  then show ?case
    proof (cases rule: watch-nat-list-cases[of S C])
    case nil-nil
    then have set-mset C = set []  $\cup$  set []
      using 3 by (metis watch-nat-lists-set-union)
    then show ?thesis
      by simp
    next
    case nil-single
    then show ?thesis
      using watch-nat-lists-set-union[of S C] 3 by (auto dest!: arg-cong[of - [] set])
    next

```

```

    case nil-other
    then show ?thesis
      using 3 by (auto dest!: arg-cong[of - [] set])
next
  case single-nil
  show ?thesis
    using watch-nat-lists-set-union[of S C] 3 mset-leD unfolding single-nil by auto
next
  case single-other
  then show ?thesis
    using 3 by (auto dest!: arg-cong[of - [] set])
next
  case other
  then show ?thesis
    using 3 by (auto dest!: arg-cong[of - [] set])[]
qed
next
case 4 note -[simp] = this
{
  fix a :: nat literal and ys' :: nat literal list and L :: nat literal and
    L' :: nat literal
  assume a1: [L←remdups (insort L (sorted-list-of-set (insert a (set ys') - {L})))
    - L ∉ lits-of (trail S)] = [a]
  assume a2: set-mset C = insert L (insert a (set ys'))
  assume a3: L' ∈# C
  assume a4: a ≠ L'
  have set (L # a # ys') = set-mset C
    using a2 by auto
  then have L' ∉ set [l←remdups (sorted-list-of-set (set-mset C)) . - l ∉ lits-of (trail S)]
    using a4 a1 by (metis List.finite-set list.set(1) list.set(2) singleton-iff
      sorted-list-of-set.insert-remove)
  then have - L' ∈ lits-of (trail S)
    using a3 by simp
} note H = this
show ?case using 4
  apply (cases rule: watch-nat-list-cases[of S C])
  apply (auto dest: filter-in-list-prop-verifiedD H simp: filter-empty-conv)[3]
  using watch-nat-lists-set-union[of S C] by (auto dest: filter-in-list-prop-verifiedD H)
next
case 5
then show ?case
  proof (cases rule: watch-nat-list-cases[of S C])
    case nil-nil
    then show ?thesis by auto
  next
    case nil-single
    then show ?thesis
      using watch-nat-lists-set-union[of S C] 5 by auto
  next
    case nil-other
    then show ?thesis
      unfolding watched-decided-most-recently.simps Ball-mset-def
      apply (intro allI impI)
      apply (subst index-uminus-index-map-uminus,
        simp add: index-uminus-index-map-uminus lits-of-def o-def)

```

```

    apply (subst index-uminus-index-map-uminus,
           simp add: index-uminus-index-map-uminus lits-of-def o-def)

    apply (subst index-filter[of - - λL. L ∈# C])
    by (auto dest: filter-in-list-prop-verifiedD
         simp: uminus-lit-swap lits-of-def o-def)
  next
  case single-nil
  then show ?thesis
    using watch-nat-lists-set-union[of S C] 5 by auto
  next
  case single-other
  then show ?thesis
    unfolding watched-decided-most-recently.simps Ball-mset-def
    apply (clarify)
    apply (subst index-uminus-index-map-uminus,
           simp add: index-uminus-index-map-uminus lits-of-def o-def)
    apply (subst index-uminus-index-map-uminus,
           simp add: index-uminus-index-map-uminus lits-of-def o-def)

    apply (subst index-filter[of - - λL. L ∈# C])
    by (auto dest: filter-in-list-prop-verifiedD simp: uminus-lit-swap lits-of-def o-def)
  next
  case other
  then show ?thesis
    apply clarsimp
    apply (elim disjE)
    prefer 2 apply (auto dest: filter-in-list-prop-verifiedD)[]
    apply (subst index-uminus-index-map-uminus,
           simp add: index-uminus-index-map-uminus lits-of-def o-def)[1]
    apply (subst index-uminus-index-map-uminus,
           simp add: index-uminus-index-map-uminus lits-of-def o-def)[1]

    apply (subst index-filter[of - - λL. L ∈# C])
    by (auto dest: filter-in-list-prop-verifiedD
         simp: index-uminus-index-map-uminus lits-of-def o-def uminus-lit-swap)
  qed
qed

```

```

lemma filter-sorted-list-of-multiset-eqD:
  assumes [x ← sorted-list-of-multiset A. p x] = x # xs (is ?comp = -)
  shows x ∈# A
proof -
  have x ∈ set ?comp
  using assms by simp
  then have x ∈ set (sorted-list-of-multiset A)
  by simp
  then show x ∈# A
  by simp
qed

```

```

lemma clause-rewatch-nat: raw-clause (rewatch-nat L S C) = raw-clause C
  apply (auto simp: rewatch-nat-def Let-def split: list.split)
  apply (subst subset-mset.add-diff-assoc2, simp)
  apply (subst subset-mset.add-diff-assoc2, simp)

```

apply (*subst subset-mset.add-diff-assoc2*)
apply (*auto dest: filter-sorted-list-of-multiset-eqD*)
by (*metis (no-types, lifting) add.assoc add-diff-cancel-right' filter-sorted-list-of-multiset-eqD insert-DiffM mset-leD mset-le-add-left*)

lemma *filter-sorted-list-of-multiset-Nil*:

$[x \leftarrow \text{sorted-list-of-multiset } M. p \ x] = [] \longleftrightarrow (\forall x \in \# M. \neg p \ x)$
by *auto (metis empty-iff filter-set list.set(1) mem-set-mset-iff member-filter set-sorted-list-of-multiset)*

lemma *filter-sorted-list-of-multiset-ConsD*:

$[x \leftarrow \text{sorted-list-of-multiset } M. p \ x] = x \ \# \ xs \implies p \ x$
by (*metis filter-set insert-iff list.set(2) member-filter*)

lemma *mset-minus-single-eq-mempty*:

$a - \{\#b\} = \{\#\} \longleftrightarrow a = \{\#b\} \vee a = \{\#\}$
by (*metis Multiset.diff-cancel add.right-neutral diff-single-eq-union diff-single-trivial zero-diff*)

lemma *size-mset-le-2-cases*:

assumes $size \ W \leq 2$
shows $W = \{\#\} \vee (\exists a. W = \{\#a\}) \vee (\exists a \ b. W = \{\#a, b\})$
by (*metis One-nat-def Suc-1 Suc-eq-plus1-left assms linorder-not-less nat-less-le not-less-eq-eq ordered-cancel-comm-monoid-diff-class.le-iff-add size-1-singleton-mset size-eq-0-iff-empty size-mset-2*)

lemma *wf-rewatch-nat'*:

assumes
wf: *wf-twl-cl*s (*trail S*) *C* **and**
n-d: *no-dup* (*trail S*) **and**
undef: *undefined-lit* (*trail S*) (*lit-of L*)
shows *wf-twl-cl*s (*L* *#* *trail S*) (*rewatch-nat L S C*)
using *filter-sorted-list-of-multiset-Nil[simp]*
proof (*cases - lit-of L ∈ # watched C*)
case falsified: *True*

let *?unwatched-nonfalsified* =

$[L' \leftarrow \text{sorted-list-of-multiset } (\text{unwatched } C) . L' \notin \# \text{ watched } C \wedge - L' \notin \text{ lits-of } (L \ \# \ \text{trail } S)]$
obtain *W UW* **where** *C*: *C* = *TWL-Clause W UW*
by (*cases C*)

show *?thesis*

proof (*cases ?unwatched-nonfalsified*)

case *Nil*

show *?thesis*

unfolding *rewatch-nat-def*

using *falsified Nil*

apply (*simp only: wf-twl-cl.simps if-True list.cases C*)

apply (*intro conjI*)

proof *goal-cases*

case *1*

then show *?case* **using** *wf C* **by** *simp*

next

case *2*

then show *?case* **using** *wf C* **by** *simp*

```

next
  case 3
  then show ?case using wf C by simp
next
  case 4
  then show ?case using wf C by auto
next
  case 5
  then show ?case
    using C apply simp
    using wf by (smt ball-msetI bspec-mset not-gr0 uminus-of-uminus-id
      watched-decided-most-recently.simps wf-twl-cls.simps)
qed
next
case (Cons L' Ls)
show ?thesis
  unfolding rewatch-nat-def C
  using falsified Cons
  apply (simp only: wf-twl-cls.simps if-True list.cases C)
  apply (intro conjI)
  proof goal-cases
    case 1
    then show ?case using wf C n-d
      by (smt Multiset.diff-le-self distinct-mset-add-single distinct-mset-single-add
        filter-sorted-list-of-multiset-ConsD insert-DiffM mset-leD twl-clause.sel(1)
        wf-twl-cls.simps)
  next
    case 2
    then show ?case using wf C by (metis insert-DiffM2 size-single size-union twl-clause.sel(1)
      wf-twl-cls.simps)
  next
    case 3
    then show ?case
      using wf C by (force simp: mset-minus-single-eq-mempty dest: subset-singletonD)
  next
    case 4
    have H:  $\forall L \in \#W. - L \in \text{lits-of } (\text{trail } S) \longrightarrow$ 
      ( $\forall L' \in \#UW. \text{count } W L' = 0 \longrightarrow - L' \in \text{lits-of } (\text{trail } S)$ )
      using wf by (auto simp: C)
    have W:  $\text{size } W \leq 2$  and W-UW:  $\text{size } W < 2 \longrightarrow \text{set-mset } UW \subseteq \text{set-mset } W$ 
      using wf by (auto simp: C)

    have distinct: distinct-mset W
      using wf by (auto simp: C)
    show ?case
      using 4
      unfolding C watched-decided-most-recently.simps Ball-mset-def twl-clause.sel
      apply (intro allI impI)
      apply (rename-tac xW xUW)
      apply (case-tac - lit-of L = xW; case-tac xW = xUW; case-tac L' = xW)
        apply (auto simp: uminus-lit-swap)[2]
        using filter-sorted-list-of-multiset-ConsD apply blast
      using H size-mset-le-2-cases[OF W]
      using distinct apply (fastforce split: split-if-asm simp: distinct-mset-size-2)
      using distinct apply (fastforce split: split-if-asm simp: distinct-mset-size-2)

```

