

Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

February 23, 2016

Contents

1	Transitions	5
1.1	More theorems about Closures	5
1.2	Full Transitions	6
1.3	Well-Foundedness and Full Transitions	7
1.4	More Well-Foundedness	8
2	Various Lemmas	11
3	More List	12
3.1	<i>upt</i>	12
3.2	Lexicographic ordering	14
4	Logics	14
4.1	Definition and abstraction	14
4.2	properties of the abstraction	16
4.3	Subformulas and properties	18
4.4	Positions	21
5	Semantics over the syntax	24
6	Rewrite systems and properties	26
6.1	Lifting of rewrite rules	26
6.2	Consistency preservation	28
6.3	Full Lifting	29
7	Transformation testing	30
7.1	Definition and first properties	30
7.2	Invariant conservation	32
7.2.1	Invariant while lifting of the rewriting relation	33
7.2.2	Invariant after all rewriting	34
8	Rewrite Rules	36
8.1	Elimination of the equivalences	36
8.2	Eliminate Implication	37
8.3	Eliminate all the True and False in the formula	39
8.4	PushNeg	45
8.5	Push inside	50
8.5.1	Only one type of connective in the formula (+ not)	59

8.5.2	Push Conjunction	62
8.5.3	Push Disjunction	63
9	The full transformations	64
9.1	Abstract Property characterizing that only some connective are inside the others	64
9.1.1	Definition	64
9.2	Conjunctive Normal Form	66
9.2.1	Full CNF transformation	66
9.3	Disjunctive Normal Form	67
9.3.1	Full DNF transform	68
10	More aggressive simplifications: Removing true and false at the beginning	68
10.1	Transformation	68
10.2	More invariants	70
10.3	The new CNF and DNF transformation	74
11	Partial Clausal Logic	75
11.1	Clauses	75
11.2	Partial Interpretations	75
11.2.1	Consistency	75
11.2.2	Atoms	76
11.2.3	Totality	78
11.2.4	Interpretations	80
11.2.5	Satisfiability	82
11.2.6	Entailment for Multisets of Clauses	83
11.2.7	Tautologies	85
11.2.8	Entailment for clauses and propositions	86
11.3	Subsumptions	91
11.4	Removing Duplicates	92
11.5	Set of all Simple Clauses	92
11.6	Experiment: Expressing the Entailments as Locales	95
11.7	Entailment to be extended	96
12	Resolution	97
12.1	Simplification Rules	97
12.2	Unconstrained Resolution	99
12.2.1	Subsumption	99
12.3	Inference Rule	100
12.4	Lemma about the simplified state	115
12.5	Resolution and Invariants	117
12.5.1	Invariants	118
12.5.2	well-foundedness if the relation	123
13	Partial Clausal Logic	138
13.1	Marked Literals	138
13.1.1	Definition	138
13.1.2	Entailment	139
13.1.3	Defined and undefined literals	141
13.2	Backtracking	143
13.3	Decomposition with respect to the marked literals	143

13.4	Negation of Clauses	150
13.5	Other	154
14	NOT's CDCL	155
14.1	Auxiliary Lemmas and Measure	155
14.2	Initial definitions	159
14.2.1	The state	159
14.2.2	Definition of the operation	162
14.3	DPLL with backjumping	163
14.3.1	Definition	163
14.3.2	Basic properties	164
14.3.3	Termination	167
14.3.4	Normal Forms	172
14.4	CDCL	179
14.4.1	Learn and Forget	179
14.4.2	Definition of CDCL	180
14.5	CDCL with invariant	183
14.6	Termination	189
14.6.1	Restricting learn and forget	189
14.7	CDCL with restarts	200
14.7.1	Definition	200
14.7.2	Increasing restarts	201
14.8	Merging backjump and learning	208
14.8.1	Instantiations	220
15	DPLL as an instance of NOT	235
15.1	DPLL with simple backtrack	235
15.2	Adding restarts	239
16	DPLL	240
16.1	Rules	240
16.2	Invariants	240
16.3	Termination	249
16.4	Final States	251
16.5	Link with NOT's DPLL	252
16.5.1	Level of literals and clauses	254
16.5.2	Properties about the levels	258
17	Weidenbach's CDCL	260
17.1	The State	260
17.2	Special Instantiation: using Triples as State	267
17.3	CDCL Rules	267
17.4	Invariants	272
17.4.1	Properties of the trail	272
17.4.2	Better-Suited Induction Principle	277
17.4.3	Compatibility with $op \sim$	280
17.4.4	Conservation of some Properties	283
17.4.5	Learned Clause	283
17.4.6	No alien atom in the state	285
17.4.7	No duplicates all around	287

17.4.8	Conflicts and co	288
17.4.9	Putting all the invariants together	297
17.4.10	No tautology is learned	300
17.5	CDCL Strong Completeness	301
17.6	Higher level strategy	302
17.6.1	Definition	302
17.6.2	Invariants	304
17.6.3	Literal of highest level in conflicting clauses	310
17.6.4	Literal of highest level in marked literals	314
17.6.5	Strong completeness	323
17.6.6	No conflict with only variables of level less than backtrack level	329
17.6.7	Final States are Conclusive	340
17.7	Termination	346
17.8	No Relearning of a clause	347
17.9	Decrease of a measure	362
18	Simple Implementation of the DPLL and CDCL	368
18.1	Common Rules	368
18.1.1	Propagation	368
18.1.2	Unit propagation for all clauses	369
18.1.3	Decide	370
18.2	Simple Implementation of DPLL	371
18.2.1	Combining the propagate and decide: a DPLL step	371
18.2.2	Adding invariants	374
18.2.3	Code export	380
18.3	CDCL Implementation	383
18.3.1	Definition of the rules	383
18.3.2	The Transitions	384
18.3.3	Code generation	396
19	Link between Weidenbach's and NOT's CDCL	409
19.1	Inclusion of the states	409
19.2	Additional Lemmas between NOT and W states	414
19.3	More lemmas conflict-propagate and backjumping	415
19.3.1	Termination	415
19.3.2	More backjumping	415
19.4	CDCL FW	429
19.5	FW with strategy	438
19.5.1	The intermediate step	438
19.6	Adding Restarts	474
20	Incremental SAT solving	484
21	2-Watched-Literal	497
21.1	Datastructure and Access Functions	497
21.2	Invariants	498
21.3	Abstract 2-WL	506
21.4	Instanciación of the previous locale	508
21.5	Interpretation for <i>cdcl_W.cdcl_W</i>	517
21.5.1	Direct Interpretation	517

21.5.2 Opaque Type with Invariant	518
21.6 We can now define the rules of the calculus	533

22 Implementation for 2 Watched-Literals 540

theory *Wellfounded-More*
imports *Main*

begin

1 Transitions

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

1.1 More theorems about Closures

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

lemma *tranclp-mono-explicit*:

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

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

lemma *tranclp-mono*:

assumes *mono*: $r \leq s$

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

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

lemma *tranclp-idemp-rel*:

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

apply (rule *iffI*)

prefer 2 apply *blast*

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

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

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

by *simp*

lemmas *tranclp-idemp*[*simp*] = *trancl-idemp*[*to-pred*]

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

lemma *rtranclp-unfold*: $rtranclp r a b \longleftrightarrow (a = b \vee tranclp r a b)$

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

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

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

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

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

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

apply (rule *iffI*)

```

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

```

```

lemma tranclp-rtranclp-rtranclp-rel:  $R^{+++} a b \longleftrightarrow R^{**} a b$ 
  by (simp add: rtranclp-unfold)

```

```

lemma tranclp-rtranclp-rtranclp[simp]:  $R^{+++} = R^{**}$ 
  by (fastforce simp: rtranclp-unfold)

```

```

lemma rtranclp-exists-last-with-prop:
  assumes  $R x z$ 
  and  $R^{**} z z'$  and  $P x z$ 
  shows  $\exists y y'. R^{**} x y \wedge R y y' \wedge P y y' \wedge (\lambda a b. R a b \wedge \neg P a b)^{**} y' z'$ 
  using assms(2,1,3)
proof (induction arbitrary: )
  case base
  then show ?case by auto
next
  case (step  $z' z''$ ) note  $z = \text{this}(2)$  and  $IH = \text{this}(3)[OF \text{this}(4-5)]$ 
  show ?case
    apply (cases  $P z' z''$ )
    apply (rule exI[of -  $z'$ ], rule exI[of -  $z''$ ])
    using  $z$  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 $\text{no-step step } S \equiv (\forall S'. \neg \text{step } S S')$

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

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

```

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

```

```

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

```

```

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

```

```

lemma tranclp-full-full1I:
   $R^{++} a b \Longrightarrow \text{full } R b c \Longrightarrow \text{full1 } R a c$ 

```

unfolding *full-def full1-def* **by** *auto*

lemma *full-fullI*:

$R\ a\ b \implies \text{full}\ R\ b\ c \implies \text{full1}\ R\ a\ c$

unfolding *full-def full1-def* **by** *auto*

lemma *full-unfold*:

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

unfolding *full-def full1-def* **by** (*auto simp add: rtranclp-unfold*)

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

by (*simp add: full-unfold*)

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

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

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

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

lemma *full1-tranclp-relation-full*:

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

by (*metis converse-tranclpE full1-def reflclp-tranclp rtranclpD rtranclp-idemp rtranclp-reflclp tranclp.r-into-trancl tranclp-into-rtranclp*)

lemma *full-tranclp-relation-full*:

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

by (*metis full-unfold full1-tranclp-relation-full tranclp.r-into-trancl tranclpD*)

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

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

proof –

have $\forall p\ a\ aa.\ \neg p^{**}\ (a::'a)\ aa \vee a = aa \vee (\exists ab.\ p^{**}\ a\ ab \wedge p\ ab\ aa)$

by (*metis rtranclp.cases*)

then obtain $aa :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ **where**

$f1: \forall p\ a\ ab.\ \neg p^{**}\ a\ ab \vee a = ab \vee p^{**}\ a\ (aa\ p\ a\ ab) \wedge p\ (aa\ p\ a\ ab)\ ab$

by *moura*

{ assume $a \neq b$

{ assume $\neg \text{full1}\ R\ a\ b \wedge a \neq b$

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

using $f1$ **by** (*metis (no-types) full1-def full1-tranclp-relation-full*)

then have *?thesis*

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

then have *?thesis*

by *auto* }

then show *?thesis*

by *fastforce*

qed

lemma *tranclp-full1-full1*:

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

by (*metis full1-def rtranclp-full1-eq-or-full1 tranclp-unfold-begin*)

1.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:

assumes $wf:wf\ \{(x, y). R\ y\ x\}$

```

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

```

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

```

```

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

```



```

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

lemma wf-wf-if-measure':
  assumes wf r and H: (⋀ x y. P x ⟹ g x y ⟹ (f y, f x) ∈ r)
  shows wf {(y, x). P x ∧ g x y}
proof -
  have wf {(b, a). (f b, f a) ∈ r} using assms(1) wf-if-measure-f by auto
  then have wf {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r}
    using wf-subset[of - {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r}] by auto
  moreover have {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r} ⊆ {(b, a). (f b, f a) ∈ r} by auto
  moreover have {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r} = {(b, a). P a ∧ g a b} using H by auto
  ultimately show ?thesis using wf-subset by simp
qed

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

lemma wfP-if-measure2: fixes f :: 'a ⇒ nat
  shows (⋀ x y. P x y ⟹ g x y ⟹ f x < f y) ⟹ wf {(x, y). P x y ∧ 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: ⋀ U. P U ⟹ U ∈ A and
    finite: finite A and
    wf: wf R and
    trans: trans R
  shows wf {(T, S). (P S ∧ P T) ∧ (T, S) ∈ lexord R}
proof (rule wfP-if-measure2)
  fix T S
  assume P: P S ∧ P T and
  s-le-t: (T, S) ∈ lexord R
  let ?f = λS. {U. (U, S) ∈ lexord R ∧ P U ∧ P S}
  have ?f T ⊆ ?f S
    using s-le-t P lexord-trans trans by auto
  moreover have T ∈ ?f S
    using s-le-t P by auto
  moreover have T ∉ ?f T
    using s-le-t by (auto simp add: lexord-irreflexive local.wf)
  ultimately have {U. (U, T) ∈ lexord R ∧ P U ∧ P T} ⊆ {U. (U, S) ∈ lexord R ∧ P U ∧ P S}
    by auto
  moreover have finite {U. (U, S) ∈ lexord R ∧ P U ∧ P S}
    using finite by (metis (no-types, lifting) P-finite finite-subset mem-Collect-eq subsetI)
  ultimately show card (?f T) < card (?f S) by (simp add: psubset-card-mono)
qed

```

```

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

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

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

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

```

```

end
theory List-More
imports Main
begin

```

2 Various Lemmas

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

thm *nat-less-induct*

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

assumes

$P\ 0$ **and**

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

shows $P\ n$

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

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

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

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

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

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* :: $('a \Rightarrow 'b::ord) \Rightarrow bool$ **where**

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

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

fixes $f :: nat \Rightarrow nat$

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

apply (*case-tac n ≤ n₀*)

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

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 $k :: nat$ **and** $f :: nat \Rightarrow nat$

assumes $k > 0$

```

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

```

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

```

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

```

3 More List

3.1 *upt*

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

```

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

```

```

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

```

```

declare upt.simps(2)[simp del]

```

```

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

```

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

```

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

```

```

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

```

qed

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

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

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

oops

A more restrictive version holds:

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

proof

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

next

assume $?P$ and $?B$

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

qed

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

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

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

proof –

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

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

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

using assms length-upt by presburger

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

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

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

qed

lemma append-cons-eq-upt-length:

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

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

using assms

proof (induction A arbitrary: m)

case Nil

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

next

case (Cons a A)

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

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

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

qed

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

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

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

proof –

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

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

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

using assms length-upt by auto

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

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

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

lemma *upt-decomp-lt*:
 assumes H : $xs\ @\ i\ \# \ ys\ @\ j\ \# \ zs = [m\ ..<\ n]$
 shows $i < j$
 proof -
 have xs : $xs = [m\ ..<\ i]$ and ys : $ys = [Suc\ i\ ..<\ j]$ and zs : $zs = [Suc\ j\ ..<\ n]$
 using H by (auto dest: *append-cons-eq-upt-length-i* *append-cons-eq-upt-length-i-end*)
 show $?thesis$
 by (metis *append-cons-eq-upt-length-i-end* *assms* *lessI* *less-trans* *self-append-conv2*
upt-eq-Cons-conv *upt-rec* ys)
 qed

3.2 Lexicographic ordering

We are working a lot on lexicographic ordering over pairs.

lemma *list-length2-append-cons*:
 $[c, d] = ys\ @\ y\ \# \ ys' \longleftrightarrow (ys = [] \wedge y = c \wedge ys' = [d]) \vee (ys = [c] \wedge y = d \wedge ys' = [])$
 by (cases ys ; cases ys') *auto*

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

end
 theory *Prop-Logic*

imports *Main*

begin

4 Logics

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

4.1 Definition and abstraction

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

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

| *FImp* 'v propo 'v propo | *FEq* 'v propo 'v propo

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

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

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

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

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

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

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

fixes $\varphi\ \psi :: 'v\ propo$
assumes *nullary*: $(\bigwedge \varphi\ x. \varphi = FF \vee \varphi = FT \vee \varphi = FVar\ x \implies P\ \varphi)$
and *unary*: $(\bigwedge \psi. P\ \psi \implies P\ (FNot\ \psi))$
and *binary*: $(\bigwedge \varphi\ \psi1\ \psi2. P\ \psi1 \implies P\ \psi2 \implies \varphi = FAnd\ \psi1\ \psi2 \vee \varphi = FOr\ \psi1\ \psi2 \vee \varphi = FImp\ \psi1\ \psi2 \vee \varphi = FEq\ \psi1\ \psi2 \implies P\ \varphi)$
shows $P\ \psi$
apply (*induct rule: propo.induct*)
using *assms* **by** *metis+*

The function *conn* is the interpretation of our representation (connective and list of arguments). We define any thing that has no sense to be false

fun *conn* :: 'v connective \Rightarrow 'v propo **list** \Rightarrow 'v propo **where**

conn *CT* [] = *FT* |
conn *CF* [] = *FF* |
conn (*CVar* *v*) [] = *FVar* *v* |
conn *CNot* [φ] = *FNot* φ |
conn *CAnd* ($\varphi \# [\psi]$) = *FAnd* $\varphi\ \psi$ |
conn *COr* ($\varphi \# [\psi]$) = *FOr* $\varphi\ \psi$ |
conn *CImp* ($\varphi \# [\psi]$) = *FImp* $\varphi\ \psi$ |
conn *CEq* ($\varphi \# [\psi]$) = *FEq* $\varphi\ \psi$ |
conn - = *FF*

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

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

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

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

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

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

inductive *wf-conn* :: 'v connective \Rightarrow 'v propo list \Rightarrow bool **for** *c* :: 'v connective **where**
wf-conn-nullary[simp]: (*c* = *CT* \vee *c* = *CF* \vee *c* = *CVar* *v*) \Longrightarrow *wf-conn* *c* [] |
wf-conn-unary[simp]: *c* = *CNot* \Longrightarrow *wf-conn* *c* [*ψ*] |
wf-conn-binary[simp]: *c* \in *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: 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  $\implies$  ( $\exists$  a b. l = a # b # [])
apply (induct l, auto)
by (rename-tac l, case-tac l, auto)

```

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

```

lemma wf-conn-bin-list-length:
  fixes l :: 'v propo list
  assumes conn: c  $\in$  binary-connectives
  shows length l = 2  $\longleftrightarrow$  wf-conn c l
proof
  assume length l = 2
  then show wf-conn c l using wf-conn-binary list-length2-decomp using conn by metis
next
  assume wf-conn c l
  then show length l = 2 (is ?P l)
  proof (cases rule: wf-conn.induct)
    case wf-conn-nullary
    then show ?P [] using conn binary-connectives-def
      using connective.distinct(11) connective.distinct(13) connective.distinct(9) by blast
  next
    fix  $\psi$  :: 'v propo
    case wf-conn-unary
    then show ?P [ $\psi$ ] using conn binary-connectives-def
      using connective.distinct by blast
  next
    fix  $\psi$   $\psi'$  :: 'v propo
    show ?P [ $\psi$ ,  $\psi'$ ] by auto
  qed
qed

```

```

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

```

Decomposing the Not into an element is moreover very useful.

```

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

```

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

```

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

```

```

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

```

```

lemma wf-conn-no-arity-change-helper:
  length (ξ @ φ # ξ') = length (ξ @ φ' # ξ')
  by auto

```

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

```

lemma conn-inj-not:
  assumes correct: wf-conn c l
  and conn: conn c l = FNot ψ
  shows c = CNot and l = [ψ]
  apply (cases c l rule: wf-conn.cases)
  using correct conn unfolding binary-connectives-def apply auto
  apply (cases c l rule: wf-conn.cases)
  using correct conn unfolding binary-connectives-def by auto

```

```

lemma conn-inj:
  fixes c ca :: 'v connective and l ψs :: 'v propo list
  assumes corr: wf-conn ca l
  and corr': wf-conn c ψs
  and eq: conn ca l = conn c ψs
  shows ca = c ∧ ψs = l
  using corr
proof (cases ca l rule: wf-conn.cases)
  case (wf-conn-nullary v)
  then show ca = c ∧ ψs = l using assms
    by (metis conn.simps(1) conn.simps(2) conn.simps(3) wf-conn-list(1-3))
next
  case (wf-conn-unary ψ')
  then have *: FNot ψ' = conn c ψs using conn-inj-not eq assms by auto
  then have c = ca by (metis conn-inj-not(1) corr' wf-conn-unary(2))
  moreover have ψs = l using * conn-inj-not(2) corr' wf-conn-unary(1) by force
  ultimately show ca = c ∧ ψs = l by auto
next
  case (wf-conn-binary ψ' ψ'')
  then show ca = c ∧ ψs = l
    using eq corr' unfolding binary-connectives-def apply (cases ca, auto simp add: wf-conn-list)
    using wf-conn-list(4-7) corr' by metis+
qed

```

4.3 Subformulas and properties

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

```

inductive subformula :: 'v propo  $\Rightarrow$  'v propo  $\Rightarrow$  bool (infix  $\preceq$  45) for φ where
  subformula-refl[simp]: φ  $\preceq$  φ |
  subformula-into-subformula: ψ ∈ set l  $\implies$  wf-conn c l  $\implies$  φ  $\preceq$  ψ  $\implies$  φ  $\preceq$  conn c l

```

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

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:

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

apply (*induct rule: subformula.induct*)

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

lemma *subformula-in-binary-conn*:

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

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

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

proof –

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

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

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

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

next

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

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

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

qed

lemma *subformula-trans*:

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

apply (*induct ψ' rule: subformula.inducts*)

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

lemma *subformula-leaf*:

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

assumes *incl*: $\varphi \preceq \psi$

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

shows $\varphi = \psi$

using *incl simple*

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

lemma *subformula-not-incl-eq*:

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

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

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

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

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

using *conn-inj* **by** *blast*

lemma *wf-subformula-conn-cases*:

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

apply *standard*

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

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

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

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

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

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

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

proof –

```

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

```

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

```

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

```

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

lemma subformula-conn-decomp[simp]:

```

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

```

proof (rule iffI)

```

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

```

moreover assume ?A

ultimately show ?B using wf by metis

next

assume ?B

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

qed

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

$\varphi \preceq FT \longleftrightarrow \varphi = FT$
 $\varphi \preceq FF \longleftrightarrow \varphi = FF$
 $\varphi \preceq FVar\ x \longleftrightarrow \varphi = FVar\ x$
apply *auto*
using *subformula-leaf* **by** *metis* +

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

primrec *vars-of-prop*:: '*v* propo \Rightarrow '*v* set **where**

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

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

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

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

case *nullary*
then have *False* **using** *corr* *incl* **by** *auto*
then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn* *c* *l*) **by** *blast*

next

case *binary* **note** *c = this*
then obtain *a* *b* **where** *ab*: *l* = [*a*, *b*]
using *wf-conn-bin-list-length* *list-length2-decomp* *corr* **by** *metis*
then have $\psi = a \vee \psi = b$ **using** *incl* **by** *auto*
then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn* *c* *l*)
using *ab* *c* **unfolding** *binary-connectives-def* **by** *auto*

next

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

qed

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:

$\varphi \preceq \psi \implies$ *vars-of-prop* $\varphi \subseteq$ *vars-of-prop* ψ
apply (*induct* *rule*: *subformula.induct*)
apply *simp*
using *vars-of-prop-incl-conn* **by** *blast*

4.4 Positions

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

datatype *sign* = *L* | *R*

We use *nil* instead of ε .

```

fun pos :: 'v propo  $\Rightarrow$  sign list set where
pos FF = {} |
pos FT = {} |
pos (FVar x) = {} |
pos (FAnd  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FOr  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FEq  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FImp  $\varphi$   $\psi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }  $\cup$  { R # p | p. p  $\in$  pos  $\psi$  } |
pos (FNot  $\varphi$ ) = {}  $\cup$  { L # p | p. p  $\in$  pos  $\varphi$  }

```

```

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

```

```

lemma finite-inj-comp-set:

```

```

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

```

```

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

```

```

next

```

```

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

```

```

qed

```

```

lemma cons-inject:

```

```

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

```

```

lemma finite-insert-nil-cons:

```

```

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

```

```

lemma cord-not[simp]:

```

```

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

```

```

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

```

```

lemma card-seperate:

```

```

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

```

```

proof -

```

```

have finite ?L using assms by auto

```

```

moreover have finite ?R using assms by auto

```

```

moreover have ?L  $\cap$  ?R = {} by blast

```

ultimately show *?thesis* **using** *assms card-Un-disjoint* **by** *blast*
qed

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

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

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

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

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

proof –

fix $\varphi 1 \ \varphi 2 :: 'v \text{ propo}$

assume *IH1*: $\text{card } (\text{vars-of-prop } \varphi 1) \leq \text{card } (\text{pos } \varphi 1)$

and *IH2*: $\text{card } (\text{vars-of-prop } \varphi 2) \leq \text{card } (\text{pos } \varphi 2)$

let $?L = \{L \# p \mid p. p \in \text{pos } \varphi 1\}$

let $?R = \{R \# p \mid p. p \in \text{pos } \varphi 2\}$

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

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

moreover have $\dots = \text{card } (\text{pos } \varphi 1) + \text{card } (\text{pos } \varphi 2)$

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

moreover have $\dots \geq \text{card } (\text{vars-of-prop } \varphi 1) + \text{card } (\text{vars-of-prop } \varphi 2)$ **using** *IH1 IH2* **by** *arith*

then have $\dots \geq \text{card } (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2)$ **using** *card-Un-le le-trans* **by** *blast*

ultimately

show $\text{card } (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc } (\text{card } (?L \cup ?R))$

$\text{card } (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc } (\text{card } (?L \cup ?R))$

$\text{card } (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc } (\text{card } (?L \cup ?R))$

$\text{card } (\text{vars-of-prop } \varphi 1 \cup \text{vars-of-prop } \varphi 2) \leq \text{Suc } (\text{card } (?L \cup ?R))$

by *auto*

qed

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

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

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

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

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

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

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

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

lemma *path-to-subformula*:

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

apply (*induct rule: path-to.induct*)

apply *simp*

apply (*metis list.set-intros*(1) *subformula-into-subformula*)

using *subformula-trans subformula-in-binary-conn*(2) **by** *metis*

lemma *subformula-path-exists*:

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

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

proof (*induct rule: subformula.induct*)

case *subformula-refl*

have *path-to* $[] \ \varphi' \ \varphi'$ **by** *auto*

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

```

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

```

```

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

```

5 Semantics over the syntax

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

```

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

```


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

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

lemma *deduction-rule*:

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

proof

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

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

$\{$

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

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

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

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

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

ultimately have $A \models FImp \varphi \psi$ **by** *blast*
 $\}$
then show $\forall A. A \models FImp \varphi \psi$ **by** *blast*
next
show $\forall A. A \models FImp \varphi \psi \implies \varphi \models_f \psi$
 proof (*rule ccontr*)
 assume $\neg \varphi \models_f \psi$
 then obtain A **where** $A \models \varphi \wedge \neg A \models \psi$ **using** *evalf-def* **by** *metis*
 then have $\neg A \models FImp \varphi \psi$ **by** *auto*
 moreover assume $\forall A. A \models FImp \varphi \psi$
 ultimately show *False* **by** *blast*
qed
qed

A shorter proof:

lemma $\varphi \models_f \psi \longleftrightarrow (\forall A. A \models FImp \varphi \psi)$
by (*simp add: evalf-def*)

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

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

lemma *same-over-set-eval*:

```

assumes same-over-set  $A\ B$  (vars-of-prop  $\varphi$ )
shows  $A \models \varphi \longleftrightarrow B \models \varphi$ 
using assms unfolding same-over-set-def by (induct  $\varphi$ , auto)

```

```

end
theory Prop-Abstract-Transformation
imports Main Prop-Logic Wellfounded-More

```

```

begin

```

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

6 Rewrite systems and properties

6.1 Lifting of rewrite rules

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

```

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

```

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

This lemma is only a health condition:

```

lemma propo-rew-step-subformula-imp:
shows  $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r\ \psi\ \psi'$ 
  apply (induct rule: propo-rew-step.induct)
  using subformula.simps subformula-into-subformula apply blast
  using wf-conn-no-arity-change subformula-into-subformula wf-conn-no-arity-change-helper in-set-conv-decomp by metis

```

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

```

lemma propo-rew-step-subformula-rec:
  fixes  $\psi\ \psi'\ \varphi :: 'v\ propo$ 
  shows  $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists\ \varphi'.\ \psi' \preceq \varphi' \wedge propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi')$ 
proof (induct  $\varphi$  rule: subformula.induct)
  case subformula-refl
  hence  $propo\text{-}rew\text{-}step\ r\ \psi\ \psi'$  using propo-rew-step.intros by auto
  moreover have  $\psi' \preceq \psi'$  using Prop-Logic.subformula-refl by auto
  ultimately show  $\exists\ \varphi'.\ \psi' \preceq \varphi' \wedge propo\text{-}rew\text{-}step\ r\ \psi\ \varphi'$  by fastforce
next
  case (subformula-into-subformula  $\psi''\ l\ c$ )
  note  $IH = this(4)$  and  $r = this(5)$  and  $\psi'' = this(1)$  and  $wf = this(2)$  and  $incl = this(3)$ 
  then obtain  $\varphi'$  where  $*: \psi' \preceq \varphi' \wedge propo\text{-}rew\text{-}step\ r\ \psi''\ \varphi'$  by metis
  moreover obtain  $\xi\ \xi' :: 'v\ propo\ list$  where
     $l: l = \xi\ @\ \psi''\ \# \xi'$  using List.split-list  $\psi''$  by metis

```

ultimately have *propo-rew-step* r (*conn* c l) (*conn* c ($\xi @ \varphi' \# \xi'$))
 using *propo-rew-step.intros*(2) *wf* **by** *metis*
 moreover have $\psi' \preceq \text{conn } c (\xi @ \varphi' \# \xi')$
 using *wf * wf-conn-no-arity-change Prop-Logic.subformula-into-subformula*
by (*metis (no-types) in-set-conv-decomp l wf-conn-no-arity-change-helper*)
 ultimately show $\exists \varphi'. \psi' \preceq \varphi' \wedge \text{propo-rew-step } r (\text{conn } c l) \varphi'$ **by** *metis*
qed

lemma *propo-rew-step-subformula*:
 $(\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi') \longleftrightarrow (\exists \varphi'. \text{propo-rew-step } r \varphi \varphi')$
 using *propo-rew-step-subformula-imp propo-rew-step-subformula-rec* **by** *metis*+

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

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

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

lemma *propo-rew-step-rewrite*:
 fixes $\varphi \varphi' :: 'v \text{ propo}$ **and** $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$
 assumes *propo-rew-step* $r \varphi \varphi'$
 shows $\exists \psi \psi' p. r \psi \psi' \wedge \text{path-to } p \varphi \psi \wedge \text{replace-at } p \varphi \psi' = \varphi'$
using *assms*
proof (*induct rule: propo-rew-step.induct*)
 case (*global-rel* $\varphi \psi$)
 moreover have *path-to* $\square \varphi \varphi$ **by** *auto*
 moreover have *replace-at* $\square \varphi \psi = \psi$ **by** *auto*
 ultimately show *?case* **by** *metis*
next
 case (*propo-rew-one-step-lift* $\varphi \varphi' c \xi \xi'$) **note** *rel = this(1)* **and** *IH0 = this(2)* **and** *corr = this(3)*
 obtain $\psi \psi' p$ where *IH*: $r \psi \psi' \wedge \text{path-to } p \varphi \psi \wedge \text{replace-at } p \varphi \psi' = \varphi'$ **using** *IH0* **by** *metis*
 {

```

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

```

6.2 Consistency preservation

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

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

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

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

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

unfolding *preserves-un-sat-def*

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

case *global-rel*

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

next

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

and *IH = this(3)[OF this(4) this(1)]* **and** *consistent = 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 *preserves-un-sat g* \implies *preserves-un-sat (f OO g)*

unfolding *preserves-un-sat-def* **by** *auto*

lemma *star-consistency-preservation-explicit*:

assumes $(\text{propo-rew-step } r)^{**} \varphi \psi$ **and** *preserves-un-sat r*

shows $\forall A. A \models \varphi \longleftrightarrow A \models \psi$

using *assms* **by** (*induct rule: rtranclp-induct*)

(*auto simp add: propo-rew-step-preservers-val-explicit*)

lemma *star-consistency-preservation*:

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

by (*simp add: star-consistency-preservation-explicit preserves-un-sat-def*)

6.3 Full Lifting

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

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

preserves-un-sat r \implies *preserves-un-sat (full (propo-rew-step r))*

by (*metis full-def preserves-un-sat-def star-consistency-preservation*)

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

full (propo-rew-step r) \varphi' \varphi $\implies \neg(\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi')$

unfolding *full-def* **using** *propo-rew-step-subformula-rec* **by** *metis*

7 Transformation testing

7.1 Definition and first properties

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

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

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

lemma *all-subformula-st-test-symb-true-phi*:
all-subformula-st test-symb $\varphi \Longrightarrow \text{test-symb } \varphi$
unfolding *all-subformula-st-def* **by** *auto*

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

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

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

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

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

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

proof –

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

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

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

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

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

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

lemma *subformula-all-subformula-st*:

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

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

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

```

fixes r:: 'v propo  $\Rightarrow$  'v propo  $\Rightarrow$  bool and test-symb:: 'v propo  $\Rightarrow$  bool and x:: 'v
and  $\varphi$  :: 'v propo
assumes test-symb-false-nullary:  $\forall x. \text{test-symb } FF \wedge \text{test-symb } FT \wedge \text{test-symb } (FVar\ x)$ 
and  $\forall \varphi'. \varphi' \preceq \varphi \longrightarrow (\neg \text{test-symb } \varphi') \longrightarrow (\exists \psi. r\ \varphi'\ \psi)$  and
 $\neg \text{all-subformula-st test-symb } \varphi$ 
shows  $(\exists \psi\ \psi'. \psi \preceq \varphi \wedge r\ \psi\ \psi')$ 
using assms
proof (induct  $\varphi$  rule: propo-induct-arity)
case (nullary  $\varphi\ x$ )
thus  $\exists \psi\ \psi'. \psi \preceq \varphi \wedge r\ \psi\ \psi'$ 
using wf-conn-nullary test-symb-false-nullary by fastforce
next
case (unary  $\varphi$ ) note IH = this(1)[OF this(2)] and r = this(2) and nst = this(3) and subf =
this(4)
from r IH nst have H:  $\neg \text{all-subformula-st test-symb } \varphi \Longrightarrow \exists \psi. \psi \preceq \varphi \wedge (\exists \psi'. r\ \psi\ \psi')$ 
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 = this(4)
and  $\varphi = \text{this}(3)$  and le = this(5) and nst = this(6)

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

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

```

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

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

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

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

assumes

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

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

$\text{propo-rew-step } r \varphi \psi$ **and**

$\text{all-subformula-st test-symb } \varphi$

shows $\text{all-subformula-st test-symb } \psi$

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

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

assumes

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

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

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

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

$\text{init: all-subformula-st test-symb } \varphi$

shows $\text{all-subformula-st test-symb } \psi$

using *assms 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: rtranclp-induct*)

case *base*

then show $\text{all-subformula-st test-symb } \varphi$ **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** $\text{all} = \text{this}(4)$

then have $\text{all-subformula-st test-symb } b$ **by** *metis*

then show $\text{all-subformula-st test-symb } c$ **using** *propo-rew-step-inv-stay'* $H \ H' \ \text{rel one}$ **by** *auto*

qed

qed

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

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

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

assumes

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

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

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

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

$\text{init: all-subformula-st test-symb } \varphi$

shows $\text{all-subformula-st test-symb } \psi$

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

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

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

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

assumes

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

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

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

init: *all-subformula-st test-symb* φ

shows *all-subformula-st test-symb* ψ

unfolding *full-def*

proof –

have *rel*: (*propo-rew-step* r)^{**} $\varphi \psi$

using *full* **unfolding** *full-def* **by** *auto*

thus *all-subformula-st test-symb* ψ

using *init*

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

case *base*

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

next

case (*step* $b \ c$)

note *star* = *this*(1) **and** *IH* = *this*(3) **and** *one* = *this*(2) **and** *all* = *this*(4)

hence *all-subformula-st test-symb* b **by** *metis*

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

using *propo-rew-step-inv-stay subformula-refl* $H \ H'$ *rel one* **by** *auto*

qed

qed

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

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

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

assumes

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

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

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

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

init: *all-subformula-st test-symb* φ

shows *all-subformula-st test-symb* ψ

proof –

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

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

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

thus *all-subformula-st test-symb* ψ

using H *full init full-propo-rew-step-inv-stay* **by** *blast*

qed

end

theory *Prop-Normalisation*

imports *Main Prop-Logic Prop-Abstract-Transformation*

begin

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

8 Rewrite Rules

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

8.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

inductive *elim-equiv* :: 'v propo \Rightarrow 'v propo \Rightarrow bool **where**
elim-equiv[simp]: *elim-equiv* (FEq φ ψ) (FAnd (FImp φ ψ) (FImp ψ φ))

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

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

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

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

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

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

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

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

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

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

The following lemma helps to reconstruct *no-equiv* expressions: this representation is easier to use than the set definition.

lemma *all-subformula-st-decomp-explicit-no-equiv*[iff]:

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

shows

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

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

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

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

assumes *no-equiv*: $\neg \text{no-equiv } \varphi$

shows $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elim-equiv } \psi \psi'$

proof –

have *test-symb-false-nullary*:

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

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

moreover {

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

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

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

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

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

}

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

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

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

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

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

ultimately show *?thesis*

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

qed

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

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

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

using *full-propo-rew-step-subformula no-equiv-elim-equiv-step* **by** *blast*

8.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* $:: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **where**

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

lemma *elim-imp-transformation-consistent*:

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

by *auto*

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

by (*induct* $\varphi \ \psi$ *rule: elim-imp.induct, auto*)

lemma *elim-imp-consistent*: *preserves-un-sat elim-imp*

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

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

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

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

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

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

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

Invariant of the *elim-imp* transformation

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

lemma *elim-imp-inv*:
fixes $\varphi \psi :: 'v \text{ propo}$
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 no-imp φ*
shows $\exists \psi \psi'. \psi \preceq \varphi \wedge$ *elim-imp $\psi \psi'$*

proof –

have *test-symb-false-nullary: $\forall x. \text{no-imp-symb } FF \wedge \text{no-imp-symb } FT \wedge \text{no-imp-symb } (FVar (x :: 'v))$*
by *auto*
moreover {
fix $c :: 'v \text{ connective}$ **and** $l :: 'v \text{ propo list}$ **and** $\psi :: 'v \text{ propo}$
have *H: elim-imp (conn c l) $\psi \implies \neg$ no-imp-symb (conn c l)*
by (*auto elim: elim-imp.cases*)
}

```

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

```

```

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

```

8.3 Eliminate all the True and False in the formula

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

inductive elimTB **where**

ElimTB1: elimTB (FAnd φ FT) φ |

ElimTB1': elimTB (FAnd FT φ) φ |

ElimTB2: elimTB (FAnd φ FF) FF |

ElimTB2': elimTB (FAnd FF φ) FF |

ElimTB3: elimTB (FOr φ FT) FT |

ElimTB3': elimTB (FOr FT φ) FT |

ElimTB4: elimTB (FOr φ FF) φ |

ElimTB4': elimTB (FOr FF φ) φ |

ElimTB5: elimTB (FNot FT) FF |

ElimTB6: elimTB (FNot FF) FT

lemma elimTB-consistent: preserves-un-sat elimTB

proof –

```

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

```

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

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

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

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

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

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

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

qed

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

```

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

```

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

inductive *no-T-F-symb-except-toplevel* **where**

```

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

```


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

fixes $x :: 'v$
shows *no-T-F-symb-except-toplevel* ($FVar\ x$)
by *simp*

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

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

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

fixes $\varphi\ \psi :: 'v\ propo$
assumes $\varphi \neq FT$ **and** $\varphi \neq FF$ **and** $\psi \neq FT$ **and** $\psi \neq FF$
and $c: c \in \text{binary-connectives}$
shows *no-T-F-symb-except-toplevel* ($\text{conn}\ c\ [\varphi, \psi]$)
by (*metis* (*no-types*, *lifting*) *assms* $c\ \text{conn.simps}(4)\ \text{list.discI}\ \text{noTrue-no-T-F-symb-except-toplevel}$
 $\text{wf-conn-no-T-F-symb-iff}\ \text{no-T-F-symb-fnot}\ \text{set.ConsD}\ \text{wf-conn-binary}\ \text{wf-conn-helper-facts}(1)$
 $\text{wf-conn-list-decomp}(1,2))$

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

fixes $l :: 'v\ propo\ \text{list}$ **and** $c :: 'v\ \text{connective}$
assumes *corr*: $\text{wf-conn}\ c\ l$
and $FT \in \text{set}\ l \vee FF \in \text{set}\ l$
shows $\neg \text{no-T-F-symb-except-toplevel}\ (\text{conn}\ c\ l)$
by (*metis* *assms* *empty-iff* *no-T-F-symb-except-toplevel.simps* $\text{wf-conn-no-T-F-symb-iff}\ \text{set-empty}$
 $\text{wf-conn-list}(1,2))$

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
 $\neg \text{no-T-F-symb-except-toplevel}\ (FAnd\ \varphi\ \psi)$
 $\neg \text{no-T-F-symb-except-toplevel}\ (FOr\ \varphi\ \psi)$
 $\neg \text{no-T-F-symb-except-toplevel}\ (FImp\ \varphi\ \psi)$
 $\neg \text{no-T-F-symb-except-toplevel}\ (FEq\ \varphi\ \psi)$
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*($14-15$) $\text{wf-conn-helper-facts}(5-8))$ +

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

proof –

have *wf: wf-conn* $c \ [\varphi, \psi]$ **using** c **by** *auto*
then have $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F-symb } (\text{conn } c \ [\varphi, \psi]) \wedge \text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
by (*simp add: all-subformula-st-decomp no-T-F-def*)
then show $\text{no-T-F } (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
using c *wf all-subformula-st-decomp list.discI no-T-F-def no-T-F-symb-except-toplevel-bin-decom*
no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(1,2) wf-conn-helper-facts(2,3)
wf-conn-list(1,2) **by** *metis*

qed

```

lemma no-T-F-bin-decomp-expanded[simp]:
  assumes c:  $c = CAnd \vee c = COr \vee c = CEq \vee c = CImp$ 
  shows  $no\text{-}T\text{-}F\ (conn\ c\ [\varphi, \psi]) \longleftrightarrow (no\text{-}T\text{-}F\ \varphi \wedge no\text{-}T\text{-}F\ \psi)$ 
  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  $\psi \preceq \varphi \implies \neg no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\ \psi \implies \exists \psi'.\ elimTB\ \psi\ \psi'$ 
proof (induct  $\psi$  rule: propo-induct-arity)
  case (nullary  $\varphi'\ x$ )
  then have False using no-T-F-symb-except-toplevel-true no-T-F-symb-except-toplevel-false by auto
  then show ?case by blast
next
  case (unary  $\psi$ )
  then have  $\psi = FF \vee \psi = FT$  using no-T-F-symb-except-toplevel-not-decom by blast
  then show ?case using ElimTB5 ElimTB6 by blast
next
  case (binary  $\varphi'\ \psi1\ \psi2$ )
  note IH1 = this(1) and IH2 = this(2) and  $\varphi' = this(3)$  and  $F\varphi = this(4)$  and  $n = this(5)$ 
  {
    assume  $\varphi' = FImp\ \psi1\ \psi2 \vee \varphi' = FEq\ \psi1\ \psi2$ 
    then have False using  $n\ F\varphi$  subformula-all-subformula-st assms
      by (metis (no-types) no-equiv-eq(1) no-equiv-def no-imp-imp(1) no-imp-def)
    then have ?case by blast
  }
moreover {

```

```

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

```

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

```

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

```

lemma *elimTB-inv*:

```

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

```

```

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

```

```

lemma elimTB-full-propo-rew-step:
  fixes  $\varphi$   $\psi$  :: 'v propo
  assumes no-equiv  $\varphi$  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 litteral.

inductive *pushNeg* **where**

```

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

```

lemma *pushNeg-transformation-consistent*:

```

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

```

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

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

lemma *pushNeg-lifted-consistant*:

```

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

```

fun *simple* **where**

```

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

```

lemma *simple-decomp*:

```

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

```

lemma *subformula-conn-decomp-simple*:

```

fixes  $\varphi$   $\psi$  :: 'v propo
assumes s: simple  $\psi$ 
shows  $\varphi \preceq \text{FNot } \psi \longleftrightarrow (\varphi = \text{FNot } \psi \vee \varphi = \psi)$ 
proof -
  have  $\varphi \preceq \text{conn CNot } [\psi] \longleftrightarrow (\varphi = \text{conn CNot } [\psi] \vee (\exists \psi \in \text{set } [\psi]. \varphi \preceq \psi))$ 

```

using *subformula-conn-decomp wf-conn-helper-facts(1)* **by** *metis*
then show $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \vee \varphi = \psi)$ **using** *s* **by** (*auto simp: simple-decomp*)
qed

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

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

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

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

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

lemma *simple-not-rew*:
fixes $\varphi :: 'v\ propo$
assumes *noTB*: \neg *simple-not* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
shows $\exists \psi\ \psi'. \psi \preceq \varphi \wedge \text{pushNeg}\ \psi\ \psi'$
proof –
have $\forall x. \text{simple-not-symb}\ (FF :: 'v\ propo) \wedge \text{simple-not-symb}\ FT \wedge \text{simple-not-symb}\ (FVar\ (x :: 'v))$
by *auto*
moreover {
fix $c :: 'v\ connective$ **and** $l :: 'v\ propo\ list$ **and** $\psi :: 'v\ propo$
have $H: \text{pushNeg}\ (\text{conn}\ c\ l)\ \psi \implies \neg \text{simple-not-symb}\ (\text{conn}\ c\ l)$
by (*cases* (*conn* $c\ l$) *rule: pushNeg.cases*) *auto*
}
moreover {
fix $x :: 'v$
have $H': \text{simple-not}\ FT\ \text{simple-not}\ FF\ \text{simple-not}\ (FVar\ x)$
by *simp-all*
}
moreover {
fix $\psi :: 'v\ propo$
have $\psi \preceq \varphi \implies \neg$ *simple-not-symb* $\psi \implies \exists \psi'. \text{pushNeg}\ \psi\ \psi'$

```

    using simple-not-step-exists no-equiv no-imp by blast
  }
  ultimately show ?thesis using no-test-symb-step-exists noTB unfolding simple-not-def by blast
qed

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

proof –

```

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

```

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

```

  propo-rew-step pushNeg  $\varphi \psi \implies$  no-T-F  $\varphi \implies$  no-T-F  $\psi$ 

```

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

```

case global-rel
then show ?case
  by (metis (no-types, lifting) no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb
      no-T-F-def no-T-F-except-top-level-pushNeg1 no-T-F-except-top-level-pushNeg2
      no-T-F-no-T-F-except-top-level all-subformula-st-decomp-explicit(3) pushNeg.simps
      simple.simps(1,2,5,6))
next
case (prop-rew-one-step-lift  $\varphi$   $\varphi'$   $c$   $\xi$   $\xi'$ )
note rel = this(1) and IH = this(2) and wf = this(3) and no-T-F = this(4)
moreover have wf': wf-conn  $c$  ( $\xi$  @  $\varphi'$  #  $\xi'$ )
  using wf-conn-no-arity-change wf-conn-no-arity-change-helper wf by metis
ultimately show no-T-F (conn  $c$  ( $\xi$  @  $\varphi'$  #  $\xi'$ ))
  using all-subformula-st-test-symb-true-phi
  by (fastforce simp: no-T-F-def all-subformula-st-decomp wf wf')
qed

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

```



```

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

```

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

```

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

```

8.5 Push inside

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

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

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

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

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

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

lemma *c-in-c'-symb-simp*:
 $\text{not-c-in-c'-symb } c \ c' \ \xi \Longrightarrow \xi = FF \vee \xi = FT \vee \xi = FVar \ x \vee \xi = FNot \ FF \vee \xi = FNot \ FT$
 $\vee \xi = FNot \ (FVar \ x) \Longrightarrow \text{False}$
apply (induct rule: *not-c-in-c'-symb.induct*, auto *simp*: *wf-conn.simps* *wf-conn-list*(1–3))
using *conn-inj-not*(2) *wf-conn-binary* **unfolding** *binary-connectives-def* **by** *fastforce+*

lemma *c-in-c'-symb-simp'[simp]*:
 $\neg \text{not-c-in-c'-symb } c \ c' \ FF$
 $\neg \text{not-c-in-c'-symb } c \ c' \ FT$

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

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

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

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

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

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

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

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

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

next

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

qed

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

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

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

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

proof –

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

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

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

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

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

```

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

```

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

```

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

```

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

```

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

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

```

```

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

```

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

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

```

  by (metis ab conn.simps(5-8) helper-fact simple-not-symb.simps(5) simple-not-symb.simps(6)
    simple-not-symb.simps(7) simple-not-symb.simps(8))
  ultimately show ?thesis
    by (simp add: ab all-subformula-st-decomp ca)
next
  case unary
  then show ?thesis
    using rew simple-propo-rew-step-push-conn-inside-inv[OF rew] IH local.wf simple by auto
qed
qed

```

```

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

```

```

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

```

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

lemma push-conn-inside-inv:

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

assumes full (propo-rew-step (push-conn-inside $c \ c'$) $\varphi \ \psi$)

and no-equiv φ and no-imp φ and no-T-F-except-top-level φ and simple-not φ

shows no-equiv ψ and no-imp ψ and no-T-F-except-top-level ψ and simple-not ψ

proof –

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

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

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

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

using H apply simp

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

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

and $x :: 'v$

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

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

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

then have False by auto

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

next

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

and $x :: 'v$

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

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

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

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

and c : $ca = CNot$

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

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

then have simple φ

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

then have simple φ'

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

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

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

next

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

and $x :: 'v$

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

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

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

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

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


```

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

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

```

```

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

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

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

```

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

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

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

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

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

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

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

lemma *only-c-inside-symb-decomp-not*[*simp*]:
fixes $c :: 'v \text{ connective}$
assumes $c: c \neq CNot$
shows *only-c-inside-symb* c (*FNot* ψ) \longleftrightarrow *simple* ψ
apply (*auto simp: only-c-inside-symb.intros(3)*)
by (*induct FNot* ψ *rule: only-c-inside-symb.induct, auto simp: wf-conn-list(8) c*)

lemma *only-c-inside-decomp-not*[*simp*]:
assumes $c: c \neq CNot$
shows *only-c-inside* c (*FNot* ψ) \longleftrightarrow *simple* ψ
by (*metis* (*no-types, hide-lams*) *all-subformula-st-def all-subformula-st-test-symb-true-phi c*
only-c-inside-def only-c-inside-symb-decomp-not simple-only-c-inside
subformula-conn-decomp-simple)

lemma *only-c-inside-decomp*:
only-c-inside c $\varphi \longleftrightarrow$

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

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

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

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

proof –

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

moreover

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

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

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

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

qed

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

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

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

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

proof –

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

qed

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

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

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

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

```

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

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

  have c1-eq: c1  $\neq$  CEq and c2-eq: c2  $\neq$  CEq
  unfolding no-equiv-def using inc $\varphi_1$  inc $\varphi_2$  by (metis  $\varphi_1$   $\varphi_2$  wf $\varphi_1$  wf $\varphi_2$  assms(1) no-equiv
  no-equiv-eq(1) no-equiv-symb.elims(3) no-equiv-symb-conn-characterization wf-conn-list(4,5)
  no-equiv-def subformula-all-subformula-st)+
  have c1-imp: c1  $\neq$  CImp and c2-imp: c2  $\neq$  CImp
  using no-imp by (metis  $\varphi_1$   $\varphi_2$  all-subformula-st-decomp-explicit-imp(2,3) assms(1)
  conn.simps(5,6) l no-imp-Imp(1) no-imp-symb.elims(3) no-imp-symb-conn-characterization
  wf $\varphi_1$  wf $\varphi_2$  all-subformula-st-decomp no-imp-symb-conn-characterization)+
  have c1c: c1  $\neq$  c'
  proof
    assume c1c: c1 = c'
    then obtain  $\xi_1$   $\xi_2$  where l1: l1 = [ $\xi_1$ ,  $\xi_2$ ]
    by (metis assms(2) connective.distinct(37,39) helper-fact wf $\varphi_1$  wf-conn.simps
    wf-conn-list-decomp(1-3))
    have c-in-c'-only c c' (conn c [conn c' l1,  $\varphi_2$ ]) using c1c l only  $\varphi_1$  by auto
    moreover have not-c-in-c'-symb c c' (conn c [conn c' l1,  $\varphi_2$ ])
    using l1  $\varphi_1$  c1c l local.wf not-c-in-c'-symb-l wf $\varphi_1$  by blast
    ultimately show False using  $\varphi_1$  c1c l l1 local.wf not-c-in-c'-simp(4) wf $\varphi_1$  by blast
  qed
  then have ( $\varphi_1$  = conn c l1  $\wedge$  wf-conn c l1)  $\vee$  ( $\exists \psi_1. \varphi_1$  = FNot  $\psi_1$ )  $\vee$  simple  $\varphi_1$ 
  by (metis  $\varphi_1$  assms(1-3) c1-eq c1-imp simple.elims(3) wf $\varphi_1$  wf-conn-list(4) wf-conn-list(5-7))
  moreover {
    assume  $\varphi_1$  = conn c l1  $\wedge$  wf-conn c l1
    then have only-c-inside c  $\varphi_1$ 
    by (metis IH $\varphi_1$   $\varphi_1$  all-subformula-st-decomp-imp inc $\varphi_1$  no-equiv no-equiv-def no-imp no-imp-def
    c-in-only $\varphi_1$  only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
    subformula-all-subformula-st)
  }
  moreover {
    assume  $\exists \psi_1. \varphi_1$  = FNot  $\psi_1$ 
    then obtain  $\psi_1$  where  $\varphi_1$  = FNot  $\psi_1$  by metis
    then have only-c-inside c  $\varphi_1$ 

```