```

    using distinct apply (fastforce split: split-if-asm simp: distinct-mset-size-2)
    using filter-sorted-list-of-multiset-ConsD apply blast
    using size-mset-le-2-cases[OF W] H by (fastforce simp: uminus-lit-swap
    dest: filter-sorted-list-of-multiset-ConsD filter-sorted-list-of-multiset-eqD)

next
  case 5
  have H:  $\forall x. x \in \# W \longrightarrow \neg x \in \text{lits-of } (\text{trail } S) \longrightarrow (\forall x. x \in \# UW \longrightarrow \text{count } W x = 0 \longrightarrow \neg x \in \text{lits-of } (\text{trail } S))$ 
    using wf by (auto simp: C)

  show ?case
    using 5 unfolding C watched-decided-most-recently.simps Ball-mset-def
    apply (intro allI impI conjI)
    apply (rename-tac xW x)
    apply (case-tac  $\neg \text{lit-of } L = xW$ ; case-tac  $xW = x$ )
      apply (auto simp: uminus-lit-swap)[3]
    apply (case-tac  $\neg \text{lit-of } L = x$ )
    apply (clarsimp)
    using H apply (blast dest: filter-sorted-list-of-multiset-ConsD
    filter-sorted-list-of-multiset-eqD)
    apply (clarsimp)
    using H apply (blast dest: filter-sorted-list-of-multiset-ConsD
    filter-sorted-list-of-multiset-eqD)
  done
qed
qed
next
  case False
  then have wf-twlc (L # trail S) C
    apply (cases C)
    using wf n-d undef apply (clarify)
    unfolding wf-twlc.simps
    apply (intro conjI)
      apply blast
      apply blast
      apply blast
    apply (smt ball-mset-cong bspec-mset insert-iff lits-of-cons nat-neq-iff twl-clause.sel(1)
    uminus-of-uminus-id)
    apply (auto simp: Marked-Propagated-in-iff-in-lits-of)
  done
  then show ?thesis
    unfolding rewatch-nat-def using False by simp
qed

```

```

interpretation twl: abstract-twlc watch-nat rewatch-nat sorted-list-of-multiset learned-clss
  apply unfold-locales
  apply (rule clause-watch-nat; simp)
  apply (rule wf-watch-nat; simp)
  apply (rule clause-rewatch-nat)
  apply (rule wf-rewatch-nat'; simp)
  apply (rule mset-sorted-list-of-multiset)
  apply (rule subset-mset.order-refl)
done

```

21.5 Interpretation for $cdcl_W\text{-ops.cdcl}_W$

context *abstract-twl*
begin

21.5.1 Direct Interpretation

interpretation *rough-cdcl*: $state_W$ *trail* *raw-init-clss* *raw-learned-clss* *backtrack-lvl* *conflicting*
cons-trail *tl-trail* *add-init-cl* *add-learned-cl* *remove-cl* *update-backtrack-lvl*
update-conflicting *init-state* *restart'*
apply *unfold-locales*
apply (*simp-all* *add*: *add-init-cl-def* *add-learned-cl-def* *clause-rewatch* *clause-watch*
cons-trail-def *remove-cl-def* *restart'-def* *tl-trail-def* *update-backtrack-lvl-def*
update-conflicting-def)
apply (*rule* *image-mset-subseteq-mono*[*OF restart-learned*])
done

interpretation *rough-cdcl*: $cdcl_W\text{-ops}$ *trail* *raw-init-clss* *raw-learned-clss* *backtrack-lvl* *conflicting*
cons-trail *tl-trail* *add-init-cl* *add-learned-cl* *remove-cl* *update-backtrack-lvl*
update-conflicting *init-state* *restart'*
by *unfold-locales*

interpretation $cdcl_{NOT}$: $cdcl_{NOT}\text{-merge-bj-learn-ops}$
 $\lambda S.$ *convert-trail-from-W* (*trail* *S*)
rough-cdcl.clauses
 $\lambda L S.$ *cons-trail* (*convert-marked-lit-from-NOT* *L*) *S*
 $\lambda S.$ *tl-trail* *S*
 $\lambda C S.$ *add-learned-cl* *C* *S*
 $\lambda C S.$ *remove-cl* *C* *S*
 $\lambda L S.$ *lit-of* $L \in \text{fst } \text{'candidates-propagate } S$
 $\lambda S.$ *conflicting* $S = C\text{-True}$
 $\lambda C C' L' S.$ $C \in \text{candidates-conflict } S \wedge \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$
by *unfold-locales*

21.5.2 Opaque Type with Invariant

declare *rough-cdcl.state-simp*[*simp del*]

definition *cons-trail-twl* :: (*'v*, *nat*, *'v literal multiset*) *marked-lit* \Rightarrow *'v wf-twl* \Rightarrow *'v wf-twl*
where
cons-trail-twl *L* *S* \equiv *twl-of-rough-state* (*cons-trail* *L* (*rough-state-of-twl* *S*))

lemma *wf-twl-state-cons-trail*:
undefined-lit (*trail* *S*) (*lit-of* *L*) \implies *wf-twl-state* *S* \implies *wf-twl-state* (*cons-trail* *L* *S*)
unfolding *wf-twl-state-def* **by** (*auto simp*: *cons-trail-def* *wf-rewatch* *defined-lit-map*)

lemma *rough-state-of-twl-cons-trail*:
undefined-lit (*trail-twl* *S*) (*lit-of* *L*) \implies
rough-state-of-twl (*cons-trail-twl* *L* *S*) = *cons-trail* *L* (*rough-state-of-twl* *S*)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-cons-trail*
unfolding *cons-trail-twl-def* **by** *blast*

abbreviation *add-init-cl-twl* **where**
add-init-cl-twl *C* *S* \equiv *twl-of-rough-state* (*add-init-cl* *C* (*rough-state-of-twl* *S*))

lemma *wf-twl-add-init-cl*: *wf-twl-state* *S* \implies *wf-twl-state* (*add-init-cl* *L* *S*)

unfolding *wf-twl-state-def* **by** (*auto simp: wf-watch add-init-cls-def split: split-if-asm*)

lemma *rough-state-of-twl-add-init-cls:*
rough-state-of-twl (add-init-cls-twl L S) = add-init-cls L (rough-state-of-twl S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-init-cls* **by** *blast*

abbreviation *add-learned-cls-twl* **where**
add-learned-cls-twl C S \equiv twl-of-rough-state (add-learned-cls C (rough-state-of-twl S))

lemma *wf-twl-add-learned-cls: wf-twl-state S \implies wf-twl-state (add-learned-cls L S)*
unfolding *wf-twl-state-def* **by** (*auto simp: wf-watch add-learned-cls-def split: split-if-asm*)

lemma *rough-state-of-twl-add-learned-cls:*
rough-state-of-twl (add-learned-cls-twl L S) = add-learned-cls L (rough-state-of-twl S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-learned-cls* **by** *blast*

abbreviation *remove-cls-twl* **where**
remove-cls-twl C S \equiv twl-of-rough-state (remove-cls C (rough-state-of-twl S))

lemma *wf-twl-remove-cls: wf-twl-state S \implies wf-twl-state (remove-cls L S)*
unfolding *wf-twl-state-def* **by** (*auto simp: wf-watch remove-cls-def split: split-if-asm*)

lemma *rough-state-of-twl-remove-cls:*
rough-state-of-twl (remove-cls-twl L S) = remove-cls L (rough-state-of-twl S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-remove-cls* **by** *blast*

abbreviation *init-state-twl* **where**
init-state-twl N \equiv twl-of-rough-state (init-state N)

lemma *wf-twl-state-wf-twl-state-fold-add-init-cls:*
assumes *wf-twl-state S*
shows *wf-twl-state (fold add-init-cls N S)*
using *assms apply (induction N arbitrary: S)*
apply (*auto simp: wf-twl-state-def*)
by (*simp add: wf-twl-add-init-cls*)

lemma *wf-twl-state-epsilon-state[simp]:*
wf-twl-state (TWL-State [] {#} {#} 0 C-True)
by (*auto simp: wf-twl-state-def*)

lemma *wf-twl-init-state: wf-twl-state (init-state N)*
unfolding *init-state-def* **by** (*auto intro!: wf-twl-state-wf-twl-state-fold-add-init-cls*)

lemma *rough-state-of-twl-init-state:*
rough-state-of-twl (init-state-twl N) = init-state N
by (*simp add: twl-of-rough-state-inverse wf-twl-init-state*)

abbreviation *tl-trail-twl* **where**
tl-trail-twl S \equiv twl-of-rough-state (tl-trail (rough-state-of-twl S))

lemma *wf-twl-state-tl-trail: wf-twl-state S \implies wf-twl-state (tl-trail S)*
by (*simp add: twl-of-rough-state-inverse wf-twl-init-state wf-twl-cls-wf-twl-cls-tl tl-trail-def wf-twl-state-def distinct-tl map-tl*)

lemma *rough-state-of-twl-tl-trail:*

rough-state-of-twl (*tl-trail-twl* *S*) = *tl-trail* (*rough-state-of-twl* *S*)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-tl-trail* **by** *blast*

abbreviation *update-backtrack-lvl-twl* **where**

update-backtrack-lvl-twl *k S* \equiv *twl-of-rough-state* (*update-backtrack-lvl* *k* (*rough-state-of-twl* *S*))

lemma *wf-twl-state-update-backtrack-lvl*:

wf-twl-state *S* \implies *wf-twl-state* (*update-backtrack-lvl* *k S*)

unfolding *wf-twl-state-def* **by** (*auto simp: update-backtrack-lvl-def*)

lemma *rough-state-of-twl-update-backtrack-lvl*:

rough-state-of-twl (*update-backtrack-lvl-twl* *k S*) = *update-backtrack-lvl* *k*
(*rough-state-of-twl* *S*)

using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-update-backtrack-lvl* **by** *fast*

abbreviation *update-conflicting-twl* **where**

update-conflicting-twl *k S* \equiv *twl-of-rough-state* (*update-conflicting* *k* (*rough-state-of-twl* *S*))

lemma *wf-twl-state-update-conflicting*:

wf-twl-state *S* \implies *wf-twl-state* (*update-conflicting* *k S*)

unfolding *wf-twl-state-def* **by** (*auto simp: update-conflicting-def*)

lemma *rough-state-of-twl-update-conflicting*:

rough-state-of-twl (*update-conflicting-twl* *k S*) = *update-conflicting* *k*
(*rough-state-of-twl* *S*)

using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-update-conflicting* **by** *fast*

abbreviation *raw-clauses-twl* **where**

raw-clauses-twl *S* \equiv *raw-clauses* (*rough-state-of-twl* *S*)

abbreviation *restart-twl* **where**

restart-twl *S* \equiv *twl-of-rough-state* (*restart'* (*rough-state-of-twl* *S*))

lemma *wf-wf-restart'*: *wf-twl-state* *S* \implies *wf-twl-state* (*restart'* *S*)

unfolding *restart'-def wf-twl-state-def* **apply** *standard*

apply *clarify*

apply (*rename-tac* *x*)

apply (*subgoal-tac wf-twl-cls* (*trail* *S*) *x*)

apply (*case-tac* *x*)

using *restart-learned* **by** *fastforce+*

lemma *rough-state-of-twl-restart-twl*:

rough-state-of-twl (*restart-twl* *S*) = *restart'* (*rough-state-of-twl* *S*)

by (*simp add: twl-of-rough-state-inverse wf-wf-restart'*)

interpretation *cdcl_W-twl-NOT*: *dpll-state*

$\lambda S.$ *convert-trail-from-W* (*trail-twl* *S*)

raw-clauses-twl

$\lambda L S.$ *cons-trail-twl* (*convert-marked-lit-from-NOT* *L*) *S*

$\lambda S.$ *tl-trail-twl* *S*

$\lambda C S.$ *add-learned-cls-twl* *C S*

$\lambda C S.$ *remove-cls-twl* *C S*

apply *unfold-locales*

apply (*simp add: rough-state-of-twl-cons-trail*)

```

    apply (metis rough-state-of-twl-tl-trail rough-cdcl.tl-trail)
    apply (metis rough-state-of-twl-add-learned-cls rough-cdcl.trail-add-clsNOT)
    apply (metis rough-state-of-twl-remove-cls rough-cdcl.trail-remove-cls)
    apply (simp add: rough-state-of-twl-cons-trail)
    apply (simp add: twl.rough-state-of-twl-tl-trail)
    using rough-cdcl.clauses-add-clsNOT rough-cdcl.clauses-def rough-state-of-twl-add-learned-cls
    apply auto[1]
    using rough-cdcl.clauses-def rough-cdcl.clauses-remove-cls rough-state-of-twl-remove-cls by auto

```

interpretation $cdcl_W$ -twl: $state_W$

```

trail-twl
init-clss-twl
learned-clss-twl
backtrack-lvl-twl
conflicting-twl
cons-trail-twl
tl-trail-twl
add-init-cls-twl
add-learned-cls-twl
remove-cls-twl
update-backtrack-lvl-twl
update-conflicting-twl
init-state-twl
restart-twl
apply unfold-locales
by (simp-all add: rough-state-of-twl-cons-trail rough-state-of-twl-tl-trail
    rough-state-of-twl-add-init-cls rough-state-of-twl-add-learned-cls rough-state-of-twl-remove-cls
    rough-state-of-twl-update-backtrack-lvl rough-state-of-twl-update-conflicting
    rough-state-of-twl-init-state rough-state-of-twl-restart-twl
    rough-cdcl.learned-clss-restart-state)

```

interpretation $cdcl_W$ -twl: $cdcl_W$ -ops

```

trail-twl
init-clss-twl
learned-clss-twl
backtrack-lvl-twl
conflicting-twl
cons-trail-twl
tl-trail-twl
add-init-cls-twl
add-learned-cls-twl
remove-cls-twl
update-backtrack-lvl-twl
update-conflicting-twl
init-state-twl
restart-twl
by unfold-locales

```

abbreviation $state\text{-}eq\text{-}twl$ (**infix** $\sim TWL$ 51) **where**

$state\text{-}eq\text{-}twl\ S\ S' \equiv rough\text{-}cdcl.state\text{-}eq\ (rough\text{-}state\text{-}of\text{-}twl\ S)\ (rough\text{-}state\text{-}of\text{-}twl\ S')$

notation $cdcl_W$ -twl.state-eq (**infix** \sim 51)

declare $cdcl_W$ -twl.state-simp[simp del]

$cdcl_W$ -twl.state-simp_{NOT}[simp del]

$cdcl_W$ -twl-NOT.state-simp_{NOT}[simp del]

To avoid ambiguities:

no-notation *CDCL-Two-Watched-Literals.twl.state-eq-tw* (**infix** \sim *TWL* 51)

definition *propagate-tw* **where**

propagate-tw $S \ S' \longleftrightarrow$
 $(\exists L \ C. (L, C) \in \text{candidates-propagate-tw } S$
 $\wedge S' \sim \text{TWL cons-trail-tw} (\text{Propagated } L \ C) \ S$
 $\wedge \text{conflicting-tw } S = C\text{-True})$

lemma *propagate-tw-iff-propagate*:

assumes *inv*: *cdcl_W-twl.cdcl_W-all-struct-inv* S
shows *cdcl_W-twl.propagate* $S \ T \longleftrightarrow \text{propagate-tw } S \ T$ (**is** $?P \longleftrightarrow ?T$)

proof

assume $?P$

then obtain $C \ L$ **where**

conflicting (*rough-state-of-tw* S) = $C\text{-True}$ **and**
 $CL\text{-Clauses}$: $C + \{\#L\# \} \in \# \text{ cdcl}_W\text{-twl.clauses } S$ **and**
 $tr\text{-CNot}$: $\text{trail-tw } S \models_{as} C\text{Not } C$ **and**
 $undef\text{-lot}$: $\text{undefined-lit} (\text{trail-tw } S) \ L$ **and**
 $T \sim \text{cons-trail-tw} (\text{Propagated } L \ (C + \{\#L\# \})) \ S$
unfolding *cdcl_W-twl.propagate.simps* **by** *blast*

have *distinct-mset* $(C + \{\#L\# \})$

using *inv* $CL\text{-Clauses}$ **unfolding** *cdcl_W-twl.cdcl_W-all-struct-inv-def*
cdcl_W-twl.distinct-cdcl_W-state-def *cdcl_W-twl.clauses-def* *distinct-mset-set-def*
by (*metis* (*no-types*, *lifting*) *add-gr-0* *mem-set-mset-iff* *plus-multiset.rep-eq*)

then have $C\text{-L-L}$: *mset-set* (*set-mset* $(C + \{\#L\# \}) - \{L\}$) = C

by (*metis* *Un-insert-right* *add-diff-cancel-left'* *add-diff-cancel-right'*
distinct-mset-set-mset-ident *finite-set-mset* *insert-absorb2* *mset-set.insert-remove*
set-mset-single *set-mset-union*)

have $(L, C + \{\#L\# \}) \in \text{candidates-propagate-tw } S$

apply (*rule* *wf-candidates-propagate-complete*)

using *rough-state-of-tw* **apply** *auto*[]

using $CL\text{-Clauses}$ **unfolding** *cdcl_W-twl.clauses-def* **apply** *auto*[]

apply *simp*

using $C\text{-L-L}$ $tr\text{-CNot}$ **apply** *simp*

using *undef-lot* **apply** *blast*

done

show $?T$ **unfolding** *propagate-tw-def*

apply (*rule* *exI*[*of* - L], *rule* *exI*[*of* - $C + \{\#L\# \}$])

apply (*auto* *simp*: $\langle (L, C + \{\#L\# \}) \in \text{candidates-propagate-tw } S \rangle$

$\langle \text{conflicting} (\text{rough-state-of-tw } S) = C\text{-True} \rangle$)

using $\langle T \sim \text{cons-trail-tw} (\text{Propagated } L \ (C + \{\#L\# \})) \ S \rangle$ *cdcl_W-twl.state-eq-backtrack-lvl*

cdcl_W-twl.state-eq-conflicting *cdcl_W-twl.state-eq-init-clss*

cdcl_W-twl.state-eq-learned-clss *cdcl_W-twl.state-eq-trail* *rough-cdcl.state-eq-def* **by** *blast*

next

assume $?T$

then obtain $L \ C$ **where**

LC : $(L, C) \in \text{candidates-propagate-tw } S$ **and**

T : $T \sim \text{TWL cons-trail-tw} (\text{Propagated } L \ C) \ S$ **and**

confl : $\text{conflicting} (\text{rough-state-of-tw } S) = C\text{-True}$

unfolding *propagate-tw-def* **by** *auto*

have [*simp*]: $C - \{\#L\# \} + \{\#L\# \} = C$

using LC **unfolding** *candidates-propagate-def*

by *clarify* (*metis* *add commute* *add-diff-cancel-right'* *count-diff* *insert-DiffM*
multi-member-last *not-gr0* *zero-diff*)

have $C \in \# \text{ raw-clauses-tw } S$