```

    by (metis all-subformula-st-def assms(1) connective.distinct(37,39) incφ1
        only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
  }
  moreover {
    assume simple φ1
    then have only-c-inside c φ1
      by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
          only-c-inside-decomp-not only-c-inside-def)
  }
  ultimately have only-c-insideφ1: only-c-inside c φ1 by metis

  have c-in-onlyφ2: c-in-c'-only c c' (conn c2 l2)
    using only l φ2 wfφ2 assms unfolding c-in-c'-only-def by auto
  have c2c: c2 ≠ c'
  proof
    assume c2c: c2 = c'
    then obtain ξ1 ξ2 where l2: l2 = [ξ1, ξ2]
      by (metis assms(2) wfφ2 wf-conn.simps connective.distinct(7,9,19,21,29,31,37,39))
    then have c-in-c'-symb c c' (conn c [φ1, conn c' l2])
      using c2c l only φ2 all-subformula-st-test-symb-true-phi unfolding c-in-c'-only-def by auto
    moreover have not-c-in-c'-symb c c' (conn c [φ1, conn c' l2])
      using assms(1) c2c l2 not-c-in-c'-symb-r wfφ2 wf-conn-helper-facts(5,6) by metis
    ultimately show False by auto
  qed
  then have (φ2 = conn c l2 ∧ wf-conn c l2) ∨ (∃ψ2. φ2 = FNot ψ2) ∨ simple φ2
    using c2-eq by (metis φ2 assms(1-3) c2-eq c2-imp simple.elims(3) wfφ2 wf-conn-list(4-7))
  moreover {
    assume φ2 = conn c l2 ∧ wf-conn c l2
    then have only-c-inside c φ2
      by (metis IHφ2 φ2 all-subformula-st-decomp incφ2 no-equiv no-equiv-def no-imp no-imp-def
          c-in-onlyφ2 only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
          subformula-all-subformula-st)
  }
  moreover {
    assume ∃ψ2. φ2 = FNot ψ2
    then obtain ψ2 where φ2 = FNot ψ2 by metis
    then have only-c-inside c φ2
      by (metis all-subformula-st-def assms(1-3) connective.distinct(38,40) incφ2
          only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
  }
  moreover {
    assume simple φ2
    then have only-c-inside c φ2
      by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
          only-c-inside-decomp-not only-c-inside-def)
  }
  ultimately have only-c-insideφ2: only-c-inside c φ2 by metis
  show ?case using l only-c-insideφ1 only-c-insideφ2 by auto
qed

```

8.5.2 Push Conjunction

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

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

unfolding *pushConj-def* **by** (*simp add: push-conn-inside-consistent*)

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

definition *and-in-or-only* **where**

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

lemma *pushConj-inv*:

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

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

and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ

shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ

using *push-conn-inside-inv* *assms* **unfolding** *pushConj-def* **by** *metis+*

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

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

assumes

no-equiv φ **and**

no-imp φ **and**

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

no-T-F-except-top-level φ **and**

simple-not φ

shows *and-in-or-only* ψ

using *assms* *push-conn-inside-full-propo-rew-step*

unfolding *pushConj-def* *and-in-or-only-def* *c-in-c'-only-def* **by** (*metis* (*no-types*))

8.5.3 Push Disjunction

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

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

unfolding *pushDisj-def* **by** (*simp* *add*: *push-conn-inside-consistent*)

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

definition *or-in-and-only* **where**

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

lemma *not-or-in-and-only-or-and[simp]*:

$\sim \text{or-in-and-only } (FOr (FAnd \psi1 \psi2) \varphi')$

unfolding *or-in-and-only-def*

by (*metis* *all-subformula-st-test-symb-true-phi* *conn.simps*(5–6) *not-c-in-c'-symb-l*

wf-conn-helper-facts(5) *wf-conn-helper-facts*(6))

lemma *pushDisj-inv*:

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

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

and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ

shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ

using *push-conn-inside-inv* *assms* **unfolding** *pushDisj-def* **by** *metis+*

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

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

assumes

no-equiv φ **and**

no-imp φ **and**
full (*propo-rew-step* *pushDisj*) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ
shows *or-in-and-only* ψ
using *assms* *push-conn-inside-full-propo-rew-step*
unfolding *pushDisj-def* *or-in-and-only-def* *c-in-c'-only-def* **by** (*metis* (*no-types*))

9 The full transformations

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

9.1.1 Definition

The normal is a super group of groups

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

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

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

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

lemma *only-c-inside-imp-grouped-by*:
assumes $c: c \neq CNot$ **and** $c': c' = CAnd \vee c' = COr$
shows *only-c-inside* c $\varphi \Rightarrow$ *grouped-by* c φ (**is** ?*O* $\varphi \Rightarrow$?*G* φ)
proof (*induct* φ *rule: propo-induct-arity*)
case (*nullary* φ x)
then show ?*G* φ **by** *auto*
next
case (*unary* ψ)
then show ?*G* (*FNot* ψ) **by** (*auto* *simp: c*)
next
case (*binary* φ $\varphi 1$ $\varphi 2$)
note *IH* $\varphi 1 = this(1)$ **and** *IH* $\varphi 2 = this(2)$ **and** $\varphi = this(3)$ **and** *only* = *this(4)*
have $\varphi\text{-conn}: \varphi = \text{conn } c$ [$\varphi 1$, $\varphi 2$] **and** *wf*: *wf-conn* c [$\varphi 1$, $\varphi 2$]


```

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

```

lemma *grouped-by-false*:

```

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

```

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

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

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

```

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

```

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

```

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

```

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

```

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

```

next

```

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

```

```

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

```

9.2 Conjunctive Normal Form

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

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

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

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

9.2.1 Full CNF transformation

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

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

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

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

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

apply *auto*

proof –

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

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

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

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

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

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

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

then have *noTB: no-T-F-except-top-level φTB*

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

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

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

then have *noNeg: simple-not φNeg*

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

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

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

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

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

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

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

simple-not \varphiDisj

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

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

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

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

qed

9.3 Disjunctive Normal Form

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

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

shows *no-equiv $\varphi \implies$ no-imp $\varphi \implies$ simple-not $\varphi \implies$ and-in-or-only $\varphi \implies$ is-disj-with-TF φ*

using *c-in-c'-only-super-grouped-by*

unfolding *is-disj-with-TF-def and-in-or-only-def c-in-c'-only-def*

by (*simp add: c-in-c'-only-def c-in-c'-only-super-grouped-by*)

definition *is-dnf :: 'a propo \Rightarrow bool where*

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

9.3.1 Full DNF transform

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

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

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

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

theorem *dnf-transformation-correction:*

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

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

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

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

10.1 Transformation

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

inductive *elimTBFull* **where**

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

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

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

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

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

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

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

The transformation is still consistent.

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

proof –

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

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

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

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

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

shows $\psi \preceq \varphi \implies \neg \text{no-T-F-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTBFull } \psi \psi'$

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

case (*nullary* φ')

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

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

next

case (*unary* ψ)

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

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

next

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

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

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

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

qed

The same applies here. We do not need the assumption, but the deep link between $\neg \text{no-T-F-except-top-level}$ φ and the existence of a rewriting step, still exists.

lemma *no-T-F-except-top-level-rew'*:

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

assumes *noTB*: $\neg \text{no-T-F-except-top-level } \varphi$

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

proof –

have *test-symb-false-nullary*:

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

by *auto*

moreover {

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

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

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

}

ultimately show *?thesis*

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

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

qed

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

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

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

shows *no-T-F-except-top-level* ψ

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

10.2 More invariants

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

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

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

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

assume *a1*: *no-T-F* φ'

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

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

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

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

then show *no-T-F* ψ' **using** *a1 a2* **by** *fastforce*

next

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

assume *rel*: *propo-rew-step elim-equiv* $\varphi \varphi'$

and *IH*: *no-T-F* $\varphi \implies \text{no-T-F } \varphi'$

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

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

{

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

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

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

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

}

moreover {

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

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

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

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

by (*metis append.simps(1) append-is-Nil-conv list.distinct(1) list.sel(3) nth-Cons-0 tl-append2*)

have $\zeta: \forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \text{no-T-F } \zeta$

using *no-T-F unfolding no-T-F-def* **using** *corr all-subformula-st-decomp* **by** *blast*

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

have $l': \xi @ \varphi' \# \xi' = [\varphi', b] \vee \xi @ \varphi' \# \xi' = [a, \varphi']$

by (*metis (no-types, hide-lams) ab append-Cons append-Nil append-Nil2 butlast.simps(2) butlast-append list.distinct(1) list.sel(3)*)

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

moreover

```

have  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \zeta \neq FT \wedge \zeta \neq FF$ 
  using  $\zeta \text{ corr no-}T\text{-}F \text{ no-}T\text{-}F\text{-except-top-level-false no-}T\text{-}F\text{-no-}T\text{-}F\text{-except-top-level}$  by blast
then have  $\text{no-}T\text{-}F\text{-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ 
  by (metis  $\varphi' l'$  ab all-subformula-st-test-symb-true-phi  $c \text{ list.distinct}(1)$ 
     $\text{list.set-intros}(1,2)$   $\text{no-}T\text{-}F\text{-symb-except-toplevel-bin-decom}$ 
     $\text{no-}T\text{-}F\text{-symb-except-toplevel-no-}T\text{-}F\text{-symb no-}T\text{-}F\text{-symb-false}(1,2)$   $\text{no-}T\text{-}F\text{-def wf-conn-binary}$ 
     $\text{wf-conn-list}(1,2))$ 
ultimately have  $\text{no-}T\text{-}F (\text{conn } c (\xi @ \varphi' \# \xi'))$ 
  by (metis  $l'$  all-subformula-st-decomp-imp  $c \text{ no-}T\text{-}F\text{-def wf-conn-binary}$ )
}
moreover {
  fix  $x$ 
  assume  $c = CVar\ x \vee c = CF \vee c = CT$ 
  then have False using corr by auto
  then have  $\text{no-}T\text{-}F (\text{conn } c (\xi @ \varphi' \# \xi'))$  by auto
}
ultimately show  $\text{no-}T\text{-}F (\text{conn } c (\xi @ \varphi' \# \xi'))$  using corr wf-conn.cases by metis
qed

```

lemma *elim-equiv-inv'*:

```

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

```

```

have no-T-F-symb-except-toplevel (conn c (ξ @ ζ' # ξ'))
proof
  have p: no-T-F-symb (conn c (ξ @ ζ # ξ'))
    using corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F
    by blast
  have l: ∀ φ ∈ set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
    using corr wf-conn-no-T-F-symb-iff p by blast
  from rel incl have ζ' ≠ FT ∧ ζ' ≠ FF
    apply (induction ζ ζ' rule: propo-rew-step.induct)
    apply (cases rule: elim-equiv.cases, auto simp: elim-equiv.simps)
    by (metis append-is-Nil-conv list.distinct wf-conn-list(1,2) wf-conn-no-arity-change
        wf-conn-no-arity-change-helper)+
  then have ∀ φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
  moreover have c ≠ CT ∧ c ≠ CF using corr by auto
  ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
    by (metis corr wf-conn-no-arity-change wf-conn-no-arity-change-helper no-T-F-symb-comp)
qed
}
ultimately show no-T-F-except-top-level ψ
  using full-propo-rew-step-inv-stay-with-inc[of elim-equiv no-T-F-symb-except-toplevel φ]
  assms subformula-refl unfolding no-T-F-except-top-level-def by metis
qed

lemma propo-rew-step-ElimImp-no-T-F: propo-rew-step elim-imp φ ψ ⇒ no-T-F φ ⇒ no-T-F ψ
proof (induct rule: propo-rew-step.induct)
  case (global-rel φ' ψ')
  then show no-T-F ψ'
    using elim-imp.cases no-T-F-comp-not no-T-F-decomp(1,2)
    by (metis no-T-F-comp-expanded-explicit(2))
next
  case (propo-rew-one-step-lift φ φ' c ξ ξ')
  note rel = this(1) and IH = this(2) and corr = this(3) and no-T-F = this(4)
  {
    assume c: c = CNot
    then have empty: ξ = [] ξ' = [] using corr by auto
    then have no-T-F φ using no-T-F c no-T-F-decomp-not by auto
    then have no-T-F (conn c (ξ @ φ' # ξ')) using c empty no-T-F-comp-not IH by auto
  }
  moreover {
    assume c: c ∈ binary-connectives
    then obtain a b where ab: ξ @ φ # ξ' = [a, b]
      using corr list-length2-decomp wf-conn-bin-list-length by metis
    then have φ: φ = a ∨ φ = b
      by (metis append-self-conv2 wf-conn-list-decomp(4) wf-conn-unary list.discI list.sel(3)
          nth-Cons-0 tl-append2)
    have ζ: ∀ ζ ∈ set (ξ @ φ # ξ'). no-T-F ζ using ab c propo-rew-one-step-lift.prem by auto

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

```



```

  have no-T-F (last (ξ @ φ' # ξ')) by (simp add: calculation)
  then have no-T-F-symb (conn c (ξ @ φ' # ξ'))
    by (metis χ φ' ζ ab all-subformula-st-test-symb-true-phi c last.simps list.distinct(1)
        list.set-intros(1) no-T-F-bin-decomp no-T-F-def)
  ultimately have no-T-F (conn c (ξ @ φ' # ξ')) using c χ by fastforce
}
moreover {
  fix x
  assume c = CVar x ∨ c = CF ∨ c = CT
  then have False using corr by auto
  then have no-T-F (conn c (ξ @ φ' # ξ')) by auto
}
ultimately show no-T-F (conn c (ξ @ φ' # ξ')) using corr wf-conn.cases by blast
qed

```

lemma *elim-imp-inv'*:

```

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

```

```

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

  have l: ∀ φ ∈ set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
    using corr wf-conn-no-T-F-symb-iff p by blast
  from rel incl have ζ' ≠ FT ∧ ζ' ≠ FF
    apply (induction ζ ζ' rule: propo-rew-step.induct)
    apply (cases rule: elim-imp.cases, auto)
    using wf-conn-list(1,2) wf-conn-no-arity-change wf-conn-no-arity-change-helper
    by (metis append-is-Nil-conv list.distinct(1))+
  then have ∀ φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
  moreover have c ≠ CT ∧ c ≠ CF using corr by auto
  ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
    using corr wf-conn-no-arity-change no-T-F-symb-comp
    by (metis wf-conn-no-arity-change-helper)
qed
}
ultimately show no-T-F-except-top-level ψ
  using full-propo-rew-step-inv-stay-with-inc[of elim-imp no-T-F-symb-except-toplevel φ]
  assms subformula-refl unfolding no-T-F-except-top-level-def by metis
qed

```

10.3 The new CNF and DNF transformation

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

definition *dnf-rew'* :: 'a propo ⇒ 'a propo ⇒ bool **where**

```

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

```

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

```

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

```

theorem *cnf-transformation-correction:*

```

  dnf-rew' φ φ' ⇒ is-dnf φ'
unfolding dnf-rew'-def OO-def
by (meson and-in-or-only-conjunction-in-disj elimTBFULL-full-propo-rew-step elim-equiv-inv'
  elim-imp-inv elim-imp-inv' is-dnf-def no-equiv-full-propo-rew-step-elim-equiv
  no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv(1-4)
  pushNeg-full-propo-rew-step pushNeg-inv(1-3))

```

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

definition *cnf-rew'* :: 'a propo ⇒ 'a propo ⇒ bool **where**

```

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

```

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

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

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

end

11 Partial Clausal Logic

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

11.1 Clauses

Clauses are (finite) multisets of literals.

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

11.2 Partial Interpretations

type-synonym 'a interp = 'a literal set

definition *true-lit :: 'a interp \Rightarrow 'a literal \Rightarrow bool (infix \models_l 50) where*
 $I \models_l L \longleftrightarrow L \in I$

declare *true-lit-def[simp]*

11.2.1 Consistency

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

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

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

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

lemma *consistent-interp-change-insert*:

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

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

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

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

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

11.2.2 Atoms

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

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

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

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

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

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

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

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

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

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

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

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

lemma *atms-of-ms-mono*:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *in-atms-of-s-decomp*[iff]:
 $P \in \text{atms-of-s } I \iff (Pos\ P \in I \vee Neg\ P \in I)$ (is ?P \iff ?Q)
proof
 assume ?P
 then show ?Q **unfolding** *atms-of-s-def* **by** (metis image-iff literal.exhaust-sel)
next
 assume ?Q
 then show ?P **unfolding** *atms-of-s-def* **by** force
qed

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

11.2.3 Totality

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

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

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

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

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

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

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

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

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

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

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

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

lemma *total-over-m-extension:*

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

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

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

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

proof $-$

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

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

moreover have *total-over-m (I \cup ?I') (A \cup B)*

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

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

qed

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

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

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

and *cons: consistent-interp I*

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

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

proof $-$

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

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

moreover have *total-over-m (I \cup ?I') (A \cup B)*

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

moreover have *consistent-interp (I \cup ?I')*

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

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

qed

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

total-over-set Ia (atms-of-s Ia)

unfolding *total-over-set-def atms-of-s-def* **by** *(metis image-iff literal.exhaust-sel)*

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

assumes $\{\#A\# \} + D \in \psi_s$

and *total-over-set I (atms-of-ms ψ_s)*

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

using *assms unfolding total-over-set-def* **by** *(metis (no-types) Neg-atm-of-iff in-m-in-literals*

literal.collapse(1) uminus-Neg uminus-Pos)

lemma *tot-over-m-remove:*

assumes *total-over-m (I \cup {L}) { ψ }*

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

shows *total-over-m I { ψ }*

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

proof

fix l

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

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

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

moreover have $\text{atm-of } L \notin \text{atms-of-ms } \{\psi\}$

proof *(rule ccontr)*

assume $\neg ?thesis$

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

lemma *total-union*:

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

lemma *total-union-2*:

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

11.2.4 Interpretations

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

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

lemma *true-cls-singleton[iff]*: $I \models \{\#L\# \} \longleftrightarrow I \models L$
 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]*: $I \models \text{replicate-mset } n L \longleftrightarrow n \neq 0 \wedge I \models L$
by (*induct n*) *auto*

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

by (auto simp add: true-cls-def)

lemma *true-cls-not-in-remove*:
 assumes $L \notin \chi$
 and $I \cup \{L\} \models \chi$
 shows $I \models \chi$
 using *assms* **unfolding** *true-cls-def* **by** *auto*

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

lemma *true-clss-empty[simp]*: $I \models_s \{\}$
unfolding *true-clss-def* **by** *blast*

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

lemma *true-clss-empty-entails-empty[iff]*: $\{\} \models_s N \longleftrightarrow N = \{\}$
unfolding *true-clss-def* **by** (auto simp add: *true-cls-def*)

lemma *true-cls-insert-l [simp]*:
 $M \models A \implies \text{insert } L \ M \models A$
unfolding *true-cls-def* **by** *auto*

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

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

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

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

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

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

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

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

lemma *notin-vars-union-true-cls-true-cls*:
 assumes $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-ms } A$

and $\text{atms-of } L \subseteq \text{atms-of-ms } A$
and $I \cup I' \models L$
shows $I \models L$
using *assms* **unfolding** *true-cls-def true-lit-def Bex-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 $CC \equiv \exists I. (I \models_s CC \wedge \text{consistent-interp } I \wedge \text{total-over-m } I \text{ } CC)$

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

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

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

$I\text{-}CC: I \models_s CC$ **and**

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

tot: *total-over-m* $I \text{ } CC$

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

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

have $I\text{-}CC: ?I \models_s CC$

using $I\text{-}CC$ *in-implies-atm-of-on-atms-of-ms* **unfolding** *true-clss-def Ball-def true-cls-def Bex-mset-def true-lit-def*

by *blast*

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

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

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

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

moreover

have *atms-CC-incl*: *atms-of-ms CC* \subseteq *atm-of* *I*
using *tot unfolding total-over-m-def total-over-set-def atms-of-ms-def*
by (*auto simp add: atms-of-def atms-of-s-def[symmetric]*)
have *atm-of* ‘*?I = 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*: $\text{set-mset } DD \subseteq \text{set-mset } CC \Longrightarrow I \models_m CC \Longrightarrow I \models_m DD$
unfolding *true-cls-mset-def subset-iff* **by** *auto*

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

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

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

theorem *true-clss-remove-unused*:
assumes $I \models_s \psi$
shows $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\} \models_s \psi$
unfolding *true-clss-def atms-of-def Ball-def*
proof (*intro allI impI*)
fix *x*
assume $x \in \psi$
then have $I \models x$
using *assms unfolding true-clss-def atms-of-def Ball-def* **by** *auto*

then have $\{v \in I. \text{atm-of } v \in \text{atms-of } x\} \models x$
by (*simp only: true-cls-remove-unused[of I]*)
moreover have $\{v \in I. \text{atm-of } v \in \text{atms-of } x\} \subseteq \{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\}$
using $\langle x \in \psi \rangle$ **by** (*auto simp add: atms-of-ms-def*)
ultimately show $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\} \models x$
using *true-cls-mono-set-mset-l* **by** *blast*

qed

A simple application of the previous theorem:

lemma *true-clss-union-decrease*:

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

proof –

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

qed

lemma *multiset-not-empty*:

assumes $M \neq \{\#\}$
and $x \in\# M$
shows $\exists A. x = \text{Pos } A \vee x = \text{Neg } A$
using *assms literal.exhaust-sel* **by** *blast*

lemma *atms-of-ms-empty*:

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

lemma *consistent-interp-disjoint*:

assumes $\text{consI}: \text{consistent-interp } I$
and $\text{disj}: \text{atms-of-s } A \cap \text{atms-of-s } I = \{\}$
and $\text{consA}: \text{consistent-interp } A$
shows $\text{consistent-interp } (A \cup I)$

proof (*rule ccontr*)

assume $\neg ?thesis$
moreover have $\bigwedge L. \neg (L \in A \wedge \neg L \in I)$
using *disj unfolding atms-of-s-def* **by** (*auto simp add: rev-image-eqI*)
ultimately show *False*
using *consA consI unfolding consistent-interp-def* **by** (*metis (full-types) Un-iff literal.exhaust-sel uminus-Neg uminus-Pos*)

qed

lemma *total-remove-unused*:

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

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

assumes $\text{insert } a \ M' \models D$
and $\text{atm-of } a \notin \text{atms-of } D$
shows $M' \models D$
using *assms* **by** (*auto simp add: atm-of-lit-in-atms-of true-clss-def*)

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

fixes $I :: 'v \text{ interp}$ **and** $K :: 'v \text{ set}$
shows $\text{total-over-set } I \ K \longleftrightarrow (\forall l \in K. l \in (\text{atm-of } I))$

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

11.2.7 Tautologies

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

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

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

shows *tautology* A

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

by (*meson* *atm-iff-pos-or-neg-lit* *true-lit-def*)

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

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

shows *tautology* A

by (*metis* *assms* *literal.exhaust* *tautology-Pos-Neg* *uminus-Neg* *uminus-Pos*)

lemma *tautology-exists-Pos-Neg*:

assumes *tautology* ψ

shows $\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi$

proof (*rule* *ccontr*)

assume $p: \neg (\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi)$

let $?I = \{-L \mid L. L \in \# \psi\}$

have *total-over-set* $?I$ (*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** (*rename-tac* $x L$, *case-tac* L) *fastforce*+

ultimately show *False* **using** *assms* **unfolding** *tautology-def* **by** *auto*

qed

lemma *tautology-decomp*:

tautology $\psi \longleftrightarrow (\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi)$

using *tautology-exists-Pos-Neg* **by** *auto*

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

unfolding *tautology-def* **by** *auto*

lemma *tautology-add-single*:

tautology $(\{\#a\} + L) \longleftrightarrow \text{tautology } L \vee -a \in \# L$

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

lemma *minus-interp-tautology*:

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

shows *tautology* χ

proof –

obtain L **where** $L \in \# \chi \wedge -L \in \# \chi$

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

then show *?thesis* **using** *tautology-decomp* *literal.exhaust* *uminus-Neg* *uminus-Pos* **by** *metis*

qed

lemma *remove-literal-in-model-tautology*:

assumes $I \cup \{\text{Pos } P\} \models \varphi$

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

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

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

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

11.2.8 Entailment for clauses and propositions

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

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

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

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

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

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

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

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

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

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

lemma *true-clss-clss-empty[simp]*:

$N \models_{ps} \{\}$
unfolding *true-clss-clss-def* **by** *auto*

lemma *true-clss-clss-subset*:
 $A \subseteq B \implies A \models_p CC \implies B \models_p CC$
unfolding *true-clss-clss-def total-over-m-union* **by** (*simp add: total-over-m-subset true-clss-mono*)

lemma *true-clss-clss-mono-l[simp]*:
 $A \models_p CC \implies A \cup B \models_p CC$
by (*auto intro: true-clss-clss-subset*)

lemma *true-clss-clss-mono-l2[simp]*:
 $B \models_p CC \implies A \cup B \models_p CC$
by (*auto intro: true-clss-clss-subset*)

lemma *true-clss-clss-mono-r[simp]*:
 $A \models_p CC \implies A \models_p CC + CC'$
unfolding *true-clss-clss-def total-over-m-union total-over-m-sum* **by** *blast*

lemma *true-clss-clss-mono-r'[simp]*:
 $A \models_p CC' \implies A \models_p CC + CC'$
unfolding *true-clss-clss-def total-over-m-union total-over-m-sum* **by** *blast*

lemma *true-clss-clss-union-l[simp]*:
 $A \models_{ps} CC \implies A \cup B \models_{ps} CC$
unfolding *true-clss-clss-def total-over-m-union* **by** *fastforce*

lemma *true-clss-clss-union-l-r[simp]*:
 $B \models_{ps} CC \implies A \cup B \models_{ps} CC$
unfolding *true-clss-clss-def total-over-m-union* **by** *fastforce*

lemma *true-clss-clss-in[simp]*:
 $CC \in A \implies A \models_p CC$
unfolding *true-clss-clss-def true-clss-def total-over-m-union* **by** *fastforce*

lemma *true-clss-clss-insert-l[simp]*:
 $A \models_p C \implies \text{insert } a \ A \models_p C$
unfolding *true-clss-clss-def true-clss-def* **using** *total-over-m-union*
by (*metis Un-iff insert-is-Un sup commute*)

lemma *true-clss-clss-insert-l[simp]*:
 $A \models_{ps} C \implies \text{insert } a \ A \models_{ps} C$
unfolding *true-clss-clss-def true-clss-clss-def true-clss-def* **by** *blast*

lemma *true-clss-clss-union-and[iff]*:
 $A \models_{ps} C \cup D \longleftrightarrow (A \models_{ps} C \wedge A \models_{ps} D)$
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 $\text{totAC}: \text{total-over-m } I \ (A \cup C)$