```

    using LC unfolding candidates-propagate-def rough-cdcl.clauses-def by auto
  then have distinct-mset C
    using inv unfolding cdclW-twl.cdclW-all-struct-inv-def cdclW-twl.distinct-cdclW-state-def
    cdclW-twl.clauses-def distinct-mset-set-def rough-cdcl.clauses-def by auto
  then have C-L-L: mset-set (set-mset C - {L}) = C - {#L#}
    by (metis ⟨C - {#L#} + {#L#} = C⟩ add-left-imp-eq diff-single-trivial
    distinct-mset-set-mset-ident finite-set-mset mem-set-mset-iff mset-set.remove
    multi-self-add-other-not-self union-commute)

  show ?P
  apply (rule cdclW-twl.propagate.intros[of - trail-twl S init-clss-twl S
    learned-clss-twl S backtrack-lvl-twl S C-{#L#} L])
    using confl apply auto[]
    using LC unfolding candidates-propagate-def apply (auto simp: cdclW-twl.clauses-def)[]
    using wf-candidates-propagate-sound[OF - LC] rough-state-of-twl apply (simp add: C-L-L)
    using wf-candidates-propagate-sound[OF - LC] rough-state-of-twl apply simp
    using T unfolding cdclW-twl.state-eq-def rough-cdcl.state-eq-def by auto
qed

term local.state-eq-twl
term CDCL-Two-Watched-Literals.twl.state-eq-twl
definition conflict-twl where
  conflict-twl S S'  $\longleftrightarrow$ 
    (∃ C. C ∈ candidates-conflict-twl S
    ∧ S' ∼TWL update-conflicting-twl (C-Clause C) S
    ∧ conflicting-twl S = C-True)

lemma conflict-twl-iff-conflict:
  shows cdclW-twl.conflict S T  $\longleftrightarrow$  conflict-twl S T (is ?C  $\longleftrightarrow$  ?T)
proof
  assume ?C
  then obtain M N U k C where
    S: rough-cdcl.state (rough-state-of-twl S) = (M, N, U, k, C-True) and
    C: C ∈# cdclW-twl.clauses S and
    M-C: M ⊨as CNot C and
    T: T ∼ update-conflicting-twl (C-Clause C) S
    by auto
  have C ∈ candidates-conflict-twl S
    apply (rule wf-candidates-conflict-complete)
    apply simp
    using C apply (auto simp: cdclW-twl.clauses-def)[]
    using M-C S by auto
  moreover have T ∼TWL twl-of-rough-state (update-conflicting (C-Clause C) (rough-state-of-twl S))
    using T unfolding rough-cdcl.state-eq-def cdclW-twl.state-eq-def by auto
  ultimately show ?T
    using S unfolding conflict-twl-def by auto
next
  assume ?T
  then obtain C where
    C: C ∈ candidates-conflict-twl S and
    T: T ∼TWL update-conflicting-twl (C-Clause C) S and
    confl: conflicting-twl S = C-True
    unfolding conflict-twl-def by auto
  have C ∈# cdclW-twl.clauses S
    using C unfolding candidates-conflict-def cdclW-twl.clauses-def by auto

```

moreover have $\text{trail-twl } S \models_{\text{as}} C \text{Not } C$
 using $\text{wf-candidates-conflict-sound}[OF - C]$ by *auto*
 ultimately show $?C$ apply –
 apply (rule $\text{cdcl}_W\text{-twl.conflict.conflict-rule}[of \text{ - - - - } C]$)
 using $\text{confl } T$ unfolding $\text{rough-cdcl.state-eq-def } \text{cdcl}_W\text{-twl.state-eq-def}$ by *auto*
 qed

inductive $\text{cdcl}_W\text{-twl} :: 'v \text{ wf-twl} \Rightarrow 'v \text{ wf-twl} \Rightarrow \text{bool}$ **for** $S :: 'v \text{ wf-twl}$ **where**
 $\text{propagate: propagate-twl } S S' \Longrightarrow \text{cdcl}_W\text{-twl } S S' \mid$
 $\text{conflict: conflict-twl } S S' \Longrightarrow \text{cdcl}_W\text{-twl } S S' \mid$
 $\text{other: cdcl}_W\text{-twl.cdcl}_W\text{-o } S S' \Longrightarrow \text{cdcl}_W\text{-twl } S S' \mid$
 $\text{rf: cdcl}_W\text{-twl.cdcl}_W\text{-rf } S S' \Longrightarrow \text{cdcl}_W\text{-twl } S S'$

lemma $\text{cdcl}_W\text{-twl-iff-cdcl}_W$:
 assumes $\text{cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv } S$
 shows $\text{cdcl}_W\text{-twl } S T \longleftrightarrow \text{cdcl}_W\text{-twl.cdcl}_W S T$
 by (simp add: $\text{assms } \text{cdcl}_W\text{-twl.cdcl}_W.\text{sims } \text{cdcl}_W\text{-twl.sims } \text{conflict-twl-iff-conflict}$
 $\text{propagate-twl-iff-propagate}$)

lemma $\text{rtranclp-cdcl}_W\text{-twl-all-struct-inv-inv}$:
 assumes $\text{cdcl}_W\text{-twl}^{**} S T$ **and** $\text{cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv } S$
 shows $\text{cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv } T$
 using assms **by** (induction rule: rtranclp-induct)
 (simp-all add: $\text{cdcl}_W\text{-twl-iff-cdcl}_W \text{ cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv-inv}$)

lemma $\text{rtranclp-cdcl}_W\text{-twl-iff-rtranclp-cdcl}_W$:
 assumes $\text{cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv } S$
 shows $\text{cdcl}_W\text{-twl}^{**} S T \longleftrightarrow \text{cdcl}_W\text{-twl.cdcl}_W^{**} S T$ (**is** $?T \longleftrightarrow ?W$)

proof
 assume $?W$
 then show $?T$
proof (induction rule: rtranclp-induct)
 case *base*
 then show $?case$ **by** *simp*
 next
 case (step $T U$) **note** $st = \text{this}(1)$ **and** $cdcl = \text{this}(2)$ **and** $IH = \text{this}(3)$
 have $\text{cdcl}_W\text{-twl } T U$
 using $\text{assms } st \text{ cdcl } \text{cdcl}_W\text{-twl.rtranclp-cdcl}_W\text{-all-struct-inv-inv } \text{cdcl}_W\text{-twl-iff-cdcl}_W$
 by *blast*
 then show $?case$ **using** IH **by** *auto*
 qed
 next
 assume $?T$
 then show $?W$
proof (induction rule: rtranclp-induct)
 case *base*
 then show $?case$ **by** *simp*
 next
 case (step $T U$) **note** $st = \text{this}(1)$ **and** $cdcl = \text{this}(2)$ **and** $IH = \text{this}(3)$
 have $\text{cdcl}_W\text{-twl.cdcl}_W T U$
 using $\text{assms } st \text{ cdcl } \text{rtranclp-cdcl}_W\text{-twl-all-struct-inv-inv } \text{cdcl}_W\text{-twl-iff-cdcl}_W$
 by *blast*
 then show $?case$ **using** IH **by** *auto*
 qed
 qed

interpretation *cdcl_{NOT}-twl: backjumping-ops*

λS. convert-trail-from-W (trail-twl S)
abstract-twl.raw-clauses-twl
λL (S:: 'v wf-twl).
cons-trail-twl
(convert-marked-lit-from-NOT L) (S:: 'v wf-twl)
tl-trail-twl
add-learned-cls-twl
remove-cls-twl
λC - - (S:: 'v wf-twl) -. C ∈ candidates-conflict-twl S
by *unfold-locales*

lemma *reduce-trail-to_{NOT}-skip-beginning-twl:*

assumes *trail-twl S = convert-trail-from-NOT (F' @ F)*
shows *trail-twl (cdcl_W-twl.reduce-trail-to_{NOT} F S) = convert-trail-from-NOT F*
using *assms* **by** *(induction F' arbitrary: S) auto*

lemma *reduce-trail-to_{NOT}-trail-tl-trail-twl-decomp[simp]:*

trail-twl S = convert-trail-from-NOT (F' @ Marked K () # F) ⇒
trail-twl (cdcl_W-twl.reduce-trail-to_{NOT} F (tl-trail-twl S)) = convert-trail-from-NOT F
apply *(rule reduce-trail-to_{NOT}-skip-beginning-twl[of - tl (F' @ Marked K () # [])])*
by *(cases F') (auto simp add:tl-append rough-cdcl.reduce-trail-to_{NOT}-skip-beginning)*

lemma *trail-twl-reduce-trail-to_{NOT}-drop:*

trail-twl (cdcl_W-twl.reduce-trail-to_{NOT} F S) =
(if length (trail-twl S) ≥ length F
then drop (length (trail-twl S) - length F) (trail-twl S)
else [])
apply *(induction F S rule: cdcl_W-twl.reduce-trail-to_{NOT}.induct)*
apply *(rename-tac F S)*
apply *(case-tac trail-twl S)*
apply *auto[]*
apply *(rename-tac list)*
apply *(case-tac Suc (length list) > length F)*
prefer 2 **apply** *simp*
apply *(subgoal-tac Suc (length list) - length F = Suc (length list - length F))*
apply *simp*
apply *simp*
done

lemma *undefined-lit-convert-trail-from-NOT[simp]:*

undefined-lit (convert-trail-from-NOT F) L ↔ undefined-lit F L
by *(induction F rule: marked-lit-list-induct) (auto simp: defined-lit-map)*

lemma *lits-of-convert-trail-from-NOT:*

lits-of (convert-trail-from-NOT F) = lits-of F
by *(induction F rule: marked-lit-list-induct) auto*

lemma *map-eq-cons-decomp:*

assumes *SF: map f l = xs @ ys*
shows *∃ xs' ys'. l = xs' @ ys' ∧ map f xs' = xs ∧ map f ys' = ys*

proof –

```

let ?F' = take (length xs) l
let ?G = drop (length xs) l
have tr1: l = ?F' @ ?G
  by simp
moreover
  have [simp]: length l = length xs + length ys
    using arg-cong[OF SF, of length] by auto
  have map f ?F' = xs and map f ?G = ys
    using arg-cong[OF SF, of take (length xs)] apply (subst (asm) tr1)
    unfolding map-append apply simp
    using arg-cong[OF SF, of drop (length xs)] apply (subst (asm) tr1)
    unfolding map-append apply simp
  done
ultimately show ?thesis by blast
qed

```

interpretation *cdcl_{NOT}-twl: dpll-with-backjumping-ops*

λS. convert-trail-from-W (trail-tw1 S)

abstract-tw1.raw-clauses-tw1

λL S.

cons-trail-tw1

(convert-marked-lit-from-NOT L) S

tl-trail-tw1

add-learned-cls-tw1

remove-cls-tw1

λL S. lit-of L ∈ fst ‘candidates-propagate-tw1 S

λS. no-dup (trail-tw1 S)

λC - - S -. C ∈ candidates-conflict-tw1 S

proof (*unfold-locales, goal-cases*)

case (*1 C' S C F' K F L*) **note** *n-d = this(1)* **and** *n-d' = this(2)* **and** *undef = this(6)*

let *?T' = (cons-trail (Propagated L {#}) (rough-state-of-tw1 (cdcl_W-tw1.reduce-trail-to_{NOT} F S)))*

let *?T = (cons-trail-tw1 (Propagated L {#}) (cdcl_W-tw1.reduce-trail-to_{NOT} F S))*

have *tr-F-S: map lit-of (trail-tw1 (cdcl_W-tw1.reduce-trail-to_{NOT} F S)) =*

map lit-of (convert-trail-from-NOT F)

apply (*subst trail-tw1-reduce-trail-to_{NOT}-drop[of F S]*)

using *1(1) arg-cong[OF 1(3), of length] arg-cong[OF 1(3), of map lit-of]*

by (*auto simp: o-def drop-map[symmetric]*)

have *no-dup (trail-tw1 S)*

using *1(1)* **by** *blast*

have *wf-tw1-state (rough-state-of-tw1 (cdcl_W-tw1.reduce-trail-to_{NOT} F S))*

using *wf-tw1-state-rough-state-of-tw1* **by** *blast*

moreover have *undef': undefined-lit (trail-tw1 (cdcl_W-tw1.reduce-trail-to_{NOT} F S)) L*

using *undef arg-cong[OF tr-F-S, of map atm-of]* **unfolding** *defined-lit-map image-set*

by (*simp add: o-def*)

ultimately have *wf-tw1-state ?T'*

by (*simp-all add: wf-tw1-state-cons-trail*)

then have *init-clss-tw1 ?T = init-clss-tw1 (cdcl_W-tw1.reduce-trail-to_{NOT} F S)*

using *1(6)* **by** (*simp add: undef'*)

then have [*simp*]: *init-clss-tw1 ?T = init-clss-tw1 S*

by (*simp add: cdcl_W-tw1.reduce-trail-to_{NOT}-reduce-trail-convert*)

have *learned-clss-tw1 ?T = learned-clss-tw1 (cdcl_W-tw1.reduce-trail-to_{NOT} F S)*

```

    by (smt 1(3) 1(6) append-assoc cdclW-twl.learned-clss-cons-trail
        cdclW-twl-NOT.reduce-trail-toNOT-eq-length cdclW-twl-NOT.reduce-trail-toNOT-nil
        cdclW-twl-NOT.reduce-trail-toNOT-skip-beginning comp-apply defined-lit-convert-trail-from-W
        list.sel(3) marked-lit.sel(2) rev.simps(2) rev-append rev-eq-Cons-iff
        cons-trail-tw-l-def)
  moreover have learned-clss-tw-l (cdclW-twl.reduce-trail-toNOT F S)
    = learned-clss-tw-l S
  by (simp add: cdclW-twl.reduce-trail-toNOT-reduce-trail-convert)
  ultimately have [simp]: learned-clss-tw-l ?T = learned-clss-tw-l S
    by simp
  have tr-L-F-S: map lit-of (trail-tw-l ?T)
    = map lit-of (Propagated L {#} # convert-trail-from-NOT F)
    using undef' tr-F-S by (simp add: o-def)
  have C-conflict-cand: C ∈ candidates-conflict-tw-l S
    apply (rule wf-candidates-tw-l-conflict-complete)
    using 1(1,4) apply (simp add: rough-cdcl.clauses-def)
    using 1(5) by (simp add: tr-L-F-S true-annots-true-cls lits-of-convert-trail-from-NOT)

  have cdclNOT-tw-l.backjump S
    (cons-trail-tw-l (convert-marked-lit-from-NOT (Propagated L ()))
      (cdclW-twl.reduce-trail-toNOT F S))
    apply (rule cdclNOT-tw-l.backjump.intros[of S F' K F - L C, OF 1(3) - 1(4-6) - 1(8-9)])
    unfolding cdclW-twl-NOT.state-eqNOT-def apply (metis convert-marked-lit-from-NOT.simps(1))
    using 1(7) 1(3) apply presburger
    using C-conflict-cand by simp
  then show ?case
    by blast
qed

interpretation cdclNOT-tw-l: dppl-with-backjumping
  λS. convert-trail-from-W (trail-tw-l S)
  abstract-tw-l.raw-clauses-tw-l
  λL (S:: 'v wf-tw-l).
    cons-trail-tw-l
    (convert-marked-lit-from-NOT L) (S:: 'v wf-tw-l)
  tl-trail-tw-l
  add-learned-clss-tw-l
  remove-clss-tw-l
  λL S. lit-of L ∈ fst 'candidates-propagate-tw-l S
  λS. no-dup (trail-tw-l S)
  λC - - (S:: 'v wf-tw-l) -. C ∈ candidates-conflict-tw-l S
  apply unfold-locals
  using cdclNOT-tw-l.dppl-bj-no-dup by (simp add: o-def)
end

end
theory Prop-Superposition
imports Partial-Clausal-Logic ../lib/Herbrand-Interpretation
begin
sledgehammer-params[verbose]
no-notation Herbrand-Interpretation.true-clss (infix |= 50)
notation Herbrand-Interpretation.true-clss (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*:
 $S \models_h C \longleftrightarrow \{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\} \models C$
unfolding *Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def*
by *auto*

lemma *herbrand-consistent-interp*:
consistent-interp ($\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\}$)
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*:
 $S \models_{hs} C \longleftrightarrow \{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\} \models_s C$
unfolding *true-clss-def Ball-def herbrand-interp-iff-partial-interp-cl 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 ($\neg \hat{\sup} \neg \hat{\sup}$)

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* = ($\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D$)

lemma *production-unfold*:

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
interp *C* \models_h *C* \wedge *S* *C* = {#}}

unfolding *interp-def* **by** (*rule* *production.simps*)

abbreviation *productive* *A* \equiv (*production* *A* \neq {#})

abbreviation *produces* :: 'a clause \Rightarrow 'a \Rightarrow bool **where**

produces *C* *A* \equiv *production* *C* = {*A*}

lemma *producesD*:

produces *C* *A* \implies *C* \in *N* \wedge *C* \neq {#} \wedge *Pos* *A* = *Max* (*set-mset* *C*) \wedge *count* *C* (*Pos* *A*) \leq 1 \wedge \neg
interp *C* \models_h *C* \wedge *S* *C* = {#}

unfolding *production-unfold* **by** *auto*

lemma *produces* *C* *A* \implies *Pos* *A* \in # *C*

by (*simp* *add*: *Max-in-lits* *producesD*)

lemma *interp'-def-in-set*:

interp *C* = ($\bigcup D \in \{D \in N. D \# \subset \# C\}. \text{production } D$)

unfolding *interp-def* **apply** *auto*

unfolding *production-unfold* **apply** *auto*

done

lemma *production-iff-produces*:

produces *D* *A* \longleftrightarrow *A* \in *production* *D*

unfolding *production-unfold* **by** *auto*

definition *Interp* :: 'a clause \Rightarrow 'a *interp* **where**

Interp *C* = *interp* *C* \cup *production* *C*

lemma

assumes *produces* *C* *P*

shows *Interp* *C* \models_h *C*

unfolding *Interp-def* *assms* **using** *producesD*[*OF* *assms*]

by (*metis* *Max-in-lits* *Un-insert-right* *insertI1* *pos-literal-in-imp-true-cls*)

definition *INTERP* :: 'a *interp* **where**

INTERP = ($\bigcup D \in N. \text{production } D$)

lemma *interp-subseteq-Interp*[*simp*]: *interp* *C* \subseteq *Interp* *C*

unfolding *Interp-def* **by** *simp*

lemma *Interp-as-UNION*: *Interp* *C* = ($\bigcup D \in \{D. D \# \subseteq \# C\}. \text{production } D$)

unfolding *Interp-def* *interp-def* *le-multiset-def* **by** *fast*

lemma *productive-not-empty*: *productive* *C* \implies *C* \neq {#}

unfolding *production-unfold* **by** *auto*

lemma *productive-imp-produces-Max-literal*: *productive* *C* \implies *produces* *C* (*atm-of* (*Max* (*set-mset* *C*)))

unfolding *production-unfold* **by** (*auto* *simp* *del*: *atm-of-Max-lit*)

lemma *productive-imp-produces-Max-atom*: $\text{productive } C \implies \text{produces } C (\text{Max } (\text{atms-of } C))$
unfolding *atms-of-def* *Max-atm-of-set-mset-commute*[*OF* *productive-not-empty*]
by (*rule* *productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-literal*: $\text{produces } C A \implies A = \text{atm-of } (\text{Max } (\text{set-mset } C))$
by (*metis* *Max-singleton insert-not-empty* *productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-atom*: $\text{produces } C A \implies A = \text{Max } (\text{atms-of } C)$
by (*metis* *Max-singleton insert-not-empty* *productive-imp-produces-Max-atom*)

lemma *produces-imp-Pos-in-lits*: $\text{produces } C A \implies \text{Pos } A \in\# C$
by (*auto* *intro*: *Max-in-lits* *dest*!: *producesD*)

lemma *productive-in-N*: $\text{productive } C \implies C \in N$
unfolding *production-unfold* **by** *auto*

lemma *produces-imp-atms-leq*: $\text{produces } C A \implies B \in \text{atms-of } C \implies B \leq A$
by (*metis* *Max-ge* *finite-atms-of* *insert-not-empty* *productive-imp-produces-Max-atom* *singleton-inject*)

lemma *produces-imp-neg-notin-lits*: $\text{produces } C A \implies \neg \text{Neg } A \in\# C$
by (*auto* *intro*!: *pos-Max-imp-neg-notin* *dest*: *producesD* *simp* *del*: *not-gr0*)

lemma *less-eq-imp-interp-subseteq-interp*: $C \# \subseteq\# D \implies \text{interp } C \subseteq \text{interp } D$
unfolding *interp-def* **by** *auto* (*metis* *multiset-order.order.strict-trans2*)

lemma *less-eq-imp-interp-subseteq-Interp*: $C \# \subseteq\# D \implies \text{interp } C \subseteq \text{Interp } D$
unfolding *Interp-def* **using** *less-eq-imp-interp-subseteq-interp* **by** *blast*

lemma *less-imp-production-subseteq-interp*: $C \# \subset\# D \implies \text{production } C \subseteq \text{interp } D$
unfolding *interp-def* **by** *fast*

lemma *less-eq-imp-production-subseteq-Interp*: $C \# \subseteq\# D \implies \text{production } C \subseteq \text{Interp } D$
unfolding *Interp-def* **using** *less-imp-production-subseteq-interp*
by (*metis* *multiset-order.le-imp-less-or-eq* *le-supI1* *sup-ge2*)

lemma *less-imp-Interp-subseteq-interp*: $C \# \subset\# D \implies \text{Interp } C \subseteq \text{interp } D$
unfolding *Interp-def*
by (*auto* *simp*: *less-eq-imp-interp-subseteq-interp* *less-imp-production-subseteq-interp*)

lemma *less-eq-imp-Interp-subseteq-Interp*: $C \# \subseteq\# D \implies \text{Interp } C \subseteq \text{Interp } D$
using *less-imp-Interp-subseteq-interp*
unfolding *Interp-def* **by** (*metis* *multiset-order.le-imp-less-or-eq* *le-supI2* *subset-refl* *sup-commute*)

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin \text{Interp } C \implies A \in \text{interp } D \implies C \# \subset\# D$
using *less-eq-imp-interp-subseteq-Interp* *multiset-linorder.not-less* **by** *blast*

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin \text{interp } C \implies A \in \text{interp } D \implies C \# \subset\# D$
using *less-eq-imp-interp-subseteq-interp* *multiset-linorder.not-less* **by** *blast*

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin \text{Interp } C \implies A \in \text{Interp } D \implies C \# \subset\# D$
using *less-eq-imp-Interp-subseteq-Interp* *multiset-linorder.not-less* **by** *blast*

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin \text{interp } C \implies A \in \text{Interp } D \implies C \# \subseteq\# D$

using *less-imp-Interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

lemma *interp-subseteq-INTERP*: *interp C ⊆ INTERP*
unfolding *interp-def INTERP-def* **by** (*auto simp: production-unfold*)

lemma *production-subseteq-INTERP*: *production C ⊆ INTERP*
unfolding *INTERP-def* **using** *production-unfold* **by** *blast*

lemma *Interp-subseteq-INTERP*: *Interp C ⊆ INTERP*
unfolding *Interp-def* **by** (*auto intro!: interp-subseteq-INTERP production-subseteq-INTERP*)

This lemma corresponds to theorem 2.7.6 page 66 of CW.

lemma *produces-imp-in-interp*:
assumes *a-in-c: Neg A ∈# C* **and** *d: produces D A*
shows *A ∈ interp C*

proof –

from *d* **have** *Max (set-mset D) = Pos A*

using *production-unfold* **by** *blast*

hence *D #⊂# {#Neg A#}*

by (*auto intro: Max-pos-neg-less-multiset*)

moreover have *{#Neg A#} #⊆# C*

by (*rule less-eq-imp-le-multiset*) (*rule mset-le-single[OF a-in-c[unfolded mem-set-mset-iff]]*)

ultimately show *?thesis*

using *d* **by** (*blast dest: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)

qed

lemma *neg-notin-Interp-not-produce*: *Neg A ∈# C ⟹ A ∉ Interp D ⟹ C #⊆# D ⟹ ¬ produces D'' A*

by (*auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-Interp*)

lemma *in-production-imp-produces*: *A ∈ production C ⟹ produces C A*

by (*metis insert-absorb productive-imp-produces-Max-atom singleton-insert-inj-eq'*)

lemma *not-produces-imp-notin-production*: *¬ produces C A ⟹ A ∉ production C*

by (*metis in-production-imp-produces*)

lemma *not-produces-imp-notin-interp*: *(∧ D. ¬ produces D A) ⟹ A ∉ 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, $I^D ⊆ I_{D'}$ does not hold.

lemma *true-Interp-imp-general*:

assumes

c-le-d: C #⊆# D **and**

d-lt-d': D #⊂# D' **and**

c-at-d: Interp D ⊨_h C **and**

subs: interp D' ⊆ (∪ C ∈ CC. 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: Pos A ∈# C* **and** *a-at-d: A ∈ Interp D*

by *blast*

from *a-at-d* **have** *A ∈ interp D'*

using *d-lt-d'* *less-imp-Interp-subseteq-interp* **by** *blast*

thus *?thesis*

```

    using subs a-in-c by (blast dest: contra-subsetD)
next
case False
then obtain A where a-in-c: Neg A ∈# C and A ∉ Interp D
    using c-at-d unfolding true-cls-def by blast
hence  $\bigwedge D''. \neg \text{produces } D'' A$ 
    using c-le-d neg-notin-Interp-not-produce by simp
thus ?thesis
    using a-in-c subs not-produces-imp-notin-production by auto
qed

lemma true-Interp-imp-imp:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{interp } D' \models_h C$ 
    using interp-def true-Interp-imp-general by simp

lemma true-Interp-imp-Interp:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{Interp } D' \models_h C$ 
    using Interp-as-UNION interp-subseteq-Interp true-Interp-imp-general by simp

lemma true-Interp-imp-INTERP:  $C \# \subseteq \# D \implies \text{Interp } D \models_h C \implies \text{INTERP} \models_h C$ 
    using INTERP-def interp-subseteq-INTERP
    true-Interp-imp-general[OF - less-multiset-right-total]
    by simp

lemma true-interp-imp-general:
  assumes
    c-le-d:  $C \# \subseteq \# D$  and
    d-lt-d':  $D \# \subset \# D'$  and
    c-at-d:  $\text{interp } D \models_h C$  and
    subs:  $\text{interp } D' \subseteq (\bigcup C \in CC. \text{production } C)$ 
  shows  $(\bigcup C \in CC. \text{production } C) \models_h C$ 
proof (cases  $\exists A. \text{Pos } A \in \# C \wedge A \in \text{interp } D$ )
case True
  then obtain A where a-in-c:  $\text{Pos } A \in \# C$  and a-at-d:  $A \in \text{interp } D$ 
    by blast
  from a-at-d have  $A \in \text{interp } D'$ 
    using d-lt-d' less-eq-imp-imp-subseteq-imp[OF multiset-order.less-imp-le] by blast
  thus ?thesis
    using subs a-in-c by (blast dest: contra-subsetD)
next
case False
  then obtain A where a-in-c:  $\text{Neg } A \in \# C$  and  $A \notin \text{interp } D$ 
    using c-at-d unfolding true-cls-def by blast
  hence  $\bigwedge D''. \neg \text{produces } D'' A$ 
    using c-le-d by (auto dest: produces-imp-in-imp less-eq-imp-imp-subseteq-imp)
  thus ?thesis
    using a-in-c subs not-produces-imp-notin-production by auto
qed