```

    and cons: consistent-interp I
    and I: I  $\models_s$  A
    then have tot: total-over-m I A and tot': total-over-m I C by auto
    obtain I' where tot': total-over-m (I  $\cup$  I') (A  $\cup$  C  $\cup$  D)
    and cons': consistent-interp (I  $\cup$  I')
    and H:  $\forall x \in I'. \text{atm-of } x \in \text{atms-of-ms } D \wedge \text{atm-of } x \notin \text{atms-of-ms } (A \cup C)$ 
      using total-over-m-consistent-extension[OF - cons, of A  $\cup$  C] tot tot' by blast
    moreover have I  $\cup$  I'  $\models_s$  A using I by simp
    ultimately have I  $\cup$  I'  $\models_s$  C  $\cup$  D using A unfolding true-clss-clss-def by auto
    then have I  $\cup$  I'  $\models_s$  C  $\cup$  D by auto
    then show I  $\models_s$  C using notin-vars-union-true-clss-true-clss[of I] H by auto
  qed
} note H = this
assume A  $\models_{ps}$  C  $\cup$  D
then show A  $\models_{ps}$  C  $\wedge$  A  $\models_{ps}$  D using H[of A] Un-commute[of C D] by metis
next
assume A  $\models_{ps}$  C  $\wedge$  A  $\models_{ps}$  D
then show A  $\models_{ps}$  C  $\cup$  D
  unfolding true-clss-clss-def by auto
qed

lemma true-clss-clss-insert[iff]:
  A  $\models_{ps}$  insert L Ls  $\longleftrightarrow$  (A  $\models_p$  L  $\wedge$  A  $\models_{ps}$  Ls)
  using true-clss-clss-union-and[of A {L} Ls] by auto

lemma true-clss-clss-subset:
  A  $\subseteq$  B  $\implies$  A  $\models_{ps}$  CC  $\implies$  B  $\models_{ps}$  CC
  by (metis subset-Un-eq true-clss-clss-union-l)

lemma union-trus-clss-clss[simp]: A  $\cup$  B  $\models_{ps}$  B
  unfolding true-clss-clss-def by auto

lemma true-clss-clss-remove[simp]:
  A  $\models_{ps}$  B  $\implies$  A  $\models_{ps}$  B - C
  by (metis Un-Diff-Int true-clss-clss-union-and)

lemma true-clss-clss-subsetE:
  N  $\models_{ps}$  B  $\implies$  A  $\subseteq$  B  $\implies$  N  $\models_{ps}$  A
  by (metis sup.orderE true-clss-clss-union-and)

lemma true-clss-clss-in-imp-true-clss-clss:
  assumes N  $\models_{ps}$  U
  and A  $\in$  U
  shows N  $\models_p$  A
  using assms mk-disjoint-insert by fastforce

lemma all-in-true-clss-clss:  $\forall x \in B. x \in A \implies A \models_{ps} B$ 
  unfolding true-clss-clss-def true-clss-def by auto

lemma true-clss-clss-left-right:
  assumes A  $\models_{ps}$  B
  and A  $\cup$  B  $\models_{ps}$  M
  shows A  $\models_{ps}$  M  $\cup$  B
  using assms unfolding true-clss-clss-def by auto

```


lemma *true-clss-clss-generalise-true-clss-clss*:

$A \cup C \models_{ps} D \implies B \models_{ps} C \implies A \cup B \models_{ps} D$

proof –

assume $a1: A \cup C \models_{ps} D$

assume $B \models_{ps} C$

then have $f2: \bigwedge M. M \cup B \models_{ps} C$

by (*meson true-clss-clss-union-l-r*)

have $\bigwedge M. C \cup (M \cup A) \models_{ps} D$

using $a1$ **by** (*simp add: Un-commute sup-left-commute*)

then show *?thesis*

using $f2$ **by** (*metis (no-types) Un-commute true-clss-clss-left-right true-clss-clss-union-and*)

qed

lemma *true-clss-clss-or-true-clss-clss-or-not-true-clss-clss-or*:

assumes $D: N \models_p D + \{\#- L\# \}$

and $C: N \models_p C + \{\#L\# \}$

shows $N \models_p D + C$

unfolding *true-clss-clss-def*

proof (*intro allI impI*)

fix I

assume *tot*: *total-over-m* $I (N \cup \{D + C\})$

and *consistent-interp* I

and $I \models_s N$

{

assume $L: L \in I \vee -L \in I$

then have *total-over-m* $I \{D + \{\#- L\# \}\}$

using *tot* **by** (*cases L*) *auto*

then have $I \models D + \{\#- L\# \}$ **using** $D \langle I \models_s N \rangle$ *tot* *consistent-interp I*

unfolding *true-clss-clss-def* **by** *auto*

moreover

have *total-over-m* $I \{C + \{\#L\# \}\}$

using L *tot* **by** (*cases L*) *auto*

then have $I \models C + \{\#L\# \}$

using $C \langle I \models_s N \rangle$ *tot* *consistent-interp I* **unfolding** *true-clss-clss-def* **by** *auto*

ultimately have $I \models D + C$ **using** \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 $?I' \models_s N$ **using** $\langle I \models_s N \rangle$ **using** *true-clss-union-increase* **by** *blast*

ultimately have $?I' \models D + \{\#- L\# \}$

using D **unfolding** *true-clss-clss-def* **by** *blast*

then have $?I' \models D$ **using** L **by** *auto*

moreover

have *total-over-set* $I (atms-of (D + C))$ **using** *tot* **by** *auto*

then have $L \notin \# D \wedge -L \notin \# D$

using L **unfolding** *total-over-set-def atms-of-def* **by** (*cases L*) *force+*

ultimately have $I \models D + C$ **unfolding** *true-clss-clss-def* **by** *auto*

}

ultimately show $I \models D + C$ **by** *blast*

qed

lemma *true-cls-union-mset*[*iff*]: $I \models C \# \cup D \longleftrightarrow I \models C \vee I \models D$
unfolding *true-cls-def* **by** *force*

lemma *true-clss-cls-union-mset-true-clss-cls-or-not-true-clss-cls-or*:

assumes $D: N \models_p D + \{\# - L\# \}$

and $C: N \models_p C + \{\# L\# \}$

shows $N \models_p D \# \cup C$

unfolding *true-clss-cls-def*

proof (*intro allI impI*)

fix I

assume

tot: *total-over-m* $I (N \cup \{D \# \cup C\})$ **and**

consistent-interp I **and**

$I \models_s N$

{

assume $L: L \in I \vee -L \in I$

then have *total-over-m* $I \{D + \{\# - L\# \}\}$

using *tot* **by** (*cases* L) *auto*

then have $I \models D + \{\# - L\# \}$

using $D \langle I \models_s N \rangle$ *tot* \langle *consistent-interp* $I \rangle$ **unfolding** *true-clss-cls-def* **by** *auto*

moreover

have *total-over-m* $I \{C + \{\# L\# \}\}$

using L *tot* **by** (*cases* L) *auto*

then have $I \models C + \{\# L\# \}$

using $C \langle I \models_s N \rangle$ *tot* \langle *consistent-interp* $I \rangle$ **unfolding** *true-clss-cls-def* **by** *auto*

ultimately have $I \models D \# \cup C$ **using** \langle *consistent-interp* $I \rangle$ **unfolding** *consistent-interp-def* **by** *auto*

}

moreover {

assume $L: L \notin I \wedge -L \notin I$

let $?I' = I \cup \{L\}$

have *consistent-interp* $?I'$ **using** $L \langle$ *consistent-interp* $I \rangle$ **by** *auto*

moreover have *total-over-m* $?I' \{D + \{\# - L\# \}\}$

using *tot* **unfolding** *total-over-m-def* *total-over-set-def* **by** (*auto simp add: atms-of-def*)

moreover have *total-over-m* $?I' N$ **using** *tot* **using** *total-union* **by** *blast*

moreover have $?I' \models_s N$ **using** $\langle I \models_s N \rangle$ **using** *true-clss-union-increase* **by** *blast*

ultimately have $?I' \models D + \{\# - L\# \}$

using D **unfolding** *true-clss-cls-def* **by** *blast*

then have $?I' \models D$ **using** L **by** *auto*

moreover

have *total-over-set* $I (atms-of (D + C))$ **using** *tot* **by** *auto*

then have $L \notin \# D \wedge -L \notin \# D$

using L **unfolding** *total-over-set-def* *atms-of-def* **by** (*cases* L) *force+*

ultimately have $I \models D \# \cup C$ **unfolding** *true-cls-def* **by** *auto*

}

ultimately show $I \models D \# \cup C$ **by** *blast*

qed

lemma *satisfiable-carac*[*iff*]:

$(\exists I. \text{consistent-interp } I \wedge I \models_s \varphi) \longleftrightarrow \text{satisfiable } \varphi \text{ (is } (\exists I. ?Q I) \longleftrightarrow ?S)$

proof

assume $?S$

then show $\exists I. ?Q I$ **unfolding** *satisfiable-def* **by** *auto*

next

```

assume  $\exists I. ?Q I$ 
then obtain  $I$  where cons: consistent-interp  $I$  and  $I: I \models_s \varphi$  by metis
let  $?I' = \{Pos\ v \mid v. v \notin \text{atms-of-}s\ I \wedge v \in \text{atms-of-}ms\ \varphi\}$ 
have consistent-interp  $(I \cup ?I')$ 
  using cons unfolding consistent-interp-def by (intro allI) (rename-tac L, case-tac L, auto)
moreover have total-over-m  $(I \cup ?I')$   $\varphi$ 
  unfolding total-over-m-def total-over-set-def by auto
moreover have  $I \cup ?I' \models_s \varphi$ 
  using  $I$  unfolding Ball-def true-clss-def true-cls-def by auto
ultimately show  $?S$  unfolding satisfiable-def by blast
qed

```

```

lemma satisfiable-carac'[simp]: consistent-interp  $I \implies I \models_s \varphi \implies \text{satisfiable } \varphi$ 
  using satisfiable-carac by metis

```

11.3 Subsumptions

lemma *subsumption-total-over-m:*

```

assumes  $A \subseteq\# B$ 
shows total-over-m  $I \{B\} \implies \text{total-over-m } I \{A\}$ 
using assms unfolding subset-mset-def total-over-m-def total-over-set-def
by (auto simp add: mset-le-exists-conv)

```

lemma *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* $\{Pos\ v \mid v. v \in \text{atms-of } D \wedge v \notin \text{atms-of } C\}$ *arbitrary: D*
rule: nat-less-induct-case)

```

case 0 note  $n = \text{this}(1)$  and  $H = \text{this}(2)$  and  $\text{incl} = \text{this}(3)$ 
then have  $\text{atms-of } D \subseteq \text{atms-of } C$  by auto
then have  $\forall I. \text{total-over-m } I \{C\} \longrightarrow \text{total-over-m } I \{D\}$ 
  unfolding total-over-m-def total-over-set-def by auto
moreover have  $\forall I. I \models C \longrightarrow I \models D$  using incl true-cls-mono-leD by blast
ultimately show  $?case$  using  $H$  by auto

```

next

```

case (Suc  $n\ D$ ) note  $IH = \text{this}(1)$  and  $\text{card} = \text{this}(2)$  and  $H = \text{this}(3)$  and  $\text{incl} = \text{this}(4)$ 
let  $?atms = \{Pos\ v \mid v. v \in \text{atms-of } D \wedge v \notin \text{atms-of } C\}$ 
have finite  $?atms$  by auto
then obtain  $L$  where  $L: L \in ?atms$ 
  using card by (metis (no-types, lifting) Collect-empty-eq card-0-eq mem-Collect-eq  

nat.simps(3))
let  $?D' = D - \text{replicate-mset } (\text{count } D\ L)\ L - \text{replicate-mset } (\text{count } D\ (-L))\ (-L)$ 
have  $\text{atms-of-}D: \text{atms-of-}ms\ \{D\} \subseteq \text{atms-of-}ms\ \{?D'\} \cup \{\text{atm-of } L\}$  by auto

```

```

{
  fix  $I$ 
  assume total-over-m  $I \{?D'\}$ 
  then have tot: total-over-m  $(I \cup \{L\}) \{D\}$ 

```

```

unfolding total-over-m-def total-over-set-def using atms-of-D by auto

assume IDL:  $I \models ?D'$ 
then have  $I \cup \{L\} \models D$  unfolding true-cls-def by force
then have  $I \cup \{L\} \models \varphi$  using H tot by auto

moreover
  have tot': total-over-m ( $I \cup \{-L\}$ )  $\{D\}$ 
    using tot unfolding total-over-m-def total-over-set-def by auto
  have  $I \cup \{-L\} \models D$  using IDL unfolding true-cls-def by force
  then have  $I \cup \{-L\} \models \varphi$  using H tot' by auto
ultimately have  $I \models \varphi \vee \text{tautology } \varphi$ 
  using L remove-literal-in-model-tautology by force
} note  $H' = \text{this}$ 

have  $L \notin\# C$  and  $-L \notin\# C$  using L atm-iff-pos-or-neg-lit by force+
then have  $C\text{-in-}D'$ :  $C \subseteq\# ?D'$  using  $\langle C \subseteq\# D \rangle$  by (auto simp add: subseteq-mset-def)
have  $\text{card } \{Pos\ v \mid v. v \in \text{atms-of } ?D' \wedge v \notin \text{atms-of } C\} <$ 
   $\text{card } \{Pos\ v \mid v. v \in \text{atms-of } D \wedge v \notin \text{atms-of } C\}$ 
  using L by (auto intro!: psubset-card-mono)
then show ?case
  using IH C-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

```

definition simple-clss :: 'v set  $\Rightarrow$  'v clause set where
  simple-clss atms =  $\{C. \text{atms-of } C \subseteq \text{atms} \wedge \neg \text{tautology } C \wedge \text{distinct-mset } C\}$ 

lemma simple-clss-empty[simp]:
  simple-clss  $\{\}$  =  $\{\{\#\}\}$ 
  unfolding simple-clss-def by auto

lemma simple-clss-insert:
  assumes  $l \notin \text{atms}$ 
  shows simple-clss (insert l atms) =
    (op +  $\{\#\text{Pos } l\#\}$ ) 'simple-clss atms)
     $\cup$  (op +  $\{\#\text{Neg } l\#\}$ ) 'simple-clss atms)
     $\cup$  simple-clss atms(is  $?I = ?U$ )
proof (standard; standard)
fix C

```

```

assume  $C \in ?I$ 
then have
   $atms: atms-of\ C \subseteq insert\ l\ atms$  and
   $taut: \neg tautology\ C$  and
   $dist: distinct-mset\ C$ 
  unfolding simple-clss-def by auto
have  $H: \bigwedge x. x \in \# C \implies atm-of\ x \in insert\ l\ atms$ 
  using atm-of-lit-in-atms-of atms by blast
consider
  (Add)  $L \text{ where } L \in \# C \text{ and } L = Neg\ l \vee L = Pos\ l$ 
  | (No)  $Pos\ l \notin \# C\ Neg\ l \notin \# C$ 
  by auto
then show  $C \in ?U$ 
proof cases
  case Add
    then have  $L \notin \# C - \{\#L\# \}$ 
    using dist unfolding distinct-mset-def by auto
    moreover have  $-L \notin \# C$ 
    using taut Add by auto
    ultimately have  $atms-of\ (C - \{\#L\# \}) \subseteq atms$ 
    using atms Add by (auto simp: atm-iff-pos-or-neg-lit split: split-if-asm dest!: H)

    moreover have  $\neg tautology\ (C - \{\#L\# \})$ 
    using taut by (metis Add(1) insert-DiffM tautology-add-single)
    moreover have  $distinct-mset\ (C - \{\#L\# \})$ 
    using dist by auto
    ultimately have  $(C - \{\#L\# \}) \in simple-clss\ atms$ 
    using Add unfolding simple-clss-def by auto
    moreover have  $C = \{\#L\# \} + (C - \{\#L\# \})$ 
    using Add by (auto simp: multiset-eq-iff)
    ultimately show ?thesis using Add by auto
  next
    case No
    then have  $C \in simple-clss\ atms$ 
    using taut atms dist unfolding simple-clss-def
    by (auto simp: atm-iff-pos-or-neg-lit split: split-if-asm dest!: H)
    then show ?thesis by blast
  qed
next
fix  $C$ 
assume  $C \in ?U$ 
then consider
  (Add)  $L\ C' \text{ where } C = \{\#L\# \} + C' \text{ and } C' \in simple-clss\ atms \text{ and }$ 
   $L = Pos\ l \vee L = Neg\ l$ 
  | (No)  $C \in simple-clss\ atms$ 
  by auto
then show  $C \in ?I$ 
proof cases
  case No
    then show ?thesis unfolding simple-clss-def by auto
  next
    case (Add L C') note  $C' = this(1)$  and  $C = this(2)$  and  $L = this(3)$ 
    then have
       $atms: atms-of\ C' \subseteq atms$  and
       $taut: \neg tautology\ C'$  and

```

```

    dist: distinct-mset C'
    unfolding simple-clss-def by auto
  have atms-of C  $\subseteq$  insert l atms
    using atms C' L by auto
  moreover have  $\neg$  tautology C
    using taut C' L by (metis assms atm-of-lit-in-atms-of atms literal.sel(1,2) subset-eq
      tautology-add-single uminus-Neg uminus-Pos)
  moreover have distinct-mset C
    using dist C' L
    by (metis assms atm-of-lit-in-atms-of atms contra-subsetD distinct-mset-add-single
      literal.sel(1,2))
  ultimately show ?thesis unfolding simple-clss-def by blast
qed
qed

lemma simple-clss-finite:
  fixes atms :: 'v set
  assumes finite atms
  shows finite (simple-clss atms)
  using assms by (induction rule: finite-induct) (auto simp: simple-clss-insert)

lemma simple-clssE:
  assumes
    x  $\in$  simple-clss atms
  shows atms-of x  $\subseteq$  atms  $\wedge$   $\neg$ tautology x  $\wedge$  distinct-mset x
  using assms unfolding simple-clss-def by auto

lemma cls-in-simple-clss:
  shows {#}  $\in$  simple-clss s
  unfolding simple-clss-def by auto

lemma simple-clss-card:
  fixes atms :: 'v set
  assumes finite atms
  shows card (simple-clss atms)  $\leq$  (3::nat)  $\wedge$  (card atms)
  using assms
proof (induct atms rule: finite-induct)
  case empty
  then show ?case by auto
next
  case (insert l C) note fin = this(1) and l = this(2) and IH = this(3)
  have notin:
     $\wedge C'. \{ \#Pos\ l \# \} + C' \notin \text{simple-clss } C$ 
     $\wedge C'. \{ \#Neg\ l \# \} + C' \notin \text{simple-clss } C$ 
    using l unfolding simple-clss-def by auto
  have H:  $\wedge C' D. \{ \#Pos\ l \# \} + C' = \{ \#Neg\ l \# \} + D \implies D \in \text{simple-clss } C \implies \text{False}$ 
  proof -
    fix C' D
    assume C'D:  $\{ \#Pos\ l \# \} + C' = \{ \#Neg\ l \# \} + D$  and D:  $D \in \text{simple-clss } C$ 
    then have Pos l  $\in$  # D by (metis insert-noteq-member literal.distinct(1) union-commute)
    then have l  $\in$  atms-of D
      by (simp add: atm-iff-pos-or-neg-lit)
    then show False using D l unfolding simple-clss-def by auto
  qed
  let ?P = (op + {#Pos l#})  $\circ$  (simple-clss C)

```

```

let ?N = (op + {#Neg l#}) ‘ (simple-clss C)
let ?O = simple-clss C
have card (?P ∪ ?N ∪ ?O) = card (?P ∪ ?N) + card ?O
  apply (subst card-Un-disjoint)
  using l fin by (auto simp: simple-clss-finite notin)
moreover have card (?P ∪ ?N) = card ?P + card ?N
  apply (subst card-Un-disjoint)
  using l fin H by (auto simp: simple-clss-finite notin)
moreover
  have card ?P = card ?O
    using inj-on-iff-eq-card[of ?O op + {#Pos l#}]
    by (auto simp: fin simple-clss-finite inj-on-def)
  moreover have card ?N = card ?O
    using inj-on-iff-eq-card[of ?O op + {#Neg l#}]
    by (auto simp: fin simple-clss-finite inj-on-def)
  moreover have (3::nat) ^ card (insert l C) = 3 ^ (card C) + 3 ^ (card C) + 3 ^ (card C)
    using l by (simp add: fin mult-2-right numeral-3-eq-3)
  ultimately show ?case using IH l by (auto simp: simple-clss-insert)
qed

```

```

lemma simple-clss-mono:
  assumes incl: atms ⊆ atms'
  shows simple-clss atms ⊆ simple-clss atms'
  using assms unfolding simple-clss-def by auto

```

```

lemma distinct-mset-not-tautology-implies-in-simple-clss:
  assumes distinct-mset χ and ¬tautology χ
  shows χ ∈ simple-clss (atms-of χ)
  using assms unfolding simple-clss-def by auto

```

```

lemma simplified-in-simple-clss:
  assumes distinct-mset-set ψ and ∀ χ ∈ ψ. ¬tautology χ
  shows ψ ⊆ simple-clss (atms-of-ms ψ)
  using assms unfolding simple-clss-def
  by (auto simp: distinct-mset-set-def atms-of-ms-def)

```

11.6 Experiment: Expressing the Entailments as Locales

```

locale entail =
  fixes entail :: 'a set ⇒ 'b ⇒ bool (infix |=e 50)
  assumes entail-insert[simp]: I ≠ {} ⇒ insert L I |=e x ⟷ {L} |=e x ∨ I |=e x
  assumes entail-union[simp]: I |=e A ⇒ I ∪ I' |=e A
begin

```

```

definition entails :: 'a set ⇒ 'b set ⇒ bool (infix |=es 50) where
  I |=es A ⟷ (∀ a ∈ A. I |=e a)

```

```

lemma entails-empty[simp]:
  I |=es {}
  unfolding entails-def by auto

```

```

lemma entails-single[iff]:
  I |=es {a} ⟷ I |=e a
  unfolding entails-def by auto

```

```

lemma entails-insert-l[simp]:

```

$M \models_{es} A \implies \text{insert } L \ M \models_{es} A$
unfolding entails-def **by** (metis Un-commute entail-union insert-is-Un)

lemma entails-union[iff]: $I \models_{es} CC \cup DD \longleftrightarrow I \models_{es} CC \wedge I \models_{es} DD$
unfolding entails-def **by** blast

lemma entails-insert[iff]: $I \models_{es} \text{insert } C \ DD \longleftrightarrow I \models_e C \wedge I \models_{es} DD$
unfolding entails-def **by** blast

lemma entails-insert-mono: $DD \subseteq CC \implies I \models_{es} CC \implies I \models_{es} DD$
unfolding entails-def **by** blast

lemma entails-union-increase[simp]:
assumes $I \models_{es} \psi$
shows $I \cup I' \models_{es} \psi$
using assms **unfolding** entails-def **by** auto

lemma true-clss-commute-l:
 $(I \cup I' \models_{es} \psi) \longleftrightarrow (I' \cup I \models_{es} \psi)$
by (simp add: Un-commute)

lemma entails-remove[simp]: $I \models_{es} N \implies I \models_{es} \text{Set.remove } a \ N$
by (simp add: entails-def)

lemma entails-remove-minus[simp]: $I \models_{es} N \implies I \models_{es} N - A$
by (simp add: entails-def)

end

interpretation true-cls: entail true-cls
by standard (auto simp add: true-cls-def)

11.7 Entailment to be extended

definition true-clss-ext :: 'a literal set \Rightarrow 'a literal multiset set \Rightarrow bool (**infix** \models_{sext} 49)
where
 $I \models_{sext} N \longleftrightarrow (\forall J. I \subseteq J \longrightarrow \text{consistent-interp } J \longrightarrow \text{total-over-m } J \ N \longrightarrow J \models_s N)$

lemma true-clss-imp-true-cls-ext:
 $I \models_s N \implies I \models_{sext} N$
unfolding true-clss-ext-def **by** (metis sup.orderE true-clss-union-increase')

lemma true-clss-ext-decrease-right-remove-r:
assumes $I \models_{sext} N$
shows $I \models_{sext} N - \{C\}$
unfolding true-clss-ext-def
proof (intro allI impI)
fix J
assume
 $I \subseteq J$ **and**
 $\text{cons: consistent-interp } J$ **and**
 $\text{tot: total-over-m } J \ (N - \{C\})$
let $?J = J \cup \{\text{Pos } (\text{atm-of } P) | P. P \in\# C \wedge \text{atm-of } P \notin \text{atm-of } J\}$
have $I \subseteq ?J$ **using** $\langle I \subseteq J \rangle$ **by** auto
moreover **have** consistent-interp $?J$
using cons **unfolding** consistent-interp-def **apply** (intro allI)


```

  by (rename-tac L, case-tac L) (fastforce simp add: image-iff)+
moreover have total-over-m ?J N
  using tot unfolding total-over-m-def total-over-set-def atms-of-ms-def
  apply clarify
  apply (rename-tac l a, case-tac a ∈ N - {C})
  apply auto[]
  using atms-of-s-def atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
  by (fastforce simp: atms-of-def)
ultimately have ?J ⊨s N
  using assms unfolding true-clss-ext-def by blast
then have ?J ⊨s N - {C} by auto
have {v ∈ ?J. atm-of v ∈ atms-of-ms (N - {C})} ⊆ J
  using tot unfolding total-over-m-def total-over-set-def
  by (auto intro!: rev-image-eqI)
then show J ⊨s N - {C}
  using true-clss-remove-unused[OF ⟨?J ⊨s N - {C}⟩] unfolding true-clss-def
  by (meson true-clss-mono-set-mset-l)
qed

```

```

lemma consistent-true-clss-ext-satisfiable:
  assumes consistent-interp I and I ⊨sext A
  shows satisfiable A
  by (metis Un-empty-left assms satisfiable-carac subset-Un-eq sup.left-idem
    total-over-m-consistent-extension total-over-m-empty true-clss-ext-def)

```

```

lemma not-consistent-true-clss-ext:
  assumes ¬consistent-interp I
  shows I ⊭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 ⇒ '*v* clauses ⇒ bool **for** *N* :: '*v* clause set **where**

tautology-deletion:

$(A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}) \in N \implies simplify\ N\ (N - \{A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\})$

condensation:

$(A + \{\#L\# \} + \{\#L\# \}) \in N \implies simplify\ N\ (N - \{A + \{\#L\# \} + \{\#L\# \}\} \cup \{A + \{\#L\# \}\})$

subsumption:

$A \in N \implies A \subset\# B \implies B \in N \implies simplify\ N\ (N - \{B\})$

lemma *simplify-preserves-un-sat'*:

fixes *N N'* :: '*v* clauses

assumes *simplify N N'*

and *total-over-m I N*

shows $I \models_s N' \longrightarrow I \models_s N$

using *assms*

proof (*induct rule: simplify.induct*)

case (*tautology-deletion A P*)

```

then have  $I \models A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}$ 
  by (metis total-over-m-def total-over-set-literal-defined true-cls-singleton true-cls-union
    true-lit-def uminus-Neg union-commute)
then show ?case by (metis Un-Diff-cancel2 true-clss-singleton true-clss-union)
next
case (condensation A P)
then show ?case by (metis Diff-insert-absorb Set.set-insert insertE true-cls-union true-clss-def
  true-clss-singleton true-clss-union)
next
case (subsumption A B)
have  $A \neq B$  using subsumption.hyps(2) by auto
then have  $I \models_s N - \{B\} \implies I \models A$  using  $\langle A \in N \rangle$  by (simp add: true-clss-def)
moreover have  $I \models A \implies I \models B$  using  $\langle A <_{\#} B \rangle$  by auto
ultimately show ?case by (metis insert-Diff-single true-clss-insert)
qed

```

```

lemma simplify-preserves-un-sat:
  fixes  $N\ N' :: 'v\ clauses$ 
  assumes simplify  $N\ N'$ 
  and total-over-m  $I\ N$ 
  shows  $I \models_s N \longrightarrow I \models_s N'$ 
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce+

```

```

lemma simplify-preserves-un-sat'':
  fixes  $N\ N' :: 'v\ clauses$ 
  assumes simplify  $N\ N'$ 
  and total-over-m  $I\ N'$ 
  shows  $I \models_s N \longrightarrow I \models_s N'$ 
  using assms apply (induct rule: simplify.induct)
  using true-clss-def by fastforce+

```

```

lemma simplify-preserves-un-sat-eq:
  fixes  $N\ N' :: 'v\ clauses$ 
  assumes simplify  $N\ N'$ 
  and total-over-m  $I\ N$ 
  shows  $I \models_s N \longleftrightarrow I \models_s N'$ 
  using simplify-preserves-un-sat simplify-preserves-un-sat' assms by blast

```

```

lemma simplify-preserves-finite:
  assumes simplify  $\psi\ \psi'$ 
  shows finite  $\psi \longleftrightarrow$  finite  $\psi'$ 
  using assms by (induct rule: simplify.induct, auto simp add: remove-def)

```

```

lemma 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  $A + \{\#P\# \} + \{\#P\# \} \in \psi \implies \exists x \in \psi. \text{atm-of } P \in \text{atm-of } x$ 
  by (metis Un-iff atms-of-def atms-of-plus atms-of-singleton insert-iff)
ultimately show ?case by (auto simp add: atms-of-def)
next
case (subsumption A P)
then show ?case by auto
qed

lemma rtrancpl-simplify-atms-of-ms:
  assumes rtrancpl_simplify  $\psi \ \psi'$ 
  shows  $\text{atms-of-ms } \psi' \subseteq \text{atms-of-ms } \psi$ 
  using assms apply (induct rule: rtrancpl-induct)
  apply (fastforce intro: simplify-atms-of-ms)
  using simplify-atms-of-ms by blast

lemma factoring-imp-simplify:
  assumes  $\{\#L\# \} + \{\#L\# \} + C \in N$ 
  shows  $\exists N'. \text{simplify } N \ N'$ 
proof -
  have  $C + \{\#L\# \} + \{\#L\# \} \in N$  using assms by (simp add: add.commute union-lcomm)
  from condensation[OF this] show ?thesis by blast
qed

```

12.2 Unconstrained Resolution

type-synonym $'v \text{ uncon-state} = 'v \text{ clauses}$
inductive $\text{uncon-res} :: 'v \text{ uncon-state} \Rightarrow 'v \text{ uncon-state} \Rightarrow \text{bool}$ **where**
resolution:
 $\{\#Pos \ p\# \} + C \in N \implies \{\#Neg \ p\# \} + D \in N \implies (\{\#Pos \ p\# \} + C, \{\#Neg \ p\# \} + D) \notin \text{already-used}$
 $\implies \text{uncon-res } (N) (N \cup \{C + D\})$ |
factoring: $\{\#L\# \} + \{\#L\# \} + C \in N \implies \text{uncon-res } N (N \cup \{C + \{\#L\# \}\})$

lemma *uncon-res-increasing:*
 assumes $\text{uncon-res } S \ S'$ and $\psi \in S$
 shows $\psi \in S'$
 using assms by (induct rule: uncon-res.induct) auto

lemma *rtrancpl-uncon-inference-increasing:*
 assumes $\text{rtrancpl uncon-res } S \ S'$ and $\psi \in S$
 shows $\psi \in S'$
 using assms by (induct rule: rtrancpl-induct) (auto simp add: uncon-res-increasing)

12.2.1 Subsumption

definition $\text{subsumes} :: 'a \text{ literal multiset} \Rightarrow 'a \text{ literal multiset} \Rightarrow \text{bool}$ **where**
 $\text{subsumes } \chi \ \chi' \longleftrightarrow$
 $(\forall I. \text{total-over-}m \ I \ \{\chi'\} \longrightarrow \text{total-over-}m \ I \ \{\chi\})$
 $\wedge (\forall I. \text{total-over-}m \ I \ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi')$

lemma *subsumes-refl[simp]:*
 $\text{subsumes } \chi \ \chi$
 unfolding subsumes-def by auto

lemma *subsumes-subsumption*:
assumes *subsumes* $D \chi$
and $C \subset\# D$ **and** $\neg \text{tautology } \chi$
shows *subsumes* $C \chi$ **unfolding** *subsumes-def*
using *assms* *subsumption-total-over-m* *subsumption-chained* **unfolding** *subsumes-def*
by (*blast intro!*: *subset-mset.less-imp-le*)

lemma *subsumes-tautology*:
assumes *subsumes* $(C + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}) \chi$
shows *tautology* χ
using *assms* **unfolding** *subsumes-def* **by** (*simp add: tautology-def*)

12.3 Inference Rule

type-synonym *'v state* = *'v clauses* \times (*'v clause* \times *'v clause*) *set*
inductive *inference-clause* :: *'v state* \Rightarrow *'v clause* \times (*'v clause* \times *'v clause*) *set* \Rightarrow *bool*
 (**infix** \Rightarrow_{Res} 100) **where**
resolution:
 $\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$
already-used
 $\Longrightarrow \text{inference-clause } (N, \text{already-used}) (C + D, \text{already-used} \cup \{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\}) \mid$
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow \text{inference-clause } (N, \text{already-used}) (C + \{\#L\#\}, \text{already-used})$

inductive *inference* :: *'v state* \Rightarrow *'v state* \Rightarrow *bool* **where**
inference-step: *inference-clause* S (*clause*, *already-used*)
 $\Longrightarrow \text{inference } S$ (*fst* $S \cup \{\text{clause}\}$, *already-used*)

abbreviation *already-used-inv*
 :: *'a literal multiset set* \times (*'a literal multiset* \times *'a literal multiset*) *set* \Rightarrow *bool* **where**
already-used-inv state \equiv
 $(\forall (A, B) \in \text{snd state}. \exists p. \text{Pos } p \in\# A \wedge \text{Neg } p \in\# B \wedge$
 $((\exists \chi \in \text{fst state}. \text{subsumes } \chi ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\})))$
 $\vee \text{tautology } ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))))$

lemma *inference-clause-preserves-already-used-inv*:
assumes *inference-clause* $S S'$
and *already-used-inv* S
shows *already-used-inv* (*fst* $S \cup \{\text{fst } S'\}$, *snd* S')
using *assms* **apply** (*induct rule: inference-clause.induct*)
by *fastforce*+

lemma *inference-preserves-already-used-inv*:
assumes *inference* $S S'$
and *already-used-inv* S
shows *already-used-inv* S'
using *assms*
proof (*induct rule: inference.induct*)
case (*inference-step* S *clause* *already-used*)
then show ?*case*
using *inference-clause-preserves-already-used-inv*[*of* S (*clause*, *already-used*)] **by** *simp*
qed

```

lemma rtrancp-inference-preserves-already-used-inv:
  assumes rtrancp inference S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms apply (induct rule: rtrancp-induct, simp)
  using inference-preserves-already-used-inv unfolding tautology-def by fast

lemma subsumes-condensation:
  assumes subsumes (C + {#L#} + {#L#}) D
  shows subsumes (C + {#L#}) D
  using assms unfolding subsumes-def by simp

lemma simplify-preserves-already-used-inv:
  assumes simplify N N'
  and already-used-inv (N, already-used)
  shows already-used-inv (N', already-used)
  using assms
proof (induct rule: simplify.induct)
  case (condensation C L)
  then show ?case
    using subsumes-condensation by simp fast
next
  {
    fix a:: 'a and A :: 'a set and P
    have  $(\exists x \in \text{Set.remove } a \ A. P \ x) \longleftrightarrow (\exists x \in A. x \neq a \wedge P \ x)$  by auto
  } note ex-member-remove = this
  {
    fix a a0 :: 'v clause and A :: 'v clauses and y
    assume a ∈ A and a0 ⊂# a
    then have  $(\exists x \in A. \text{subsumes } x \ y) \longleftrightarrow (\text{subsumes } a \ y \vee (\exists x \in A. x \neq a \wedge \text{subsumes } x \ y))$ 
    by auto
  } note tt2 = this
case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and inv = this(4)
show ?case
  proof (standard, standard)
    fix x a b
    assume x: x ∈ snd (N - {B}, already-used) and [simp]: x = (a, b)
    obtain p where p: Pos p ∈# a ∧ Neg p ∈# b and
      q: (∃χ∈N. subsumes χ (a - {#Pos p#} + (b - {#Neg p#})))
       $\vee \text{tautology } (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \}))$ 
    using inv x by fastforce
    consider (taut) tautology (a - {#Pos p#} + (b - {#Neg p#})) |
      (χ) χ where χ ∈ N subsumes χ (a - {#Pos p#} + (b - {#Neg p#}))
       $\neg \text{tautology } (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \}))$ 
    using q by auto
    then show
       $\exists p. \text{Pos } p \in \# a \wedge \text{Neg } p \in \# b$ 
       $\wedge ((\exists \chi \in \text{fst } (N - \{B\}, \text{already-used}). \text{subsumes } \chi (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \})))$ 
       $\vee \text{tautology } (a - \{\#Pos \ p\# \} + (b - \{\#Neg \ p\# \})))$ 
    proof cases
      case taut
      then show ?thesis using p by auto
    next
      case χ note H = this
      show ?thesis using p A AB B subsumes-subsumption[OF - AB H(3)] H(1,2) by auto

```

```

      qed
    qed
  next
  case (tautology-deletion C P)
  then show ?case apply clarify
  proof -
    fix a b
    assume C + {#Pos P#} + {#Neg P#} ∈ N
    assume already-used-inv (N, already-used)
    and (a, b) ∈ snd (N - {C + {#Pos P#} + {#Neg P#}}, already-used)
    then obtain p where
      Pos p ∈# a ∧ Neg p ∈# b ∧
      ((∃χ∈fst (N ∪ {C + {#Pos P#} + {#Neg P#}}, already-used).
        subsumes χ (a - {#Pos p#} + (b - {#Neg p#})))
        ∨ tautology (a - {#Pos p#} + (b - {#Neg p#})))
      by fastforce
    moreover have tautology (C + {#Pos P#} + {#Neg P#}) by auto
    ultimately show
      ∃p. Pos p ∈# a ∧ Neg p ∈# b
      ∧ ((∃χ∈fst (N - {C + {#Pos P#} + {#Neg P#}}, already-used).
        subsumes χ (a - {#Pos p#} + (b - {#Neg p#})))
        ∨ tautology (a - {#Pos p#} + (b - {#Neg p#})))
      by (metis (no-types) Diff-iff Un-insert-right empty-iff fst-conv insertE subsumes-tautology
        sup-bot.right-neutral)
    qed
  qed

```

lemma

factoring-satisfiable: $I \models \{\#L\# \} + \{\#L\# \} + C \longleftrightarrow I \models \{\#L\# \} + C$ **and**

resolution-satisfiable:

consistent-interp $I \implies I \models \{\#Pos p\# \} + C \implies I \models \{\#Neg p\# \} + D \implies I \models C + D$ **and**

factoring-same-vars: $\text{atms-of } (\{\#L\# \} + \{\#L\# \} + C) = \text{atms-of } (\{\#L\# \} + C)$

unfolding *true-cls-def consistent-interp-def* **by** (*fastforce split: split-if-asm*)+

lemma *inference-increasing:*

assumes *inference S S'* **and** $\psi \in \text{fst } S$

shows $\psi \in \text{fst } S'$

using *assms* **by** (*induct rule: inference.induct, auto*)

lemma *rtranclp-inference-increasing:*

assumes *rtranclp inference S S'* **and** $\psi \in \text{fst } S$

shows $\psi \in \text{fst } S'$

using *assms* **by** (*induct rule: rtranclp-induct, auto simp add: inference-increasing*)

lemma *inference-clause-already-used-increasing:*

assumes *inference-clause S S'*

shows $\text{snd } S \subseteq \text{snd } S'$

using *assms* **by** (*induct rule: inference-clause.induct, auto*)

lemma *inference-already-used-increasing:*

assumes *inference S S'*

shows $\text{snd } S \subseteq \text{snd } S'$

using *assms* **apply** (*induct rule: inference.induct*)

using *inference-clause-already-used-increasing* **by** *fastforce*

lemma *inference-clause-preserves-un-sat*:

fixes $N\ N' :: 'v\ clauses$
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* $(fst\ (fst\ S \cup \{fst\ S'\},\ snd\ S')) \subseteq \text{atms-of-ms}\ (fst\ S)$
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* $(fst\ T') \subseteq \text{atms-of-ms}\ (fst\ T)$
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* $I\ N \implies \text{total-over-m}\ I\ N'$
using *assms* *inference-preserves-atms-of-ms* **unfolding** *total-over-m-def* *total-over-set-def*
by *fastforce*

lemma *rtranclp-inference-preserves-total*:

assumes *rtranclp* *inference* $T\ T'$
shows *total-over-m* $I\ (fst\ T) \implies \text{total-over-m}\ I\ (fst\ T')$
using *assms* **by** (*induct* rule: *rtranclp-induct*, *auto* *simp* add: *inference-preserves-total*)

lemma *rtranclp-inference-preserves-un-sat*:

assumes *rtranclp* *inference* $N\ N'$
and *total-over-m* $I\ (fst\ N)$
and *consistent*: *consistent-interp* I

shows $I \models_s \text{fst } N \longleftrightarrow I \models_s \text{fst } N'$
using *assms* **apply** (*induct rule: rtranclp-induct*)
apply (*simp add: inference-preserves-un-sat*)
using *inference-preserves-un-sat rtranclp-inference-preserves-total* **by** *blast*

lemma *inference-preserves-finite*:
assumes *inference* $\psi \ \psi'$ **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
using *assms* **by** (*induct rule: inference.induct, auto simp add: simplify-preserves-finite*)

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause* $\psi \ \psi'$ **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
using *assms* **by** (*induct rule: inference-clause.induct, auto*)

lemma *inference-preserves-finite-snd*:
assumes *inference* $\psi \ \psi'$ **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
using *assms inference-clause-preserves-finite-snd* **by** (*induct rule: inference.induct, fastforce*)

lemma *rtranclp-inference-preserves-finite*:
assumes *rtranclp inference* $\psi \ \psi'$ **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
using *assms* **by** (*induct rule: rtranclp-induct*)
(auto simp add: simplify-preserves-finite inference-preserves-finite)

lemma *consistent-interp-insert*:
assumes *consistent-interp* I
and *atm-of* $P \notin \text{atm-of } I$
shows *consistent-interp* (*insert* $P \ I$)
proof –
have $P: \text{insert } P \ I = I \cup \{P\}$ **by** *auto*
show *?thesis* **unfolding** P
apply (*rule consistent-interp-disjoint*)
using *assms* **by** (*auto simp add: atms-of-s-def*)
qed

lemma *simplify-clause-preserves-sat*:
assumes *simp: simplify* $\psi \ \psi'$
and *satisfiable* ψ'
shows *satisfiable* ψ
using *assms*
proof *induction*
case (*tautology-deletion* $A \ P$) **note** $AP = \text{this}(1)$ **and** $\text{sat} = \text{this}(2)$
let $?A' = A + \{\#Pos \ P\# \} + \{\#Neg \ P\# \}$
let $? \psi' = \psi - \{?A'\}$
obtain I **where**
 $I: I \models_s ? \psi'$ **and**
cons: consistent-interp I **and**
tot: total-over-m $I \ ? \psi'$
using *sat* **unfolding** *satisfiable-def* **by** *auto*
{ assume $Pos \ P \in I \vee Neg \ P \in I$


```

then have  $I \models ?A'$  by auto
then have  $I \models \psi$  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  $\{Pos\ P\} \cup I \models \psi - \{A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\}\}$ 
    using  $\langle I \models \psi - \{A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\}\} \rangle$  true-clss-union-increase' by blast
  then have  $I \cup \{Pos\ P\} \models \psi$ 
    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 ( $\psi - \{A + \{\#L\#\} + \{\#L\#\}\} \cup \{A + \{\#L\#\}\}) \models \psi - \{A + \{\#L\#\} + \{\#L\#\}\} \cup \{A + \{\#L\#\}\}$ 
     $\wedge$  consistent-interp (LL ( $\psi - \{A + \{\#L\#\} + \{\#L\#\}\} \cup \{A + \{\#L\#\}\}$ ))
     $\wedge$  total-over-m (LL ( $\psi - \{A + \{\#L\#\} + \{\#L\#\}\} \cup \{A + \{\#L\#\}\}$ ))
  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  $? \psi' = \psi - \{B\}$ 
obtain I where I:  $I \models ? \psi'$  and cons: consistent-interp I and tot: total-over-m I  $? \psi'$ 
  using sat unfolding satisfiable-def by auto
have  $I \models A$  using A I by (metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def)
then have  $I \models B$  using AB subset-mset.less-imp-le true-clss-mono-leD by blast
then have  $I \models \psi$  using I by (metis insert-Diff-single true-clss-insert)
then show  $?case$  using cons satisfiable-carac' by blast
qed

```

```

lemma simplify-preserves-unsat:
  assumes inference  $\psi\ \psi'$ 
  shows satisfiable (fst  $\psi'$ )  $\longrightarrow$  satisfiable (fst  $\psi$ )
  using assms apply (induct rule: inference.induct)
  using satisfiable-decreasing by (metis fst-conv)+

```

```

lemma inference-preserves-unsat:
  assumes inference** S S'
  shows satisfiable (fst S')  $\longrightarrow$  satisfiable (fst S)
  using assms apply (induct rule: rtranclp-induct)

```

apply *simp-all*
using *simplify-preserves-unsat* **by** *blast*

datatype 'v *sem-tree* = *Node* 'v 'v *sem-tree* 'v *sem-tree* | *Leaf*

fun *sem-tree-size* :: 'v *sem-tree* \Rightarrow *nat* **where**
sem-tree-size *Leaf* = 0 |
sem-tree-size (*Node* - *ag* *ad*) = 1 + *sem-tree-size* *ag* + *sem-tree-size* *ad*

lemma *sem-tree-size*[*case-names bigger*]:
 $(\bigwedge xs:: 'v \text{ sem-tree. } (\bigwedge ys:: 'v \text{ sem-tree. } \text{sem-tree-size } ys < \text{sem-tree-size } xs \Rightarrow P \text{ } ys) \Rightarrow P \text{ } xs)$
 $\Rightarrow P \text{ } xs$
by (*fact* *Nat.measure-induct-rule*)

fun *partial-interps* :: 'v *sem-tree* \Rightarrow 'v *interp* \Rightarrow 'v *clauses* \Rightarrow *bool* **where**
partial-interps *Leaf* *I* ψ = $(\exists \chi. \neg I \models \chi \wedge \chi \in \psi \wedge \text{total-over-m } I \{ \chi \})$ |
partial-interps (*Node* *v* *ag* *ad*) *I* $\psi \longleftrightarrow$
 $(\text{partial-interps } ag \text{ } (I \cup \{Pos \text{ } v\}) \psi \wedge \text{partial-interps } ad \text{ } (I \cup \{Neg \text{ } v\}) \psi)$

lemma *simplify-preserve-partial-leaf*:
simplify *N* *N'* \Rightarrow *partial-interps* *Leaf* *I* *N* \Rightarrow *partial-interps* *Leaf* *I* *N'*
apply (*induct rule: simplify.induct*)
using *union-lcomm* **apply** *auto*[1]
apply (*simp, metis* *atms-of-plus total-over-set-union true-cls-union*)
apply *simp*
by (*metis* *atms-of-ms-singleton mset-le-exists-conv subset-mset-def true-cls-mono-leD*
total-over-m-def total-over-m-sum)

lemma *simplify-preserve-partial-tree*:
assumes *simplify* *N* *N'*
and *partial-interps* *t* *I* *N*
shows *partial-interps* *t* *I* *N'*
using *assms* **apply** (*induct t arbitrary: I, simp*)
using *simplify-preserve-partial-leaf* **by** *metis*

lemma *inference-preserve-partial-tree*:
assumes *inference* *S* *S'*
and *partial-interps* *t* *I* (*fst* *S*)
shows *partial-interps* *t* *I* (*fst* *S'*)
using *assms* **apply** (*induct t arbitrary: I, simp-all*)
by (*meson inference-increasing*)

lemma *rtranclp-inference-preserve-partial-tree*:
assumes *rtranclp inference* *N* *N'*
and *partial-interps* *t* *I* (*fst* *N*)
shows *partial-interps* *t* *I* (*fst* *N'*)
using *assms* **apply** (*induct rule: rtranclp-induct, auto*)
using *inference-preserve-partial-tree* **by** *force*

```

function build-sem-tree :: 'v :: linorder set  $\Rightarrow$  'v clauses  $\Rightarrow$  'v sem-tree where
build-sem-tree atms  $\psi$  =
  (if atms = {}  $\vee$   $\neg$  finite atms
   then Leaf
   else Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ ))
by auto
termination
  apply (relation measure ( $\lambda(A, -).$  card A), simp-all)
  apply (metis Min-in card-Diff1-less remove-def)+
done
declare build-sem-tree.induct[case-names tree]

lemma unsatisfiable-empty[simp]:
   $\neg$ unsatisfiable {}
  unfolding satisfiable-def apply auto
  using consistent-interp-def unfolding total-over-m-def total-over-set-def atms-of-ms-def by blast

lemma partial-interps-build-sem-tree-atms-general:
  fixes  $\psi :: 'v :: linorder$  clauses and  $p :: 'v$  literal list
  assumes unsat: unsatisfiable  $\psi$  and finite  $\psi$  and consistent-interp I
  and finite atms
  and atms-of-ms  $\psi$  = atms  $\cup$  atms-of-s I and atms  $\cap$  atms-of-s I = {}
  shows partial-interps (build-sem-tree atms  $\psi$ ) I  $\psi$ 
  using assms
proof (induct arbitrary: I rule: build-sem-tree.induct)
case (1 atms  $\psi$  Ia) note IH1 = this(1) and IH2 = this(2) and unsat = this(3) and finite = this(4)
  and cons = this(5) and f = this(6) and un = this(7) and disj = this(8)
  {
    assume atms: atms = {}
    then have atmsIa: atms-of-ms  $\psi$  = atms-of-s Ia using un by auto
    then have total-over-m Ia  $\psi$  unfolding total-over-m-def atmsIa by auto
    then have  $\chi$ :  $\exists \chi \in \psi. \neg Ia \models \chi$ 
      using unsat cons unfolding true-clss-def satisfiable-def by auto
    then have build-sem-tree atms  $\psi$  = Leaf using atms by auto
    moreover
      have tot:  $\bigwedge \chi. \chi \in \psi \implies$  total-over-m Ia  $\{\chi\}$ 
      unfolding total-over-m-def total-over-set-def atms-of-ms-def atms-of-s-def
      using atmsIa atms-of-ms-def by fastforce
    have partial-interps Leaf Ia  $\psi$ 
      using  $\chi$  tot by (auto simp add: total-over-m-def total-over-set-def atms-of-ms-def)

    ultimately have ?case by metis
  }
moreover {
  assume atms: atms  $\neq$  {}
  have build-sem-tree atms  $\psi$  = Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    using build-sem-tree.simps[of atms  $\psi$ ] f atms by metis

  have consistent-interp (Ia  $\cup$  {Pos (Min atms)}) unfolding consistent-interp-def
    by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
      f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2)
      uminus-Neg uminus-Pos)
  moreover have atms-of-ms  $\psi$  = 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  $\langle$ atms-of-ms  $\psi$  = Set.remove (Min atms) atms  $\cup$  atms-of-s (Ia  $\cup$  {Pos (Min atms)}) $\rangle$  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  $p :: 'v$  literal list
  assumes unsat: unsatisfiable  $\psi$  and finite: finite  $\psi$ 
  shows partial-interps (build-sem-tree (atms-of-ms  $\psi$ )  $\psi$ ) {}  $\psi$ 
proof -
  have consistent-interp {} unfolding consistent-interp-def by auto
  moreover have atms-of-ms  $\psi$  = atms-of-ms  $\psi$   $\cup$  atms-of-s {} unfolding atms-of-s-def by auto
  moreover have atms-of-ms  $\psi$   $\cap$  atms-of-s {} = {} unfolding atms-of-s-def by auto
  moreover have finite (atms-of-ms  $\psi$ ) unfolding atms-of-ms-def using finite by simp
  ultimately show partial-interps (build-sem-tree (atms-of-ms  $\psi$ )  $\psi$ ) {}  $\psi$ 
    using partial-interps-build-sem-tree-atms-general[of  $\psi$  {} atms-of-ms  $\psi$ ] assms by metis
qed

```

lemma can-decrease-count:

```

  fixes  $\psi'' :: 'v$  clauses  $\times$  ('v clause  $\times$  'v clause  $\times$  'v) set
  assumes count  $\chi$  L = n
  and L  $\in \# \chi$  and  $\chi \in \text{fst } \psi$ 
  shows  $\exists \psi' \chi'. \text{inference}^{**} \psi \psi' \wedge \chi' \in \text{fst } \psi' \wedge (\forall L. L \in \# \chi \longleftrightarrow L \in \# \chi')$ 
     $\wedge \text{count } \chi' L = 1$ 
     $\wedge (\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi')$ 
     $\wedge (I \models \chi \longleftrightarrow I \models \chi')$ 
     $\wedge (\forall I'. \text{total-over-m } I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi'\})$ 
  using assms
proof (induct n arbitrary:  $\chi \psi$ )
  case 0

```