```

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

```

lemma true-interp-imp-imp:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{interp } D' \models_h C$ 
    using interp-def true-interp-imp-general by simp

```

```

lemma true-interp-imp-Interp:  $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{Interp } D' \models_h C$ 
    using Interp-as-UNION interp-subseteq-Interp[of D'] true-interp-imp-general by simp

```

```

lemma true-interp-imp-INTERP:  $C \# \subseteq \# D \implies \text{interp } D \models_h C \implies \text{INTERP} \models_h C$ 

```

```

using INTERP-def interp-subseteq-INTERP
      true-interp-imp-general[OF - less-multiset-right-total]
by simp

```

```

lemma productive-imp-false-interp: productive C  $\implies \neg \text{interp } C \models_h C$ 
unfolding production-unfold by auto

```

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

```

lemma cls-gt-double-pos-no-production:
  assumes D: {#Pos P, Pos P#}  $\# \subset \#$  C
  shows  $\neg \text{produces } C \ P$ 
proof -
  let ?D = {#Pos P, Pos P#}
  note D' = D[unfolded less-multisetHO]
  consider
    (P) count C (Pos P) ≥ 2
  | (Q) Q where Q > Pos P and Q ∈ # C
    using HOL.spec[OF HOL.conjunct2[OF D'], of Pos P] by auto
  thus ?thesis
    proof cases
      case Q
      have Q ∈ set-mset C
      using Q(2) by (auto split: split-if-asm)
      then have Max (set-mset C) > Pos P
      using Q(1) Max-gr-iff by blast
      thus ?thesis
        unfolding production-unfold by auto
    next
      case P
      thus ?thesis
        unfolding production-unfold by auto
  qed
qed

```

This lemma corresponds to theorem 2.7.6 page 66 of CW.

```

lemma
  assumes D: C + {#Neg P#}  $\# \subset \#$  D
  shows production D ≠ {P}
proof -
  note D' = D[unfolded less-multisetHO]
  consider
    (P) Neg P ∈ # D
  | (Q) Q where Q > Neg P and count D Q > count (C + {#Neg P#}) Q
    using HOL.spec[OF HOL.conjunct2[OF D'], of Neg P] by fastforce
  thus ?thesis
    proof cases
      case Q
      have Q ∈ set-mset D
      using Q(2) by (auto split: split-if-asm)
      then have Max (set-mset D) > Neg P
      using Q(1) Max-gr-iff by blast
      hence Max (set-mset D) > Pos P
      using less-trans[of Pos P Neg P Max (set-mset D)] by auto
      thus ?thesis
        unfolding production-unfold by auto
    qed
  qed

```

```

next
case P
hence Max (set-mset D) > Pos P
  by (meson Max-ge finite-set-mset le-less-trans linorder-not-le mem-set-mset-iff
      pos-less-neg)
thus ?thesis
  unfolding production-unfold by auto
qed
qed

```

```

lemma in-interp-is-produced:
  assumes P ∈ INTERP
  shows ∃ D. D + {#Pos P#} ∈ N ∧ 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 $MMax\ M \equiv Max\ (set-mset\ M)$

21.6 We can now define the rules of the calculus

inductive *superposition-rules* :: 'a clause \Rightarrow 'a clause \Rightarrow 'a clause \Rightarrow bool **where**
factoring: *superposition-rules* ($C + \{\#Pos\ P\# \} + \{\#Pos\ P\# \}$) B ($C + \{\#Pos\ P\# \}$) |
superposition-l: *superposition-rules* ($C_1 + \{\#Pos\ P\# \}$) ($C_2 + \{\#Neg\ P\# \}$) ($C_1 + C_2$)

inductive *superposition* :: 'a clauses \Rightarrow 'a clauses \Rightarrow bool **where**
superposition: $A \in N \Longrightarrow B \in N \Longrightarrow$ *superposition-rules* $A\ B\ C$
 \Longrightarrow *superposition* $N\ (N \cup \{C\})$

definition *abstract-red* :: 'a::wellorder clause \Rightarrow 'a clauses \Rightarrow bool **where**
abstract-red $C\ N = (clss-lt\ N\ C \models_p C)$

lemma *less-multiset[iff]*: $M < N \longleftrightarrow M \# \subset \# N$
unfolding *less-multiset-def* **by** auto

lemma *less-eq-multiset[iff]*: $M \leq N \longleftrightarrow M \# \subseteq \# N$
unfolding *less-eq-multiset-def* **by** auto

lemma *herbrand-true-clss-true-clss-clss-herbrand-true-clss*:

```

assumes
  AB: A ⊨hs B and
  BC: B ⊨p C
shows A ⊨h C

```

proof –

```

let ?I = {Pos P | P. P ∈ A} ∪ {Neg P | P. P ∉ A}
have B: ?I ⊨s B using AB
  by (auto simp add: herbrand-interp-iff-partial-interp-clss)

```

```

have IH: ⋀ I. total-over-set I (atms-of C)  $\Longrightarrow$  total-over-m I B  $\Longrightarrow$  consistent-interp I
   $\Longrightarrow$  I ⊨s B  $\Longrightarrow$  I ⊨ C using BC
  by (auto simp add: true-clss-clss-def)
show ?thesis

```

unfolding *herbrand-interp-iff-partial-interp-cls*
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 \mid 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 \mid 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 \mid 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 \mid 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-clss-def by auto
thus ?thesis
oops
lemma
  assumes
    CP:  $\neg$  clss-lt N ( $\{\#C\# \} + \{\#E\# \}$ )  $\models_p$   $\{\#C\# \} + \{\#Neg P\# \}$  and
    clss-lt N ( $\{\#C\# \} + \{\#E\# \}$ )  $\models_p$   $\{\#E\# \} + \{\#Pos P\# \} \vee$  clss-lt N ( $\{\#C\# \} + \{\#E\# \}$ )  $\models_p$ 
 $\{\#C\# \} + \{\#Neg P\# \}$ 
  shows clss-lt N ( $\{\#C\# \} + \{\#E\# \}$ )  $\models_p$   $\{\#E\# \} + \{\#Pos P\# \}$ 
oops

locale ground-ordered-resolution-with-redundancy =
  ground-resolution-with-selection +
  fixes redundant :: 'a::wellorder clause  $\Rightarrow$  'a clauses  $\Rightarrow$  bool
  assumes
    redundant-iff-abstract: redundant A N  $\longleftrightarrow$  abstract-red A N
begin
definition saturated :: 'a clauses  $\Rightarrow$  bool where
  saturated N  $\longleftrightarrow$  ( $\forall A B C. A \in N \longrightarrow B \in N \longrightarrow \neg$ redundant A N  $\longrightarrow \neg$ redundant B N
 $\longrightarrow$  superposition-rules A B C  $\longrightarrow$  redundant C N  $\vee C \in N$ )
lemma
  assumes
    saturated: saturated N and
    finite: finite N and
    empty:  $\{\#\} \notin N$ 
  shows INTERP N  $\models_{hs}$  N
proof (rule ccontr)
  let ?NI = INTERP N
  assume  $\neg$  ?thesis
  hence not-empty:  $\{E \in N. \neg ?N_I \models_h E\} \neq \{\}$ 
  unfolding true-clss-def Ball-def by auto
  def D  $\equiv$  Min  $\{E \in N. \neg ?N_I \models_h E\}$ 
  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:  $\neg ?N_I \models_h D$ 
  unfolding D-def
  by (metis (mono-tags, lifting) Min-in finite mem-Collect-eq not-empty rev-finite-subset subsetI)
  have cls-not-D:  $\bigwedge E. E \in N \implies E \neq D \implies \neg ?N_I \models_h E \implies D \leq E$ 
  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 =  $\{\#\}$   $\wedge$  Max (set-mset D)
= L)
  proof (cases S D =  $\{\#\}$ )
  case False
  then obtain L where L  $\in \#$  S D
  using Max-in-lits by blast
  moreover
  hence L  $\in \#$  D
  using S-selects-subseteq[of D] by auto
  hence D = (D -  $\{\#L\#\}$ ) +  $\{\#L\#\}$ 