```

then show ?case by simp
next
case (Suc n χ)
note IH = this(1) and count = this(2) and L = this(3) and χ = this(4)
{
  assume n = 0
  then have inference** ψ ψ
  and χ ∈ fst ψ
  and ∀ L. (L ∈# χ) ⟷ (L ∈# χ)
  and count χ L = (1::nat)
  and ∀ φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ
    by (auto simp add: count L χ)
  then have ?case by metis
}
moreover {
  assume n > 0
  then have ∃ C. χ = C + {#L, L#}
    by (metis L One-nat-def add-diff-cancel-right' count-diff count-single diff-Suc-Suc diff-zero
      local.count multi-member-split union-assoc)
  then obtain C where C: χ = C + {#L, L#} by metis
  let ?χ' = C + {#L#}
  let ?ψ' = (fst ψ ∪ {?χ'}, snd ψ)
  have φ: ∀ φ ∈ fst ψ. (φ ∈ fst ψ ∨ φ ≠ ?χ') ⟷ φ ∈ fst ?ψ' unfolding C by auto
  have inf: inference ψ ?ψ'
    using C factoring χ prod.collapse union-commute inference-step by metis
  moreover have count': count ?χ' L = n using C count by auto
  moreover have Lχ': L :# ?χ' by auto
  moreover have χ'ψ': ?χ' ∈ fst ?ψ' by auto
  ultimately obtain ψ'' and χ''
  where
    inference** ?ψ' ψ'' and
    α: χ'' ∈ fst ψ'' and
    ∀ La. (La ∈# ?χ') ⟷ (La ∈# χ'') and
    β: count χ'' L = (1::nat) and
    φ': ∀ φ. φ ∈ fst ?ψ' ⟶ φ ∈ fst ψ'' and
    Iχ: I ⊨ ?χ' ⟷ I ⊨ χ'' and
    tot: ∀ I'. total-over-m I' {?χ'} ⟶ total-over-m I' {χ''}
    using IH[of ?χ' ?ψ'] count' Lχ' χ'ψ' by blast

  then have inference** ψ ψ''
  and ∀ La. (La ∈# χ) ⟷ (La ∈# χ'')
  using inf unfolding C by auto
  moreover have ∀ φ. φ ∈ fst ψ ⟶ φ ∈ fst ψ'' using φ φ' by metis
  moreover have I ⊨ χ ⟷ I ⊨ χ'' using Iχ unfolding true-cls-def C by auto
  moreover have ∀ I'. total-over-m I' {χ} ⟶ total-over-m I' {χ''}
    using tot unfolding C total-over-m-def by auto
  ultimately have ?case using φ φ' α β by metis
}
ultimately show ?case by auto
qed

```

lemma *can-decrease-tree-size:*

```

fixes ψ :: 'v state and tree :: 'v sem-tree
assumes finite (fst ψ) and already-used-inv ψ
and partial-interps tree I (fst ψ)

```

```

shows  $\exists (tree': 'v \text{ sem-tree}) \psi'. \text{inference}^{**} \psi \psi' \wedge \text{partial-interps tree}' I (\text{fst } \psi')$ 
 $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0)$ 
using assms
proof (induct arbitrary: I rule: sem-tree-size)
  case (bigger xs I) note  $IH = \text{this}(1)$  and  $\text{finite} = \text{this}(2)$  and  $\text{a-u-i} = \text{this}(3)$  and  $\text{part} = \text{this}(4)$ 

  {
    assume  $\text{sem-tree-size xs} = 0$ 
    then have ?case using part by blast
  }

moreover {
  assume  $\text{sn0: sem-tree-size xs} > 0$ 
  obtain  $ag \text{ ad } v$  where  $xs = \text{Node } v \text{ ag ad}$  using sn0 by (cases xs, auto)
  {
    assume  $\text{sem-tree-size ag} = 0$  and  $\text{sem-tree-size ad} = 0$ 
    then have  $ag: ag = \text{Leaf}$  and  $ad: ad = \text{Leaf}$  by (cases ag, auto) (cases 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  $\text{Posv: } \neg \text{Pos } v \in \# \chi$  using  $\chi$  unfolding true-cls-def true-lit-def by auto
    have  $\text{Negv: } \neg \text{Neg } v \in \# \chi'$  using  $\chi'$  unfolding true-cls-def true-lit-def by auto
    {
      assume  $\text{Neg}\chi: \neg \text{Neg } v \in \# \chi$ 
      have  $\neg I \models \chi$  using  $\chi$  Posv unfolding true-cls-def true-lit-def by auto
      moreover have  $\text{total-over-m } I \{\chi\}$ 
        using Posv Negχ atm-imp-pos-or-neg-lit totχ 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  $\text{inference}^{**} \psi \psi$ 
      unfolding xs by (auto simp add: χψ)
    }
  }
moreover {
  assume  $\text{Pos}\chi: \neg \text{Pos } v \in \# \chi'$ 
  then have  $I\chi: \neg I \models \chi'$  using  $\chi'$  Posv unfolding true-cls-def true-lit-def by auto
  moreover have  $\text{total-over-m } I \{\chi'\}$ 
    using Negv Posχ atm-imp-pos-or-neg-lit totχ'
    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
     $\text{inference}^{**} \psi \psi$ 
    using  $\chi'\psi \text{ } I\chi$  unfolding xs by auto
  }
moreover {
  assume  $\text{neg: Neg } v \in \# \chi$  and  $\text{pos: Pos } v \in \# \chi'$ 
  then obtain  $\psi' \chi 2$  where  $\text{inf: rtrnclp inference } \psi \psi'$  and  $\chi 2 \text{incl: } \chi 2 \in \text{fst } \psi'$ 
    and  $\chi \chi 2 \text{-incl: } \forall L. L : \# \chi \longleftrightarrow L : \# \chi 2$ 
    and  $\text{count}\chi 2: \text{count } \chi 2 (\text{Neg } v) = 1$ 
  }

```

```

and  $\varphi: \forall \varphi::'v \text{ literal multiset. } \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi'$ 
and  $I\chi: I \models \chi \longleftrightarrow I \models \chi^2$ 
and  $\text{tot-imp}\chi: \forall I'. \text{total-over-m } I' \{ \chi \} \longrightarrow \text{total-over-m } I' \{ \chi^2 \}$ 
using  $\text{can-decrease-count[of } \chi \text{ Neg } v \text{ count } \chi \text{ (Neg } v) \text{ } \psi \text{ } I] \chi \psi \chi' \psi$  by auto

have  $\chi' \in \text{fst } \psi'$  by (simp add:  $\chi' \psi \varphi$ )
with pos
obtain  $\psi'' \chi^2'$  where
 $\text{inf}'$ : inference**  $\psi' \psi''$ 
and  $\chi^2'\text{-incl: } \chi^2' \in \text{fst } \psi''$ 
and  $\chi' \chi^2\text{-incl: } \forall L::'v \text{ literal. } (L \in \# \chi') = (L \in \# \chi^2')$ 
and  $\text{count}\chi^2': \text{count } \chi^2' \text{ (Pos } v) = (1::\text{nat})$ 
and  $\varphi': \forall \varphi::'v \text{ literal multiset. } \varphi \in \text{fst } \psi' \longrightarrow \varphi \in \text{fst } \psi''$ 
and  $I\chi': I \models \chi' \longleftrightarrow I \models \chi^2'$ 
and  $\text{tot-imp}\chi': \forall I'. \text{total-over-m } I' \{ \chi' \} \longrightarrow \text{total-over-m } I' \{ \chi^2' \}$ 
using  $\text{can-decrease-count[of } \chi' \text{ Pos } v \text{ count } \chi' \text{ (Pos } v) \text{ } \psi' \text{ } I]$  by auto

obtain  $C$  where  $\chi^2: \chi^2 = C + \{ \# \text{Neg } v \# \}$  and  $\text{neg}C: \text{Neg } v \notin \# C$  and  $\text{pos}C: \text{Pos } v \notin \# C$ 
by (metis (no-types, lifting) One-nat-def Posv Suc-inject Suc-pred  $\chi \chi^2\text{-incl count}\chi^2$ 
count-diff count-single gr0I insert-DiffM insert-DiffM2 multi-member-skip
old.nat.distinct(2))

obtain  $C'$  where
 $\chi^2': \chi^2' = C' + \{ \# \text{Pos } v \# \}$  and
 $\text{pos}C': \text{Pos } v \notin \# C'$  and
 $\text{neg}C': \text{Neg } v \notin \# C'$ 
proof –
assume  $a1: \bigwedge C'. \llbracket \chi^2' = C' + \{ \# \text{Pos } v \# \}; \text{Pos } v \notin \# C'; \text{Neg } v \notin \# C' \rrbracket \Longrightarrow \text{thesis}$ 
have  $f2: \bigwedge n. (n::\text{nat}) - n = 0$ 
by simp
have  $\text{Neg } v \notin \# \chi^2' - \{ \# \text{Pos } v \# \}$ 
using  $\text{Negv } \chi' \chi^2\text{-incl}$  by auto
then show ?thesis
using  $f2 \ a1$  by (metis add commute count $\chi^2'$  count-diff count-single insert-DiffM
less-nat-zero-code zero-less-one)
qed

have already-used-inv  $\psi'$ 
using rtrancplp-inference-preserves-already-used-inv[of  $\psi \psi'$  a-u-i inf] by blast
then have a-u-i- $\psi''$ : already-used-inv  $\psi''$ 
using rtrancplp-inference-preserves-already-used-inv a-u-i inf' unfolding tautology-def
by simp

have  $\text{tot}C: \text{total-over-m } I \{ C \}$ 
using  $\text{tot-imp}\chi \ \text{tot}\chi \ \text{tot-over-m-remove[of } I \text{ Pos } v \text{ } C] \ \text{neg}C \ \text{pos}C$  unfolding  $\chi^2$ 
by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
have  $\text{tot}C': \text{total-over-m } I \{ C' \}$ 
using  $\text{tot-imp}\chi' \ \text{tot}\chi' \ \text{total-over-m-sum tot-over-m-remove[of } I \text{ Neg } v \text{ } C'] \ \text{neg}C' \ \text{pos}C'$ 
unfolding  $\chi^2'$  by (metis total-over-m-sum uminus-Neg)
have  $\neg I \models C + C'$ 
using  $\chi \ I\chi \ \chi' \ I\chi' \ \text{unfolding } \chi^2 \ \chi^2' \ \text{true-cls-def Bex-mset-def}$ 
by (metis add-gr-0 count-union true-cls-singleton true-cls-union-increase)
then have part-I- $\psi'''$ : partial-interps Leaf  $I \text{ (fst } \psi'' \cup \{ C + C' \})$ 
using  $\text{tot}C \ \text{tot}C'$  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  $\text{inf}'': \text{inference } \psi'' (\text{fst } \psi'' \cup \{C + C'\}, \text{snd } \psi'' \cup \{(\chi 2', \chi 2)\})$ 
    using  $\text{add.commute } \varphi' \chi 2 \text{incl } (\chi 2' \in \text{fst } \psi'')$  unfolding  $\chi 2 \chi 2'$ 
    by  $(\text{metis prod.collapse inference-step resolution})$ 
  have  $\text{inference}^{**} \psi (\text{fst } \psi'' \cup \{C + C'\}, \text{snd } \psi'' \cup \{(\chi 2', \chi 2)\})$ 
    using  $\text{inf inf' inf'' rtranclp-trans}$  by auto
  moreover have  $\text{sem-tree-size Leaf} < \text{sem-tree-size } xs$  unfolding  $xs$  by auto
  ultimately have  $?case$  using  $\text{part-I-}\psi'''$  by  $(\text{metis fst-conv})$ 
}
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
     $\text{decomp}: ((\exists \chi \in \text{fst } \psi''.$ 
       $(\forall I. \text{total-over-m } I \{(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\}$ 
         $+ ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))$ 
         $\longrightarrow \text{total-over-m } I \{\chi\})$ 
       $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi$ 
         $\longrightarrow I \models (\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))$ 
       $)$ 
       $\vee \text{tautology } ((\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})))$ 
    using  $a$  by  $(\text{blast intro: allE[OF a-u-i-}\psi''[\text{unfolded subsumes-def Ball-def}],$ 
       $\text{of } (\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C))]$ 
    { assume  $p \neq v$ 
      then have  $Pos\ p \in \# C' \wedge Neg\ p \in \# C$  using  $p\ n$  by force
      then have  $?thesis$  by  $(\text{metis add-gr-0 count-union tautology-Pos-Neg})$ 
    }
    moreover {
      assume  $p = v$ 
      then have  $?thesis$  using  $\text{decomp}$  by  $(\text{metis add.commute add-diff-cancel-left'})$ 
    }
    ultimately show  $?thesis$  by auto
  qed
  moreover {
    assume  $\exists \chi \in \text{fst } \psi''. (\forall I. \text{total-over-m } I \{C+C'\} \longrightarrow \text{total-over-m } I \{\chi\})$ 
     $\wedge (\forall I. \text{total-over-m } I \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$ 
    then obtain  $\vartheta$  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  $\text{partial-interps Leaf } I (\text{fst } \psi'')$ 
      using  $\text{tot-}\vartheta\text{-CC' } \vartheta\text{-inv totC totC' } \hookrightarrow I \models C + C'$   $\text{total-over-m-sum}$  by fastforce
    moreover have  $\text{sem-tree-size Leaf} < \text{sem-tree-size } xs$  unfolding  $xs$  by auto
    ultimately have  $?case$  by  $(\text{metis inf inf' rtranclp-trans})$ 
  }
  moreover {
    assume  $\text{tautCC': tautology } (C' + C)$ 
    have  $\text{total-over-m } I \{C'+C\}$  using  $\text{totC totC' total-over-m-sum}$  by auto
    then have  $\neg \text{tautology } (C' + C)$ 
      using  $\hookrightarrow I \models C + C'$  unfolding  $\text{add.commute[of } C\ C']$   $\text{total-over-m-def}$ 
      unfolding  $\text{tautology-def}$  by auto
  }
}

```



```

    then have False using tautCC' unfolding tautology-def by auto
  }
  ultimately have ?case by auto
}
ultimately have ?case by auto
}
ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
}
moreover {
  assume size-ag: sem-tree-size ag > 0
  have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
  moreover have partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi$ )
    and partad: partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ag < sem-tree-size xs  $\longrightarrow$  finite (fst  $\psi$ )  $\longrightarrow$  already-used-inv  $\psi$ 
 $\longrightarrow$  ( partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi$ )  $\longrightarrow$ 
  ( $\exists$  tree'  $\psi'$ . inference**  $\psi \psi' \wedge$  partial-interps tree' (I  $\cup$  {Pos v}) (fst  $\psi'$ )
 $\wedge$  (sem-tree-size tree' < sem-tree-size ag  $\vee$  sem-tree-size ag = 0)))
    using IH by auto
  ultimately obtain  $\psi' :: 'v$  state and tree' :: 'v sem-tree where
    inf: inference**  $\psi \psi'$ 
    and part: partial-interps tree' (I  $\cup$  {Pos v}) (fst  $\psi'$ )
    and size: sem-tree-size tree' < sem-tree-size ag  $\vee$  sem-tree-size ag = 0
    using finite part rtranclp.rtrancl-refl a-u-i by blast

  have partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi'$ )
    using rtranclp-inference-preserve-partial-tree inf partad by metis
  then have partial-interps (Node v tree' ad) I (fst  $\psi'$ ) using part by auto
  then have ?case using inf size size-ag part unfolding xs by fastforce
}
moreover {
  assume size-ad: sem-tree-size ad > 0
  have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
  moreover have partag: partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi$ ) and
    partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )
    using part partial-interps.simps(2) unfolding xs by metis+
  moreover have sem-tree-size ad < sem-tree-size xs  $\longrightarrow$  finite (fst  $\psi$ )  $\longrightarrow$  already-used-inv  $\psi$ 
 $\longrightarrow$  ( partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )
 $\longrightarrow$  ( $\exists$  tree'  $\psi'$ . inference**  $\psi \psi' \wedge$  partial-interps tree' (I  $\cup$  {Neg v}) (fst  $\psi'$ )
 $\wedge$  (sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0)))
    using IH by auto
  ultimately obtain  $\psi' :: 'v$  state and tree' :: 'v sem-tree where
    inf: inference**  $\psi \psi'$ 
    and part: partial-interps tree' (I  $\cup$  {Neg v}) (fst  $\psi'$ )
    and size: sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0
    using finite part rtranclp.rtrancl-refl a-u-i by blast

  have partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi'$ )
    using rtranclp-inference-preserve-partial-tree inf partag by metis
  then have partial-interps (Node v ag tree') I (fst  $\psi'$ ) using part by auto
  then have ?case using inf size size-ad unfolding xs by fastforce
}
ultimately have ?case by auto
}
ultimately show ?case by auto

```

qed

lemma *inference-completeness-inv*:

fixes $\psi :: 'v :: \text{linorder state}$

assumes

unsat: $\neg \text{satisfiable (fst } \psi)$ **and**

finite: $\text{finite (fst } \psi)$ **and**

a-u-v: $\text{already-used-inv } \psi$

shows $\exists \psi'. (\text{inference}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$

proof –

obtain *tree* **where** $\text{partial-interps tree } \{\} (\text{fst } \psi)$

using *partial-interps-build-sem-tree-atms* **assms** **by** *metis*

then show *?thesis*

using *unsat finite a-u-v*

proof (*induct tree arbitrary: ψ rule: sem-tree-size*)

case (*bigger tree ψ*) **note** $H = \text{this}$

{

fix χ

assume $\text{tree: tree} = \text{Leaf}$

obtain χ **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** *tree* **by** *auto*

moreover have $\{\#\} = \chi$

using $\text{tot}\chi$ **unfolding** *total-over-m-def total-over-set-def* **by** *fastforce*

moreover have $\text{inference}^{**} \psi \psi$ **by** *auto*

ultimately have *?case* **by** *metis*

}

moreover {

fix $v \text{ tree1 tree2}$

assume $\text{tree: tree} = \text{Node } v \text{ tree1 tree2}$

obtain

$\text{tree}' \psi'$ **where** $\text{inf: inference}^{**} \psi \psi'$ **and**

$\text{part}': \text{partial-interps tree}' \{\} (\text{fst } \psi')$ **and**

$\text{decrease: sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0$

using *can-decrease-tree-size[of ψ]* $H(2,4,5)$ **unfolding** *tautology-def* **by** *meson*

have $\text{sem-tree-size tree}' < \text{sem-tree-size tree}$ **using** *decrease* **unfolding** *tree* **by** *auto*

moreover have $\text{finite (fst } \psi')$ **using** *rtranclp-inference-preserves-finite inf* $H(4)$ **by** *metis*

moreover have $\neg \text{satisfiable (fst } \psi')$

using *inference-preserves-unsat inf bigger.premis(2)* **by** *blast*

moreover have $\text{already-used-inv } \psi'$

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** (*cases tree, auto*)

qed

qed

lemma *inference-completeness*:

fixes $\psi :: 'v :: \text{linorder state}$

assumes *unsat*: $\neg \text{satisfiable (fst } \psi)$

and *finite*: $\text{finite (fst } \psi)$

and $\text{snd } \psi = \{\}$

shows $\exists \psi'. (\text{rtranclp inference } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$

proof –

have $\text{already-used-inv } \psi$ **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* ψ ψ' **and** $\{\#\} \in \text{fst } \psi'$

shows *unsatisfiable* (*fst* ψ)

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* ψ)

and *snd* $\psi = \{\}$

shows $(\exists \psi'. (\text{inference}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow \text{unsatisfiable } (\text{fst } \psi)$

using *assms* *inference-completeness inference-soundness* **by** *metis*

12.4 Lemma about the simplified state

abbreviation *simplified* $\psi \equiv (\text{no-step simplify } \psi)$

lemma *simplified-count*:

assumes *simp*: *simplified* ψ **and** $\chi: \chi \in \psi$

shows *count* χ $L \leq 1$

proof –

{

let $? \chi' = \chi - \{\#L, L\# \}$

assume *count* χ $L \geq 2$

then have *f1*: *count* $(\chi - \{\#L, L\# \} + \{\#L, L\# \})$ $L = \text{count } \chi$ L

by *simp*

then have $L \in \# \chi - \{\#L\# \}$

by *simp*

then have χ' : $? \chi' + \{\#L\# \} + \{\#L\# \} = \chi$

using *f1* **by** (*metis* (*no-types*) *diff-diff-add diff-single-eq-union union-assoc union-single-eq-member*)

have $\exists \psi'. \text{simplify } \psi \psi'$

by (*metis* (*no-types*, *hide-lams*) χ χ' *add.commute factoring-imp-simplify union-assoc*)

then have *False* **using** *simp* **by** *auto*

}

then show *?thesis* **by** *arith*

qed

lemma *simplified-no-both*:

assumes *simp*: *simplified* ψ **and** $\chi: \chi \in \psi$

shows $\neg (L \in \# \chi \wedge \neg L \in \# \chi)$

proof (*rule ccontr*)

assume $\neg \neg (L \in \# \chi \wedge \neg L \in \# \chi)$

then have $L \in \# \chi \wedge \neg L \in \# \chi$ **by** *metis*

then obtain χ' **where** $\chi = \chi' + \{\# \text{Pos } (\text{atm-of } L)\# \} + \{\# \text{Neg } (\text{atm-of } L)\# \}$

by (*metis* *Neg-atm-of-iff Pos-atm-of-iff diff-union-swap insert-DiffM2 uminus-Neg uminus-Pos*)

then show *False* **using** χ *simp tautology-deletion* **by** *fastforce*

qed

lemma *simplified-not-tautology*:

assumes *simplified* $\{\psi\}$

shows $\sim \text{tautology } \psi$

proof (*rule ccontr*)

```

assume ~ ?thesis
then obtain p where Pos p ∈# ψ ∧ Neg p ∈# ψ using tautology-decomp by metis
then obtain χ where ψ = χ + {#Pos p#} + {#Neg p#}
  by (metis insert-noteq-member literal.distinct(1) multi-member-split)
then have ~ simplified {ψ} by (auto intro: tautology-deletion)
then show False using assms by auto
qed

```

```

lemma simplified-remove:
  assumes simplified {ψ}
  shows simplified {ψ - {#l#}}
proof (rule ccontr)
  assume ns: ¬ simplified {ψ - {#l#}}
  {
    assume ¬ l ∈# ψ
    then have ψ - {#l#} = ψ by simp
    then have False using ns assms by auto
  }
  moreover {
    assume lψ: l ∈# ψ
    have A: ∧ A. A ∈ {ψ - {#l#}} ↔ A + {#l#} ∈ {ψ} by (auto simp add: lψ)
    obtain l' where l': simplify {ψ - {#l#}} l' using ns by metis
    then have ∃ l'. simplify {ψ} l'
    proof (induction rule: simplify.induct)
      case (tautology-deletion A P)
      have {#Neg P#} + ({#Pos P#} + (A + {#l#})) ∈ {ψ}
        by (metis (no-types) A add.commute tautology-deletion.hyps union-lcomm)
      then show ?thesis
        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  $\psi$ ] incl by blast
next
    case (condensation A L)
    then show ?case using simplify.condensation[of A L  $\psi$ ] incl by blast
next
    case (subsumption A B)
    then show ?case using simplify.subsumption[of A  $\psi$  B] incl by auto
qed
then show False using assms(1) by blast
qed

```

```

lemma simplified-in:
  assumes simplified  $\psi$ 
  and  $N \in \psi$ 
  shows simplified  $\{N\}$ 
  using assms by (metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono)

```

```

lemma subsumes-imp-formula:
  assumes  $\psi \leq \# \varphi$ 
  shows  $\{\psi\} \models_p \varphi$ 
  unfolding true-clss-cls-def apply auto
  using assms true-cls-mono-leD by blast

```

```

lemma simplified-imp-distinct-mset-tauto:
  assumes simp: simplified  $\psi'$ 
  shows distinct-mset-set  $\psi'$  and  $\forall \chi \in \psi'. \neg \text{tautology } \chi$ 
proof -
  show  $\forall \chi \in \psi'. \neg \text{tautology } \chi$ 
  using simp by (auto simp add: simplified-in simplified-not-tautology)

```

```

show distinct-mset-set  $\psi'$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $\chi$  where  $\chi \in \psi'$  and  $\neg \text{distinct-mset } \chi$  unfolding distinct-mset-set-def by auto
  then obtain L where count  $\chi$  L  $\geq 2$ 
  unfolding distinct-mset-def by (metis gr-implies-not0 le-antisym less-one not-le simp
    simplified-count)
  then show False by (metis Suc-1  $\langle \chi \in \psi' \rangle$  not-less-eq-eq simp simplified-count)
qed
qed

```

```

lemma simplified-no-more-full1-simplified:
  assumes simplified  $\psi$ 
  shows  $\neg \text{full1 simplify } \psi \psi'$ 
  using assms unfolding full1-def by (meson tranclpD)

```

12.5 Resolution and Invariants

```

inductive resolution :: 'v state  $\Rightarrow$  'v state  $\Rightarrow$  bool where
  full1-simp: full1 simplify N N'  $\Longrightarrow$  resolution (N, already-used) (N', already-used) |
  inferring: inference (N, already-used) (N', already-used')  $\Longrightarrow$  simplified N
   $\Longrightarrow$  full simplify N' N''  $\Longrightarrow$  resolution (N, already-used) (N'', already-used')

```

12.5.1 Invariants

lemma *resolution-finite*:

assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
using *assms* **by** (*induct* rule: *resolution.induct*)
(auto simp add: full1-def full-def rtranclp-simplify-preserves-finite
dest: tranclp-into-rtranclp inference-preserves-finite)

lemma *rtranclp-resolution-finite*:

assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
using *assms* **by** (*induct* rule: *rtranclp-induct*, *auto simp add: resolution-finite*)

lemma *resolution-finite-snd*:

assumes *resolution* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
using *assms* **apply** (*induct* rule: *resolution.induct*, *auto simp add: inference-preserves-finite-snd*)
using *inference-preserves-finite-snd* *snd-conv* **by** *metis*

lemma *rtranclp-resolution-finite-snd*:

assumes *resolution*** ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
using *assms* **by** (*induct* rule: *rtranclp-induct*, *auto simp add: resolution-finite-snd*)

lemma *resolution-always-simplified*:

assumes *resolution* ψ ψ'
shows *simplified* (*fst* ψ')
using *assms* **by** (*induct* rule: *resolution.induct*)
(auto simp add: full1-def full-def)

lemma *tranclp-resolution-always-simplified*:

assumes *tranclp resolution* ψ ψ'
shows *simplified* (*fst* ψ')
using *assms* **by** (*induct* rule: *tranclp.induct*, *auto simp add: resolution-always-simplified*)

lemma *resolution-atms-of*:

assumes *resolution* ψ ψ' **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*** ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') \subseteq *atms-of-ms* (*fst* ψ)
using *assms* **apply** (*induct* rule: *rtranclp-induct*)
using *resolution-atms-of* *rtranclp-resolution-finite* **by** *blast+*

lemma *resolution-include*:

assumes *res: resolution* ψ ψ' **and** *finite: finite* (*fst* ψ)
shows *fst* $\psi' \subseteq$ *simple-cls* (*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 $\text{fst } \psi' \subseteq \text{simple-clss } (\text{atms-of-ms } (\text{fst } \psi'))$
using *simplified-in-simple-clss finite' simplified-imp-distinct-mset-tauto*[of $\text{fst } \psi'$] **by** *auto*
moreover have $\text{atms-of-ms } (\text{fst } \psi') \subseteq \text{atms-of-ms } (\text{fst } \psi)$
using *res finite resolution-atms-of*[of $\psi \psi'$] **by** *auto*
ultimately show *?thesis* **by** (*meson atms-of-ms-finite local.finite order.trans rev-finite-subset simple-clss-mono*)
qed

lemma *rtrancpl-resolution-include*:
assumes *res: trancpl resolution $\psi \psi'$ and finite: finite (fst ψ)*
shows $\text{fst } \psi' \subseteq \text{simple-clss } (\text{atms-of-ms } (\text{fst } \psi))$
using *assms apply* (*induct rule: trancpl.induct*)
apply (*simp add: resolution-include*)
by (*meson simple-clss-mono order-class.le-trans resolution-include*
rtrancpl-resolution-atms-of rtrancpl-resolution-finite trancpl-into-rtrancpl)

abbreviation *already-used-all-simple*
 $:: ('a \text{ literal multiset} \times 'a \text{ literal multiset}) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ **where**
already-used-all-simple already-used vars \equiv
 $(\forall (A, B) \in \text{already-used. simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars})$

lemma *already-used-all-simple-vars-incl*:
assumes $\text{vars} \subseteq \text{vars}'$
shows *already-used-all-simple a vars \implies already-used-all-simple a vars'*
using *assms by fast*

lemma *inference-clause-preserves-already-used-all-simple*:
assumes *inference-clause S S'*
and *already-used-all-simple (snd S) vars*
and *simplified (fst S)*
and $\text{atms-of-ms } (\text{fst } S) \subseteq \text{vars}$
shows *already-used-all-simple (snd (fst S \cup {fst S'}, snd S')) vars*
using *assms*
proof (*induct rule: inference-clause.induct*)
case (*factoring L C N already-used*)
then show *?case* **by** (*simp add: simplified-in factoring-imp-simplify*)
next
case (*resolution P C N D already-used*) **note** $H = \text{this}$
show *?case* **apply** *clarify*
proof –
fix $A B v$
assume $(A, B) \in \text{snd } (\text{fst } (N, \text{already-used}))$
 $\cup \{\text{fst } (C + D, \text{already-used} \cup \{(\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)\}),$
 $\text{snd } (C + D, \text{already-used} \cup \{(\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)\})\}$
then have $(A, B) \in \text{already-used} \vee (A, B) = (\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)$ **by** *auto*
moreover {
assume $(A, B) \in \text{already-used}$
then have $\text{simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars}$
using $H(4)$ **by** *auto*
}
moreover {
assume $\text{eq: } (A, B) = (\{\#Pos P\# \} + C, \{\#Neg P\# \} + D)$
then have $\text{simplified } \{A\}$ **using** *simplified-in H(1,5)* **by** *auto*
moreover have $\text{simplified } \{B\}$ **using** *eq simplified-in H(2,5)* **by** *auto*
moreover have $\text{atms-of } A \subseteq \text{atms-of-ms } N$

```

      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 rtranclp-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: rtranclp-induct)
  case base
  then show ?case by simp
next
  case (step S' S'')
  note infstar = this(1) and IH = this(3) and res = this(2) and
    already = this(4) and atms = this(5) and finite = this(6)
  have already-used-all-simple (snd S') vars using IH already atms finite by simp
  moreover have atms-of-ms (fst S')  $\subseteq$  atms-of-ms (fst S)

```


by (simp add: infstar local.finite rtranclp-resolution-atms-of)
 then have $\text{atms-of-ms } (\text{fst } S') \subseteq \text{vars}$ using *atms* by auto
 ultimately show ?case
 using *already-used-all-simple-inv*[OF *res*] by simp
 qed

lemma *inference-clause-simplified-already-used-subset*:
 assumes *inference-clause* $S S'$
 and *simplified* (fst S)
 shows $\text{snd } S \subset \text{snd } S'$
 using *assms* apply (induct rule: *inference-clause.induct*, auto)
 using *factoring-imp-simplify* by blast

lemma *inference-simplified-already-used-subset*:
 assumes *inference* $S S'$
 and *simplified* (fst S)
 shows $\text{snd } S \subset \text{snd } S'$
 using *assms* apply (induct rule: *inference.induct*)
 by (metis *inference-clause-simplified-already-used-subset* *snd-conv*)

lemma *resolution-simplified-already-used-subset*:
 assumes *resolution* $S S'$
 and *simplified* (fst S)
 shows $\text{snd } S \subset \text{snd } S'$
 using *assms* apply (induct rule: *resolution.induct*, *simp-all* add: *full1-def*)
 apply (meson *trancpD*)
 by (metis *inference-simplified-already-used-subset* *fst-conv* *snd-conv*)

lemma *trancp-resolution-simplified-already-used-subset*:
 assumes *trancp* *resolution* $S S'$
 and *simplified* (fst S)
 shows $\text{snd } S \subset \text{snd } S'$
 using *assms* apply (induct rule: *trancp.induct*)
 using *resolution-simplified-already-used-subset* apply metis
 by (meson *trancp-resolution-always-simplified* *resolution-simplified-already-used-subset* *less-trans*)

abbreviation *already-used-top vars* \equiv *simple-clss vars* \times *simple-clss vars*

lemma *already-used-all-simple-in-already-used-top*:
 assumes *already-used-all-simple* $s \text{ vars}$ and *finite vars*
 shows $s \subseteq \text{already-used-top vars}$
proof
 fix x
 assume $x-s: x \in s$
 obtain $A B$ where $x: x = (A, B)$ by (cases x , auto)
 then have *simplified* $\{A\}$ and $\text{atms-of } A \subseteq \text{vars}$ using *assms*(1) $x-s$ by *fastforce*+
 then have $A: A \in \text{simple-clss vars}$
 using *simple-clss-mono*[of $\text{atms-of } A \text{ vars}$] x *assms*(2)
simplified-imp-distinct-mset-tauto[of $\{A\}$]
distinct-mset-not-tautology-implies-in-simple-clss by *fast*
 moreover have *simplified* $\{B\}$ and $\text{atms-of } B \subseteq \text{vars}$ using *assms*(1) $x-s$ x by *fast*+
 then have $B: B \in \text{simple-clss vars}$
 using *simplified-imp-distinct-mset-tauto*[of $\{B\}$]
distinct-mset-not-tautology-implies-in-simple-clss

$\text{simple-clss-mono}[\text{of } \text{atms-of } B \text{ vars}] \ x \ \text{assms}(2) \text{ by fast}$
ultimately show $x \in \text{simple-clss vars} \times \text{simple-clss vars}$
 unfolding x by *auto*
qed

lemma *already-used-top-finite*:
 assumes *finite vars*
 shows *finite (already-used-top vars)*
 using *simple-clss-finite assms* by *auto*

lemma *already-used-top-increasing*:
 assumes $\text{var} \subseteq \text{var}'$ and *finite var'*
 shows *already-used-top var \subseteq already-used-top var'*
 using *assms simple-clss-mono* by *auto*

lemma *already-used-all-simple-finite*:
 fixes $s :: ('a \text{ literal multiset} \times 'a \text{ literal multiset}) \text{ set}$ and $\text{vars} :: 'a \text{ set}$
 assumes *already-used-all-simple s vars* and *finite vars*
 shows *finite s*
 using *assms already-used-all-simple-in-already-used-top[OF assms(1)]*
rev-finite-subset[OF already-used-top-finite[of vars]] by *auto*

abbreviation *card-simple vars $\psi \equiv \text{card (already-used-top vars} - \psi)$*

lemma *resolution-card-simple-decreasing*:
 assumes *res: resolution $\psi \ \psi'$*
 and *a-u-s: already-used-all-simple (snd ψ) vars*
 and *finite-v: finite vars*
 and *finite-fst: finite (fst ψ)*
 and *finite-snd: finite (snd ψ)*
 and *simp: simplified (fst ψ)*
 and *atms-of-ms (fst ψ) \subseteq vars*
 shows *card-simple vars (snd ψ') < card-simple vars (snd ψ)*

proof –
 let $?vars = \text{vars}$
 let $?top = \text{simple-clss } ?vars \times \text{simple-clss } ?vars$
 have 1: *card-simple vars (snd ψ) = card ?top – card (snd ψ)*
 using *card-Diff-subset finite-snd already-used-all-simple-in-already-used-top[OF a-u-s]*
finite-v by *metis*
 have *a-u-s': already-used-all-simple (snd ψ') vars*
 using *already-used-all-simple-inv res a-u-s assms(7)* by *blast*
 have *f: finite (snd ψ')* using *already-used-all-simple-finite a-u-s' finite-v* by *auto*
 have 2: *card-simple vars (snd ψ') = card ?top – card (snd ψ')*
 using *card-Diff-subset[OF f] already-used-all-simple-in-already-used-top[OF a-u-s' finite-v]*
 by *auto*
 have *card (already-used-top vars) \geq card (snd ψ')*
 using *already-used-all-simple-in-already-used-top[OF a-u-s' finite-v]*
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 *trancp-resolution-card-simple-decreasing*:

assumes *trancpl resolution* $\psi \psi'$ **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*: *trancpl-induct*)
case (*base* ψ')
then show ?*case* **by** (*simp add*: *resolution-card-simple-decreasing*)
next
case (*step* $\psi' \psi''$) **note** *res* = *this*(1) **and** *res'* = *this*(2) **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 *rtrancpl-already-used-all-simple-inv*[*OF* *trancpl-into-rtrancpl*[*OF* *res*] *a-u-s atms f-fst*] .
have *finite* (*fst* ψ')
by (*meson finite-fst res rtrancpl-resolution-finite trancpl-into-rtrancpl*)
moreover have *finite* (*snd* ψ') **using** *already-used-all-simple-finite*[*OF* *a-u-s' f-v*] .
moreover have *simplified* (*fst* ψ') **using** *res trancpl-resolution-always-simplified* **by** *blast*
moreover have *atms-of-ms* (*fst* ψ') \subseteq *vars*
by (*meson atms f-fst order.trans res rtrancpl-resolution-atms-of trancpl-into-rtrancpl*)
ultimately show ?*case*
using *resolution-card-simple-decreasing*[*OF* *res' a-u-s' f-v*] *f-v*
less-trans[*of card-simple vars* (*snd* ψ'') *card-simple vars* (*snd* ψ')
card-simple vars (*snd* ψ)]
by *blast*
qed

lemma *trancpl-resolution-card-simple-decreasing-2*:
assumes *trancpl resolution* $\psi \psi'$
and *finite-fst*: *finite* (*fst* ψ)
and *empty-snd*: *snd* ψ = {}
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) *trancpl-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*:: *linorder state*, *x*). (*atms-of-ms* (*fst* *x*) \subseteq *vars* \wedge *simplified* (*fst* *x*)
 \wedge *finite* (*snd* *x*) \wedge *finite* (*fst* *x*) \wedge *already-used-all-simple* (*snd* *x*) *vars*) \wedge *resolution* *x y*}
proof –
{
fix *a b* :: '*v*::*linorder state*

```

assume  $(b, a) \in \{(y, x). (atms-of-ms (fst x) \subseteq vars \wedge simplified (fst x) \wedge finite (snd x) \wedge finite (fst x) \wedge already-used-all-simple (snd x) vars) \wedge resolution x y)\}$ 
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  $wf-bounded-set[of\ \{(y:: 'v:: linorder\ state, x). (atms-of-ms (fst x) \subseteq vars \wedge simplified (fst x) \wedge finite (snd x) \wedge finite (fst x) \wedge already-used-all-simple (snd x) vars) \wedge resolution x y)\} \lambda-. already-used-top vars\ snd]$  by auto
qed

```

lemma *wf-simplified-resolution'*:

```

assumes  $f-vars: finite\ vars$ 
shows  $wf\ \{(y:: 'v:: linorder\ state, x). (atms-of-ms (fst x) \subseteq vars \wedge \neg simplified (fst x) \wedge finite (snd x) \wedge finite (fst x) \wedge already-used-all-simple (snd x) vars) \wedge resolution x y)\}$ 
unfolding  $wf-def$ 
apply  $(simp\ add: resolution-always-simplified)$ 
by  $(metis\ (mono-tags, hide-lams)\ fst-conv\ resolution-always-simplified)$ 

```

lemma *wf-resolution*:

```

assumes  $f-vars: finite\ vars$ 
shows  $wf\ (\{(y:: 'v:: linorder\ state, x). (atms-of-ms (fst x) \subseteq vars \wedge simplified (fst x) \wedge finite (snd x) \wedge finite (fst x) \wedge already-used-all-simple (snd x) vars) \wedge resolution x y\} \cup \{(y, x). (atms-of-ms (fst x) \subseteq vars \wedge \neg simplified (fst x) \wedge finite (snd x) \wedge finite (fst x) \wedge already-used-all-simple (snd x) vars) \wedge resolution x y)\})$  (is  $wf\ (?R \cup ?S)$ )

```

proof –

```

have  $Domain\ ?R\ Int\ Range\ ?S = \{\}$  using  $resolution-always-simplified$  by auto blast
then show  $wf\ (?R \cup ?S)$ 
  using  $wf-simplified-resolution[OF\ f-vars]\ wf-simplified-resolution'[OF\ f-vars]\ wf-Un[of\ ?R\ ?S]$ 
  by fast

```

qed

lemma *rtrncp-simplify-already-used-inv*:

```

assumes  $simplify^{**}\ S\ S'$ 
and  $already-used-inv\ (S, N)$ 
shows  $already-used-inv\ (S', N)$ 
using assms apply induction
using  $simplify-preserves-already-used-inv$  by fast+

```

lemma *full1-simplify-already-used-inv*:

```

assumes  $full1\ simplify\ S\ S'$ 

```

and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
using *assms* *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** $\psi \psi'$*
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 $\psi \psi'$*
shows *satisfiable $\psi' \longrightarrow$ satisfiable ψ*
using *assms* *rtanclp-simplify-preserves-unsat*[*of $\psi \psi'$*] *trancp-into-rtrancp*
unfolding *full1-def* **by** *metis*

lemma *full-simplify-preserves-unsat*:
assumes *full simplify $\psi \psi'$*
shows *satisfiable $\psi' \longrightarrow$ satisfiable ψ*
using *assms* *rtanclp-simplify-preserves-unsat*[*of $\psi \psi'$*] **unfolding** *full-def* **by** *metis*

lemma *resolution-preserves-unsat*:
assumes *resolution $\psi \psi'$*
shows *satisfiable (fst 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 rtrancpl-resolution-preserves-unsat:
  assumes resolution**  $\psi \ \psi'$ 
  shows satisfiable (fst  $\psi'$ )  $\longrightarrow$  satisfiable (fst  $\psi$ )
  using assms apply induction
  using resolution-preserves-unsat by fast+

lemma rtrancpl-simplify-preserve-partial-tree:
  assumes simplify**  $N \ N'$ 
  and partial-interps  $t \ I \ N$ 
  shows partial-interps  $t \ I \ N'$ 
  using assms apply (induction, simp)
  using simplify-preserve-partial-tree by metis

lemma full1-simplify-preserve-partial-tree:
  assumes full1 simplify  $N \ N'$ 
  and partial-interps  $t \ I \ N$ 
  shows partial-interps  $t \ I \ N'$ 
  using assms rtrancpl-simplify-preserve-partial-tree[of  $N \ N' \ t \ I$ ] trancpl-into-rtrancpl
  unfolding full1-def by fast

lemma full-simplify-preserve-partial-tree:
  assumes full simplify  $N \ N'$ 
  and partial-interps  $t \ I \ N$ 
  shows partial-interps  $t \ I \ N'$ 
  using assms rtrancpl-simplify-preserve-partial-tree[of  $N \ N' \ t \ I$ ] trancpl-into-rtrancpl
  unfolding full-def by fast

lemma resolution-preserve-partial-tree:
  assumes resolution  $S \ S'$ 
  and partial-interps  $t \ I \ (\text{fst } S)$ 
  shows partial-interps  $t \ I \ (\text{fst } S')$ 
  using assms apply induction
  using full1-simplify-preserve-partial-tree fst-conv apply metis
  using full-simplify-preserve-partial-tree inference-preserve-partial-tree by fastforce

lemma rtrancpl-resolution-preserve-partial-tree:
  assumes resolution**  $S \ S'$ 
  and partial-interps  $t \ I \ (\text{fst } S)$ 
  shows partial-interps  $t \ I \ (\text{fst } S')$ 
  using assms apply induction
  using resolution-preserve-partial-tree by fast+
  thm nat-less-induct nat.induct

lemma nat-ge-induct[case-names 0 Suc]:
  assumes  $P \ 0$ 
  and  $(\bigwedge n. (\bigwedge m. m < \text{Suc } n \implies P \ m) \implies P \ (\text{Suc } n))$ 
  shows  $P \ n$ 
  using assms apply (induct rule: nat-less-induct)
  by (rename-tac  $n$ , case-tac  $n$ ) auto

lemma wf-always-more-step-False:
  assumes wf  $R$ 
  shows  $(\forall x. \exists z. (z, x) \in R) \implies \text{False}$ 

```

using *assms* **unfolding** *wf-def* **by** (*meson Domain.DomainI assms wfE-min*)

lemma *finite-finite-mset-element-of-mset[simp]*:

assumes *finite N*

shows *finite {f φ L | φ 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 φ L | φ L. ($\varphi = x \vee \varphi \in N$) $\wedge L \in \# \varphi \wedge P \varphi L$ } \subseteq {f *x* L | L. L $\in \# x \wedge P x L$ }
 \cup {f φ L | φ L. $\varphi \in N \wedge L \in \# \varphi \wedge P \varphi L$ } **by** *auto**

moreover **have** *finite {f *x* L | L. L $\in \# x$ }* **by** *auto*

ultimately **show** *?case* **using** *IH finite-subset* **by** *fastforce*

qed

value *card*

value *filter-mset*

value *{#count φ L | L $\in \# \varphi$. $2 \leq \text{count } \varphi L \#$ }*

value ($\lambda \varphi$. *msetsum {#count φ L | L $\in \# \varphi$. $2 \leq \text{count } \varphi L \#$ }*)

syntax

-comprehension1'-mset :: 'a \Rightarrow 'b \Rightarrow 'b multiset \Rightarrow 'a multiset

(({#-/. - : setof -#}))

translations

{#e. x: setof M#} == CONST set-mset (CONST image-mset (%x. e) M)

value *{# a. a : setof {#1,1,2::int#}#} = {1,2}*

definition *sum-count-ge-2 :: 'a multiset set \Rightarrow nat (Ξ) where*

sum-count-ge-2 \equiv folding.F ($\lambda \varphi$. op + (msetsum {#count φ L | L $\in \# \varphi$. $2 \leq \text{count } \varphi L \#$ })) 0

interpretation *sum-count-ge-2:*

folding ($\lambda \varphi$. op + (msetsum {#count φ L | L $\in \# \varphi$. $2 \leq \text{count } \varphi L \#$ })) 0

rewrites

folding.F ($\lambda \varphi$. op + (msetsum {#count φ L | L $\in \# \varphi$. $2 \leq \text{count } \varphi L \#$ })) 0 = sum-count-ge-2

proof –

show *folding ($\lambda \varphi$. op + (msetsum (image-mset (count φ) {# L :# φ . $2 \leq \text{count } \varphi L \#$ }))*

by standard auto

then interpret *sum-count-ge-2:*

folding ($\lambda \varphi$. op + (msetsum {#count φ L | L $\in \# \varphi$. $2 \leq \text{count } \varphi L \#$ })) 0 .

show *folding.F ($\lambda \varphi$. op + (msetsum (image-mset (count φ) {# L :# φ . $2 \leq \text{count } \varphi L \#$ })) 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 ∈ A)
  assume a ∉ A
  then have A ⊆ F using AF by auto
  then show ?case using IH[of A] by (simp add: aF local.finite)
next
  assume aA: a ∈ A
  then have A − {a} ⊆ F using AF by auto
  then have ∃ (A − {a}) ≤ ∃ F using IH by blast
  then show ?case
    proof −
      obtain nn :: nat ⇒ nat ⇒ nat where
        ∀ x0 x1. (∃ v2. x0 = x1 + v2) = (x0 = x1 + nn x0 x1)
      by moura
      then have ∃ F = ∃ (A − {a}) + nn (∃ F) (∃ (A − {a}))
      by (meson (∃ (A − {a}) ≤ ∃ F) le-iff-add)
      then show ?thesis
        by (metis (no-types) 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 simplify-finite-measure-decrease:
  simplify N N' ⇒ finite N ⇒ card N' + ∃ N' < card N + ∃ N
proof (induction rule: simplify.induct)
  case (tautology-deletion A P) note an = this(1) and fin = this(2)
  let ?N' = N − {A + {#Pos P#} + {#Neg P#}}
  have card ?N' < card N
    by (meson card-Diff1-less tautology-deletion.hyps tautology-deletion.prem)
  moreover have ?N' ⊆ N by auto
  then have sum-count-ge-2 ?N' ≤ sum-count-ge-2 N using finite-incl-le-setsum[OF fin] by blast
  ultimately show ?case by linarith
next
  case (condensation A L) note AN = this(1) and fin = this(2)
  let ?C' = A + {#L#}
  let ?C = A + {#L#} + {#L#}
  let ?N' = N − {?C} ∪ {?C'}
  have card ?N' ≤ 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 ∃ {?C'} < ∃ {?C}
  proof −
    have mset-decomp:
      {# La ∈# A. (L = La ⟶ Suc 0 ≤ count A La) ∧ (L ≠ La ⟶ 2 ≤ count A La)#}
      = {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} +
        {# La ∈# A. L = La ∧ Suc 0 ≤ count A L#}
    by (auto simp: multiset-eq-iff ac-simps)
    have mset-decomp2: {# La ∈# A. L ≠ La ⟶ 2 ≤ count A La#} =
      {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} + replicate-mset (count A L) L
    by (auto simp: multiset-eq-iff)
    show ?thesis
      by (auto simp: mset-decomp mset-decomp2 filter-mset-eq ac-simps)
  qed
have ∃ ?N' < ∃ N
  proof cases

```



```

assume a1: ?C' ∈ 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  $\Xi (N - \{A + \{\#L\# \} + \{\#L\# \}) < \Xi N$ 
      using f5 f4 by (metis Un-empty-right  $\langle \Xi \{A + \{\#L\# \}\} < \Xi \{A + \{\#L\# \} + \{\#L\# \} \rangle$ 
        add.right-neutral add-diff-cancel-left' add-gr-0 diff-less fin finite.emptyI not-le
        sum-count-ge-2.empty sum-count-ge-2.insert-remove trans-le-add2)
    then show ?thesis
      using f3 f2 a1 by (metis (no-types) Un-empty-right Un-insert-right condensation.hyps
        insert-iff multi-self-add-other-not-self)
    qed
  next
    assume ?C' ∉ N
    have mset-decomp:
       $\{\# La \in \# A. (L = La \longrightarrow \text{Suc } 0 \leq \text{count } A La) \wedge (L \neq La \longrightarrow 2 \leq \text{count } A La)\# \}$ 
      =  $\{\# La \in \# A. L \neq La \wedge 2 \leq \text{count } A La\# \} +$ 
       $\{\# La \in \# A. L = La \wedge \text{Suc } 0 \leq \text{count } A L\# \}$ 
      by (auto simp: multiset-eq-iff ac-simps)
    have mset-decomp2:  $\{\# La \in \# A. L \neq La \longrightarrow 2 \leq \text{count } A La\# \} =$ 
       $\{\# La \in \# A. L \neq La \wedge 2 \leq \text{count } A La\# \} + \text{replicate-mset } (\text{count } A L) L$ 
      by (auto simp: multiset-eq-iff)

    show ?thesis
      using  $\langle \Xi \{A + \{\#L\# \}\} < \Xi \{A + \{\#L\# \} + \{\#L\# \}\rangle$  condensation.hyps fin
        sum-count-ge-2.remove[of - A +  $\{\#L\# \} + \{\#L\# \}$ ] (?C' ∉ N)
      by (auto simp: mset-decomp mset-decomp2 filter-mset-eq)
    qed
  ultimately show ?case by linarith
next
  case (subsumption A B) note AN = this(1) and AB = this(2) and BN = this(3) and fin = this(4)
  have card (N - {B}) < card N using BN by (meson card-Diff1-less subsumption.prem)
  moreover have  $\Xi (N - \{B\}) \leq \Xi N$ 
    by (simp add: Diff-subset finite-incl-le-setsum subsumption.prem)
  ultimately show ?case by linarith
qed

lemma simplify-terminates:
  wf {(N', N). finite N ∧ simplify N N'}
  using assms apply (rule wfP-if-measure[of finite simplify λN. 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 $\text{wf: wf } \{(N', N). \text{simplify } N N' \wedge \text{finite } N\}$
using *simplify-terminates* **by** *(simp add: H)*
obtain N' **where** $N': (N', N) \in \{(b, a). \text{simplify } a b \wedge \text{finite } a\}^*$ **and**
more: $(\forall N''. (N'', N') \notin \{(b, a). \text{simplify } a b \wedge \text{finite } a\})$
using *Prop-Resolution.wf-terminates[OF wf, of N]* **by** *blast*
have $1: \text{simplify}^{**} N N'$
using N' **by** *(induction rule: rtrancl.induct) auto*
then have *finite N'* **using** *fin rtranclp-simplify-preserves-finite* **by** *blast*
then have $2: \forall N''. \neg \text{simplify } N' N''$ **using** *more* **by** *auto*

show *?thesis* **using** $1\ 2$ **by** *blast*
qed

lemma *finite-simplified-full1-simp*:

assumes *finite N*
shows $\text{simplified } N \vee (\exists N'. \text{full1 simplify } N N')$
using *rtranclp-simplify-terminates[OF assms]* **unfolding** *full1-def*
by *(metis Nitpick.rtranclp-unfold)*

lemma *finite-simplified-full-simp*:

assumes *finite N*
shows $\exists N'. \text{full simplify } N N'$
using *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 $\exists (tree':: 'v \text{ sem-tree}) \psi'. \text{resolution}^{**} \psi \psi' \wedge \text{partial-interps } tree' I (\text{fst } \psi')$
 $\wedge (\text{sem-tree-size } tree' < \text{sem-tree-size } tree \vee \text{sem-tree-size } tree = 0)$
using *assms*
proof (*induct arbitrary: I rule: sem-tree-size*)
case (*bigger* *xs* *I*) **note** *IH* = *this*(1) **and** *finite* = *this*(2) **and** *a-u-i* = *this*(3) **and** *part* = *this*(4)
and *simp* = *this*(5)