```

```

    by auto
    ultimately show ?thesis using that by blast
next
let ?L = 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 D L] by (cases L) auto
thus False
proof cases
  case L note P = this(1) and S = this(2) and max = this(3)
  have count D L > 1
  proof (rule ccontr)
    assume ~ ?thesis
    hence count: count D L = 1
    unfolding D by auto
    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 obtain C' where C':D = C' + {#Pos P#} + {#Pos P#}
  unfolding D by (metis P add.left-neutral add-less-cancel-right count-single count-union
  multi-member-split)
have sup: superposition-rules D D (D - {#L#})
  unfolding C' L by (auto simp add: superposition-rules.simps)

```

have $C' + \{\#Pos\ P\# \} \# \subset \# C' + \{\#Pos\ P\# \} + \{\#Pos\ P\# \}$
by *auto*
moreover have $\neg ?N_{\mathcal{I}} \models h (D - \{\#L\# \})$
using *not-d-interp unfolding* $C' L$ **by** *auto*
ultimately have $C' + \{\#Pos\ P\# \} \notin N$
by (*metis (no-types, lifting) C' P add-diff-cancel-right' cls-not-D less-multiset multi-self-add-other-not-self not-le*)
have $D - \{\#L\# \} \# \subset \# D$
unfolding $C' L$ **by** *auto*
have $c'-p-p: C' + \{\#Pos\ P\# \} + \{\#Pos\ P\# \} - \{\#Pos\ P\# \} = C' + \{\#Pos\ P\# \}$
by *auto*
have *redundant* $(C' + \{\#Pos\ P\# \}) N$
using *saturated red sup* $\langle D \in N \rangle \langle C' + \{\#Pos\ P\# \} \notin N \rangle$ **unfolding** *saturated-def* $C' L c'-p-p$
by *blast*
moreover have $C' + \{\#Pos\ P\# \} \subseteq \# C' + \{\#Pos\ P\# \} + \{\#Pos\ P\# \}$
by *auto*
ultimately show *False*
using *red unfolding* C' *redundant-iff-abstract* **by** (*blast dest: abstract-red-subset-mset-abstract-red*)
next
case $Lneg$ **note** $L = this(1)$
have $P \in ?N_{\mathcal{I}}$
using *not-d-interp unfolding* D *true-cls-def* L **by** (*auto split: split-if-asm*)
then obtain E **where**
 $DPN: E + \{\#Pos\ P\# \} \in N$ **and**
 $prod: production\ N\ (E + \{\#Pos\ P\# \}) = \{P\}$
using *in-interp-is-produced* **by** *blast*
have *sup-EC: superposition-rules* $(E + \{\#Pos\ P\# \}) (C + \{\#Neg\ P\# \}) (E + C)$
using *superposition-l* **by** *fast*
hence *superposition* $N (N \cup \{E+C\})$
using $DPN \langle D \in N \rangle$ **unfolding** $D L$ **by** (*auto simp add: superposition.simps*)
have
 $PMax: Pos\ P = MMax\ (E + \{\#Pos\ P\# \})$ **and**
 $count\ (E + \{\#Pos\ P\# \})\ (Pos\ P) \leq 1$ **and**
 $S\ (E + \{\#Pos\ P\# \}) = \{\#\}$ **and**
 $\neg interp\ N\ (E + \{\#Pos\ P\# \}) \models h\ E + \{\#Pos\ P\# \}$
using *prod unfolding production-unfold* **by** *auto*
have $Neg\ P \notin \# E$
using *prod produces-imp-neg-notin-lits* **by** *force*
hence $\bigwedge y. y \in \# (E + \{\#Pos\ P\# \})$
 $\implies count\ (E + \{\#Pos\ P\# \})\ (Neg\ P) < count\ (C + \{\#Neg\ P\# \})\ (Neg\ P)$
by (*auto split: split-if-asm*)
moreover have $\bigwedge y. y \in \# (E + \{\#Pos\ P\# \}) \implies y < Neg\ P$
using $PMax$ **by** (*metis DPN Max-less-iff empty finite-set-mset mem-set-mset-iff pos-less-neg set-mset-eq-empty-iff*)
moreover have $E + \{\#Pos\ P\# \} \neq C + \{\#Neg\ P\# \}$
using *prod produces-imp-neg-notin-lits* **by** *force*
ultimately have $E + \{\#Pos\ P\# \} \# \subset \# C + \{\#Neg\ P\# \}$
unfolding *less-multiset_{HO}* **by** (*metis add.left-neutral add-lessD1*)
have *ce-lt-d: C + E* $\# \subset \# D$
unfolding $D L$
by (*metis (mono-tags, lifting) Max-pos-neg-less-multiset One-nat-def PMax count-single less-multiset-plus-right-nonempty mult-less-trans single-not-empty union-less-mono2 zero-less-Suc*)
have $?N_{\mathcal{I}} \models h\ E + \{\#Pos\ P\# \}$

```

    using  $\langle P \in ?N_{\mathcal{I}} \rangle$  by blast
  have  $?N_{\mathcal{I}} \models_h C+E \vee C+E \notin N$ 
    using ce-lt-d cls-not-D unfolding D-def by fastforce
  have  $Pos\ P \notin \# C+E$ 
    using  $D \langle P \in \text{ground-resolution-with-selection.INTERP } S\ N \rangle$ 
       $\langle \text{count } (E + \{\#Pos\ P\# \}) (Pos\ P) \leq 1 \rangle$  multi-member-skip not-d-interp by auto
  hence  $\bigwedge y. y \in \# C+E$ 
     $\implies \text{count } (C+E) (Pos\ P) < \text{count } (E + \{\#Pos\ P\# \}) (Pos\ P)$ 
    by (auto split: split-if-asm)

  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  $\text{clss-lt } N\ (C + E) \models_p E + \{\#Pos\ P\# \} \vee \text{clss-lt } N\ (C + E) \models_p C + \{\#Neg\ P\# \}$ 
    proof clarify
      assume  $CP: \neg \text{clss-lt } N\ (C + E) \models_p C + \{\#Neg\ P\# \}$ 
      { fix I
        assume
          total-over-m  $I\ (\text{clss-lt } N\ (C + E) \cup \{E + \{\#Pos\ P\# \}\})$  and
          consistent-interp I and
           $I \models_s \text{clss-lt } N\ (C + E)$ 
        hence  $I \models C + E$ 
          using abs sorry
        moreover have  $\neg I \models C + \{\#Neg\ P\# \}$ 
          using CP unfolding true-clss-cls-def
          sorry
        ultimately have  $I \models E + \{\#Pos\ P\# \}$  by auto
      }
      then show  $\text{clss-lt } N\ (C + E) \models_p E + \{\#Pos\ P\# \}$ 
        unfolding true-clss-cls-def by auto
    qed
    moreover have  $\text{clss-lt } N\ (C + E) \subseteq \text{clss-lt } N\ (C + \{\#Neg\ P\# \})$ 
      using ce-lt-d mult-less-trans unfolding clss-lt-def D L by force
    ultimately have  $\text{redundant } (C + \{\#Neg\ P\# \})\ N \vee \text{clss-lt } N\ (C + E) \models_p E + \{\#Pos\ P\# \}$ 
      unfolding redundant-iff-abstract abstract-red-def using true-clss-cls-subset by blast
    show False sorry
  qed
  moreover have  $\neg \text{redundant } (E + \{\#Pos\ P\# \})\ N$ 
    sorry
  ultimately have  $CEN: C + E \in N$ 
    using  $\langle D \in N \rangle \langle E + \{\#Pos\ P\# \} \in N \rangle$  saturated sup-EC red unfolding saturated-def D L
    by (metis union-commute)
  have  $CED: C + E \neq D$ 
    using D ce-lt-d by auto
  have  $\text{interp}: \neg \text{INTERP } N \models_h C + E$ 
    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  $\in$  clss-lt N B using AN AB unfolding clss-lt-def
    by (auto dest: less-eq-imp-le-multiset simp add: multiset-order.dual-order.order-iff-strict)
  thus ?thesis
    using AB unfolding abstract-red-def true-clss-cls-def Partial-Clausal-Logic.true-clss-def
    by blast
qed

inductive redundant :: 'a clause  $\Rightarrow$  'a clauses  $\Rightarrow$  bool where
  subsumption: A  $\in$  N  $\Longrightarrow$  A  $\subset\#$  B  $\Longrightarrow$  redundant B N

lemma redundant-is-redundancy-criterion:
  fixes A :: 'a :: wellorder clause and N :: 'a :: wellorder clauses
  assumes redundant A N
  shows abstract-red A N
  using assms
proof (induction rule: redundant.induct)
  case (subsumption A B N)
  thus ?case
    using subsumed-is-redundant[of A N B] unfolding abstract-red-def clss-lt-def by auto
qed

lemma redundant-mono:
  redundant A N  $\Longrightarrow$  A  $\subseteq\#$  B  $\Longrightarrow$  redundant B N
  apply (induction rule: redundant.induct)
  by (meson subset-mset.less-le-trans subsumption)

locale truc =
  selection S for S :: nat clause  $\Rightarrow$  nat clause
begin

end

end
theory Weidenbach-Book
imports
  Prop-Normalisation

  Prop-Resolution

  Prop-Superposition

  CDCL-NOT DPLL-NOT DPLL-W-Implementation CDCL-W-Implementation CDCL-W-Incremental
  CDCL-WNOT CDCL-Two-Watched-Literals

```

begin

end