{ **assume** *sem-tree-size* *xs* = 0
then have ?*case* **using** *part* **by** *blast*
 }

moreover {
assume *sn0*: *sem-tree-size* *xs* > 0
obtain *ag* *ad* *v* **where** *xs*: *xs* = *Node* *v* *ag* *ad* **using** *sn0* **by** (*cases* *xs*, *auto*)
 {
assume *sem-tree-size* *ag* = 0 \wedge *sem-tree-size* *ad* = 0
then have *ag*: *ag* = *Leaf* **and** *ad*: *ad* = *Leaf* **by** (*cases* *ag*, *auto*, *cases* *ad*, *auto*)

then obtain χ χ' **where**
 χ : $\neg I \cup \{Pos\ v\} \models \chi$ **and**
 $tot\chi$: *total-over-m* ($I \cup \{Pos\ v\}$) $\{\chi\}$ **and**
 $\chi\psi$: $\chi \in \text{fst } \psi$ **and**
 χ' : $\neg I \cup \{Neg\ v\} \models \chi'$ **and**
 $tot\chi'$: *total-over-m* ($I \cup \{Neg\ v\}$) $\{\chi'\}$ **and** $\chi'\psi$: $\chi' \in \text{fst } \psi$
using *part unfolding* *xs* **by** *auto*
have *Posv*: *Pos* *v* $\notin \# \chi$ **using** χ **unfolding** *true-cls-def* *true-lit-def* **by** *auto*
have *Negv*: *Neg* *v* $\notin \# \chi'$ **using** χ' **unfolding** *true-cls-def* *true-lit-def* **by** *auto*
 {
assume *Neg* χ : $\neg Neg\ v \in \# \chi$
then have $\neg I \models \chi$ **using** χ *Posv* **unfolding** *true-cls-def* *true-lit-def* **by** *auto*
moreover have *total-over-m* *I* $\{\chi\}$
using *Posv* *Neg* χ *atm-imp-pos-or-neg-lit* *tot* χ **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 *resolution*^{**} ψ ψ
unfolding *xs* **by** (*auto* *simp* *add*: $\chi\psi$)
 }
moreover {
assume *Pos* χ : $\neg Pos\ v \in \# \chi'$
then have *I* χ : $\neg I \models \chi'$ **using** χ' *Posv* **unfolding** *true-cls-def* *true-lit-def* **by** *auto*
moreover have *total-over-m* *I* $\{\chi'\}$
using *Negv* *Pos* χ *atm-imp-pos-or-neg-lit* *tot* χ'
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 *resolution*^{**} ψ ψ **using** $\chi'\psi$ *I* χ **unfolding** *xs* **by** *auto*
 }
moreover {
assume *neg*: *Neg* *v* $\in \# \chi$ **and** *pos*: *Pos* *v* $\in \# \chi'$
have *count* χ (*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) count  $\chi\ (Neg\ v) = 1$ ) add.commute multi-member-split
      zero-less-one)
    then have Pos v  $\notin\#$  mm  $\chi\ (Neg\ v)$ 
    using f2 by (metis (no-types) Posv count  $\chi\ (Neg\ v) = 1$ ) add.right-neutral
      add-left-cancel count-single count-union less-nat-zero-code)
    then show ?thesis
    using f3 a1 by (metis (no-types) count  $\chi\ (Neg\ v) = 1$ ) add.commute
      add.right-neutral add-left-cancel count-single count-union less-nat-zero-code)
  qed
  obtain C' where
     $\chi C'$ :  $\chi' = C' + \{\#Pos\ v\#\}$  and
    posC': Pos v  $\notin\#$  C' and
    negC': Neg v  $\notin\#$  C'
  by (metis (no-types, hide-lams) Negv count  $\chi'\ (Pos\ v) = 1$ ) add.diff-cancel-right'
    cancel-comm-monoid-add-class.diff-cancel count-diff count-single less-nat-zero-code
    mset-leD mset-le-add-left multi-member-split zero-less-one)

  have totC: total-over-m I {C}
  using tot $\chi$  tot-over-m-remove[of I Pos v C] negC posC unfolding  $\chi C$ 
  by (metis total-over-m-sum uminus-Neg uminus-of-uminus-id)
  have totC': total-over-m I {C'}
  using tot $\chi'$  total-over-m-sum tot-over-m-remove[of I Neg v C'] negC' posC'
  unfolding  $\chi C'$  by (metis total-over-m-sum uminus-Neg)
  have  $\neg I \models C + C'$ 
  using  $\chi\ \chi'\ \chi C\ \chi C'$  by auto
  then have part-I- $\psi'''$ : partial-interps Leaf I (fst  $\psi \cup \{C + C'\}$ )
  using totC totC'  $\neg I \models C + C'$  by (metis Un-insert-right insertI1
    partial-interps.simps(1) total-over-m-sum)
  {
    assume ( $\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C$ )  $\notin$  snd  $\psi$ 
    then have inf'': inference  $\psi$  (fst  $\psi \cup \{C + C'\}$ , snd  $\psi \cup \{(\chi', \chi)\}$ )
    by (metis  $\chi'\psi\ \chi C\ \chi C'\ \chi\psi$  add.commute inference-step prod.collapse resolution)
    obtain N' where full: full simplify (fst  $\psi \cup \{C + C'\}$ ) N'
    by (metis finite-simplified-full-simp fst-conv inf'' inference-preserves-finite
      local.finite)
    have resolution  $\psi$  (N', snd  $\psi \cup \{(\chi', \chi)\}$ )
    using resolution.intros(2)[OF - simp full, of snd  $\psi$  snd  $\psi \cup \{(\chi', \chi)\}$ ] inf''
    by (metis surjective-pairing)
    moreover have partial-interps Leaf I N'
    using full-simplify-preserve-partial-tree[OF full part-I- $\psi'''$ ] .
    moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
    ultimately have ?case
    by (metis (no-types) prod.sel(1) rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
  }
}

```

```

moreover {
  assume  $a: (\{\#Pos\ v\# \} + C', \{\#Neg\ v\# \} + C) \in \text{snd } \psi$ 
  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 (fst ψ)
    using  $\text{tot-}\vartheta\text{-}CC' \vartheta \vartheta\text{-inv totC totC'} \hookrightarrow I \models C + C'$  total-over-m-sum by fastforce
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding  $xs$  by auto
  ultimately have ?case by blast
}
moreover {
  assume  $\text{tautCC}': \text{tautology } (C' + C)$ 
  have  $\text{total-over-m } I \{C'+C\}$  using  $\text{totC totC'}$  total-over-m-sum by auto
  then have  $\neg \text{tautology } (C' + C)$ 
    using  $\hookrightarrow I \models C + C'$  unfolding  $\text{add.commute[of } C\ C']$  total-over-m-def
    unfolding tautology-def by auto
  then have False using  $\text{tautCC}'$  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  $\text{size-ag}: \text{sem-tree-size ag} > 0$ 
  have  $\text{sem-tree-size ag} < \text{sem-tree-size xs}$  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  $\implies$  finite (fst  $\psi$ )  $\implies$  already-used-inv  $\psi$ 
 $\implies$  partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi$ )  $\implies$  simplified (fst  $\psi$ )
 $\implies$   $\exists$  tree'  $\psi'$ . resolution**  $\psi \psi' \wedge$  partial-interps tree' (I  $\cup$  {Pos v}) (fst  $\psi'$ )
 $\wedge$  (sem-tree-size tree' < sem-tree-size ag  $\vee$  sem-tree-size ag = 0)
using IH[of ag I  $\cup$  {Pos v}] by auto
ultimately obtain  $\psi' :: 'v$  state and tree' :: 'v sem-tree where
inf: resolution**  $\psi \psi'$ 
and part: partial-interps tree' (I  $\cup$  {Pos v}) (fst  $\psi'$ )
and size: sem-tree-size tree' < sem-tree-size ag  $\vee$  sem-tree-size ag = 0
using finite part rtranclp.rtrancl-refl a-u-i simp by blast

have partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi'$ )
using rtranclp-resolution-preserve-partial-tree inf partad by fast
then have partial-interps (Node v tree' ad) I (fst  $\psi'$ ) using part by auto
then have ?case using inf size size-ag part unfolding xs by fastforce
}
moreover {
assume size-ad: sem-tree-size ad > 0
have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
moreover
have
partag: partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi$ ) and
partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )
using part partial-interps.simps(2) unfolding xs by metis+
moreover have sem-tree-size ad < sem-tree-size xs  $\longrightarrow$  finite (fst  $\psi$ )  $\longrightarrow$  already-used-inv  $\psi$ 
 $\longrightarrow$  (partial-interps ad (I  $\cup$  {Neg v}) (fst  $\psi$ )  $\longrightarrow$  simplified (fst  $\psi$ )
 $\longrightarrow$  ( $\exists$  tree'  $\psi'$ . resolution**  $\psi \psi' \wedge$  partial-interps tree' (I  $\cup$  {Neg v}) (fst  $\psi'$ )
 $\wedge$  (sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0)))
using IH by blast
ultimately obtain  $\psi' :: 'v$  state and tree' :: 'v sem-tree where
inf: resolution**  $\psi \psi'$ 
and part: partial-interps tree' (I  $\cup$  {Neg v}) (fst  $\psi'$ )
and size: sem-tree-size tree' < sem-tree-size ad  $\vee$  sem-tree-size ad = 0
using finite part rtranclp.rtrancl-refl a-u-i simp by blast

have partial-interps ag (I  $\cup$  {Pos v}) (fst  $\psi'$ )
using rtranclp-resolution-preserve-partial-tree inf partag by fast
then have partial-interps (Node v ag tree') I (fst  $\psi'$ ) using part by auto
then have ?case using inf size size-ad unfolding xs by fastforce
}
ultimately have ?case by auto
}
ultimately show ?case by auto
qed

```

lemma *resolution-completeness-inv:*

fixes *$\psi :: 'v :: \text{linorder}$ state*

assumes

unsat: \neg satisfiable (fst ψ) **and**

finite: finite (fst ψ) **and**

a-u-v: already-used-inv ψ

shows $\exists \psi'. (\text{resolution** } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$

proof –

obtain *tree* **where** *partial-interps tree {} (fst ψ)*

using *partial-interps-build-sem-tree-atms asms* **by** *metis*

```

then show ?thesis
using unsat finite a-u-v
proof (induct tree arbitrary:  $\psi$  rule: sem-tree-size)
case (bigger tree  $\psi$ ) note  $H = \text{this}$ 
{
  fix  $\chi$ 
  assume tree: tree = Leaf
  obtain  $\chi$  where  $\chi: \neg \{\} \models \chi$  and tot $\chi$ : total-over-m  $\{\} \{\chi\}$  and  $\chi\psi: \chi \in \text{fst } \psi$ 
  using  $H$  unfolding tree by auto
  moreover have  $\{\#\} = \chi$ 
  using  $H$  atms-empty-iff-empty tot $\chi$ 
  unfolding true-cls-def total-over-m-def total-over-set-def by fastforce
  moreover have resolution**  $\psi \psi$  by auto
  ultimately have ?case by metis
}
moreover {
  fix  $v$  tree1 tree2
  assume tree: tree = Node  $v$  tree1 tree2
  obtain  $\psi_0$  where  $\psi_0$ : resolution**  $\psi \psi_0$  and simp: simplified (fst  $\psi_0$ )
  proof -
    { assume simplified (fst  $\psi$ )
      moreover have resolution**  $\psi \psi$  by auto
      ultimately have thesis using that by blast
    }
    moreover {
      assume  $\neg$ simplified (fst  $\psi$ )
      then have  $\exists \psi'. \text{full1 simplify (fst } \psi) \psi'$ 
      by (metis Nitpick.rtranclp-unfold bigger.prem(3) full1-def
        rtranclp-simplify-terminates)
      then obtain  $N$  where full1 simplify (fst  $\psi$ )  $N$  by metis
      then have resolution  $\psi (N, \text{snd } \psi)$ 
      using resolution.intros(1)[of fst  $\psi$   $N$  snd  $\psi$ ] by auto
      moreover have simplified  $N$ 
      using  $\langle \text{full1 simplify (fst } \psi) N \rangle$  unfolding full1-def by blast
      ultimately have ?thesis using that by force
    }
    ultimately show ?thesis by auto
  qed

  have  $p$ : partial-interps tree  $\{\}$  (fst  $\psi_0$ )
  and uns: unsatisfiable (fst  $\psi_0$ )
  and  $f$ : finite (fst  $\psi_0$ )
  and a-u-v: already-used-inv  $\psi_0$ 
  using  $\psi_0$  bigger.prem(1) rtranclp-resolution-preserve-partial-tree apply blast
  using  $\psi_0$  bigger.prem(2) rtranclp-resolution-preserves-unsat apply blast
  using  $\psi_0$  bigger.prem(3) rtranclp-resolution-finite apply blast
  using rtranclp-resolution-already-used-inv[OF  $\psi_0$  bigger.prem(4)] by blast
  obtain tree'  $\psi'$  where
  inf: resolution**  $\psi_0 \psi'$  and
  part': partial-interps tree'  $\{\}$  (fst  $\psi'$ ) and
  decrease: sem-tree-size tree' < sem-tree-size tree  $\vee$  sem-tree-size tree = 0
  using can-decrease-tree-size-resolution[OF  $f$  a-u-v  $p$  simp] unfolding tautology-def
  by meson
  have  $s$ : sem-tree-size tree' < sem-tree-size tree using decrease unfolding tree by auto

```

```

    have fin: finite (fst  $\psi'$ )
      using f inf rtrancpl-resolution-finite by blast
    have unsat: unsatisfiable (fst  $\psi'$ )
      using rtrancpl-resolution-preserves-unsat inf uns by metis
    have a-u-i': already-used-inv  $\psi'$ 
      using a-u-v inf rtrancpl-resolution-already-used-inv[of  $\psi_0$   $\psi'$ ] by auto
    have ?case
      using inf rtrancpl-trans[of resolution]  $H(1)[OF s part' unsat fin a-u-i'] \psi_0$  by blast
  }
  ultimately show ?case by (cases tree, auto)
qed
qed

```

```

lemma resolution-preserves-already-used-inv:
  assumes resolution S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
  apply (induct rule: resolution.induct)
  apply (rule full1-simplify-already-used-inv; simp)
  apply (rule full-simplify-already-used-inv, simp)
  apply (rule inference-preserves-already-used-inv, simp)
  apply blast
done

```

```

lemma rtrancpl-resolution-preserves-already-used-inv:
  assumes resolution** S S'
  and already-used-inv S
  shows already-used-inv S'
  using assms
  apply (induct rule: rtrancpl-induct)
  apply simp
  using resolution-preserves-already-used-inv by fast

```

```

lemma resolution-completeness:
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes unsat:  $\neg \text{satisfiable (fst } \psi)$ 
  and finite: finite (fst  $\psi$ )
  and snd  $\psi = \{\}$ 
  shows  $\exists \psi'. (\text{resolution** } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$ 
proof -
  have already-used-inv  $\psi$  unfolding assms by auto
  then show ?thesis using assms resolution-completeness-inv by blast
qed

```

```

lemma rtrancpl-preserves-sat:
  assumes simplify** S S'
  and satisfiable S
  shows satisfiable S'
  using assms apply induction
  apply simp
  by (meson satisfiable-carac satisfiable-def simplify-preserves-un-sat-eq)

```

```

lemma resolution-preserves-sat:
  assumes resolution S S'

```


and *satisfiable* (*fst S*)
shows *satisfiable* (*fst S'*)
using *assms* **apply** (*induction rule: resolution.induct*)
 using *rtrancplp-preserves-sat* *trancplp-into-rtrancplp* **unfolding** *full1-def* **apply** *fastforce*
by (*metis fst-conv full-def inference-preserves-un-sat* *rtrancplp-preserves-sat*
 satisfiable-carac' *satisfiable-def*)

lemma *rtrancplp-resolution-preserves-sat*:
 assumes *resolution** S S'*
 and *satisfiable* (*fst S*)
 shows *satisfiable* (*fst S'*)
 using *assms* **apply** (*induction rule: rtrancplp-induct*)
 apply *simp*
 using *resolution-preserves-sat* **by** *blast*

lemma *resolution-soundness*:
 fixes $\psi :: 'v :: \text{linorder state}$
 assumes *resolution** $\psi \psi'$* **and** $\{\#\} \in \text{fst } \psi'$
 shows *unsatisfiable* (*fst ψ*)
 using *assms* **by** (*meson rtrancplp-resolution-preserves-sat satisfiable-def true-cls-empty*
 true-clss-def)

lemma *resolution-soundness-and-completeness*:
fixes $\psi :: 'v :: \text{linorder state}$
assumes *finite: finite* (*fst ψ*)
and *snd: snd* $\psi = \{\}$
shows $(\exists \psi'. (\text{resolution** } \psi \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow \text{unsatisfiable } (\text{fst } \psi)$
 using *assms* *resolution-completeness* *resolution-soundness* **by** *metis*

lemma *simplified-falsity*:
 assumes *simp: simplified* ψ
 and $\{\#\} \in \psi$
 shows $\psi = \{\{\#\}\}$
proof (*rule ccontr*)
 assume $H: \neg ?thesis$
 then obtain χ **where** $\chi \in \psi$ **and** $\chi \neq \{\#\}$ **using** *assms*(2) **by** *blast*
 then have $\{\#\} \subsetneq \chi$ **by** (*simp add: mset-less-empty-nonempty*)
 then have *simplify* $\psi (\psi - \{\chi\})$
 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 *rtrancplp-simplify-falsity-in-preserved*:
 assumes *simplify*** $\chi s \chi s'$
 and $\{\#\} \in \chi s$
 shows $\{\#\} \in \chi s'$
 using *assms*

```

by induction (auto intro: simplify-falsity-in-preserved)

lemma resolution-falsity-get-falsity-alone:
  assumes finite (fst  $\psi$ )
  shows  $(\exists \psi'. (\text{resolution}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow (\exists a-u-v. \text{resolution}^{**} \psi (\{\#\}, a-u-v))$ 
    (is  $?A \longleftrightarrow ?B$ )
proof
  assume  $?B$ 
  then show  $?A$  by auto
next
  assume  $?A$ 
  then obtain  $\chi s$  a-u-v where  $\chi s: \text{resolution}^{**} \psi (\chi s, a-u-v)$  and  $F: \{\#\} \in \chi s$  by auto
  { assume simplified  $\chi s$ 
    then have  $?B$  using simplified-falsity[OF - F]  $\chi s$  by blast
  }
  moreover {
    assume  $\neg$  simplified  $\chi s$ 
    then obtain  $\chi s'$  where full1 simplify  $\chi s \chi s'$ 
      by (metis  $\chi s$  assms finite-simplified-full1-simp fst-conv rtranclp-resolution-finite)
    then have  $\{\#\} \in \chi s'$ 
      unfolding full1-def by (meson F rtranclp-simplify-falsity-in-preserved
        trancplp-into-rtranclp)
    then have  $?B$ 
      by (metis  $\chi s$  (full1 simplify  $\chi s \chi s'$ ) fst-conv full1-simp resolution-always-simplified
        rtranclp.rtrancl-into-rtrancl simplified-falsity)
  }
  ultimately show  $?B$  by blast
qed

lemma resolution-soundness-and-completeness':
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes
    finite: finite (fst  $\psi$ ) and
    snd: snd  $\psi = \{\}$ 
  shows  $(\exists a-u-v. (\text{resolution}^{**} \psi (\{\#\}, a-u-v))) \longleftrightarrow \text{unsatisfiable (fst } \psi)$ 
    using assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone
    by metis

```

end

theory Partial-Annotated-Clausal-Logic
 imports Partial-Clausal-Logic

begin

13 Partial Clausal Logic

We here define marked literals (that will be used in both DPLL and CDCL) and the entailment corresponding to it.

13.1 Marked Literals

13.1.1 Definition

datatype ($'v$, $'wl$, $'mark$) marked-lit =

is-marked: *Marked* (*lit-of*: 'v literal) (*level-of*: 'lvl) |
is-proped: *Propagated* (*lit-of*: 'v literal) (*mark-of*: 'mark)

lemma *marked-lit-list-induct*[*case-names nil marked proped*]:

assumes $P \square$ **and**
 $\bigwedge L \ l \ xs. P \ xs \implies P \ (\text{Marked } L \ l \ \# \ xs)$ **and**
 $\bigwedge L \ m \ xs. P \ xs \implies P \ (\text{Propagated } L \ m \ \# \ xs)$
shows $P \ xs$
using *assms apply* (*induction xs, simp*)
by (*rename-tac a xs, case-tac a*) *auto*

lemma *is-marked-ex-Marked*:

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*]:

lits-of $\square = \{\}$ **unfolding** *lits-of-def* **by** *auto*

lemma *lits-of-cons*[*simp*]:

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*]:

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*]:

set (*map* *lit-of* T) = *lits-of* T
unfolding *lits-of-def* **by** *auto*

abbreviation *unmark* **where**

unmark $M \equiv (\lambda a. \ \{\#\text{lit-of } a\#\}) \ ' \ \text{set } M$

lemma *atms-of-ms-lambda-lit-of-is-atm-of-lit-of*[*simp*]:

atms-of-ms (*unmark* M) = *atm-of* ' *lits-of* M '
unfolding *atms-of-ms-def lits-of-def* **by** *auto*

lemma *lits-of-empty-is-empty*[*iff*]:

lits-of $M = \{\} \longleftrightarrow M = \square$
by (*induct M*) *auto*

13.1.2 Entailment

definition *true-annot* :: (*'a, 'l, 'm*) *marked-lits* \Rightarrow '*a clause* \Rightarrow *bool* (*infix* \models_a 49) **where**

$I \models_a C \longleftrightarrow (\text{lits-of } I) \models C$

definition *true-annots* :: ('a, 'l, 'm) marked-lits \Rightarrow 'a clauses \Rightarrow bool (**infix** \models_{as} 49) **where**
 $I \models_{as} CC \longleftrightarrow (\forall C \in CC. I \models_a C)$

lemma *true-annot-empty-model[simp]*:
 $\neg [] \models_a \psi$
unfolding *true-annot-def true-cl-def* **by** *simp*

lemma *true-annot-empty[simp]*:
 $\neg I \models_a \{\#\}$
unfolding *true-annot-def true-cl-def* **by** *simp*

lemma *empty-true-annots-def[iff]*:
 $[] \models_{as} \psi \longleftrightarrow \psi = \{\}$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-empty[simp]*:
 $I \models_{as} \{\}$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-single-true-annot[iff]*:
 $I \models_{as} \{C\} \longleftrightarrow I \models_a C$
unfolding *true-annots-def* **by** *auto*

lemma *true-annot-insert-l[simp]*:
 $M \models_a A \implies L \# M \models_a A$
unfolding *true-annot-def* **by** *auto*

lemma *true-annots-insert-l [simp]*:
 $M \models_{as} A \implies L \# M \models_{as} A$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-union[iff]*:
 $M \models_{as} A \cup B \longleftrightarrow (M \models_{as} A \wedge M \models_{as} B)$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-insert[iff]*:
 $M \models_{as} \text{insert } a \ A \longleftrightarrow (M \models_a a \wedge M \models_{as} A)$
unfolding *true-annots-def* **by** *auto*

Link between \models_{as} and \models_s :

lemma *true-annots-true-cl*:
 $I \models_{as} CC \longleftrightarrow (\text{lits-of } I) \models_s CC$
unfolding *true-annots-def Ball-def true-annot-def true-clss-def* **by** *auto*

lemma *in-lit-of-true-annot*:
 $a \in \text{lits-of } M \longleftrightarrow M \models_a \{\#a\#\}$
unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annot-lit-of-notin-skip*:
 $L \# M \models_a A \implies \text{lit-of } L \notin \# A \implies M \models_a A$
unfolding *true-annot-def true-cl-def* **by** *auto*

lemma *true-clss-singleton-lit-of-implies-incl*:

$I \models_s \text{unmark } MLs \implies \text{lits-of } MLs \subseteq I$
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]*:

$\text{map } (\lambda M. \text{Marked } M \ a) \ M \models_{as} N \iff \text{set } M \models_s N$

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]*: $M \models_a \{\#L\# \} \iff L \in \text{lits-of } M$

unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annots-true-clss-clss*:

$A \models_{as} \Psi \implies \text{unmark } A \models_{ps} \Psi$
unfolding *true-clss-clss-def true-annots-def true-clss-def*
by (*auto*
dest!: true-clss-singleton-lit-of-implies-incl
simp add: lits-of-def true-annot-def true-clss-def)

lemma *true-annot-commute*:

$M @ M' \models_a D \iff M' @ M \models_a D$
unfolding *true-annot-def* **by** (*simp add: Un-commute*)

lemma *true-annots-commute*:

$M @ M' \models_{as} D \iff M' @ M \models_{as} D$
unfolding *true-annots-def* **by** (*auto simp add: true-annot-commute*)

lemma *true-annot-mono[dest]*:

$\text{set } I \subseteq \text{set } I' \implies I \models_a N \implies I' \models_a N$
using *true-clss-mono-set-mset-l* **unfolding** *true-annot-def lits-of-def*
by (*metis (no-types) Un-commute Un-upper1 image-Un sup.orderE*)

lemma *true-annots-mono*:

$\text{set } I \subseteq \text{set } I' \implies I \models_{as} N \implies I' \models_{as} N$
unfolding *true-annots-def* **by** *auto*

13.1.3 Defined and undefined literals

definition *defined-lit* :: $('a, 'l, 'm) \text{ marked-lit list} \Rightarrow 'a \text{ literal} \Rightarrow \text{bool}$

where

$\text{defined-lit } I \ L \iff (\exists l. \text{Marked } L \ l \in \text{set } I) \vee (\exists P. \text{Propagated } L \ P \in \text{set } I)$
 $\vee (\exists l. \text{Marked } (-L) \ l \in \text{set } I) \vee (\exists P. \text{Propagated } (-L) \ P \in \text{set } I)$

abbreviation *undefined-lit* :: ('a, 'l, 'm) *marked-lit list* \Rightarrow 'a *literal* \Rightarrow bool
where *undefined-lit* I L $\equiv \neg$ *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
 $(\exists l. \text{Marked } (\neg \text{lit-of } x) l \in \text{set } I)$
 $\vee (\exists l. \text{Marked } (\text{lit-of } x) l \in \text{set } I)$
 $\vee (\exists l. \text{Propagated } (\neg \text{lit-of } x) l \in \text{set } I)$
 $\vee (\exists l. \text{Propagated } (\text{lit-of } x) l \in \text{set } I)$
using *assms* *marked-lit.exhaust-sel* **by** *metis*

lemma *literal-is-lit-of-marked*:
assumes $L = \text{lit-of } x$
shows $(\exists l. x = \text{Marked } L l) \vee (\exists l'. x = \text{Propagated } L l')$
using *assms* **by** (*cases* x) *auto*

lemma *true-annot-iff-marked-or-true-lit*:
defined-lit I L $\longleftrightarrow ((\text{lits-of } I) \models L \vee (\text{lits-of } I) \models \neg L)$
unfolding *defined-lit-def* **by** (*auto simp add: lits-of-def rev-image-eqI*
dest!: literal-is-lit-of-marked)

lemma *consistent-interp* (*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 $\longleftrightarrow \text{atm-of } L \in (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } Ls$
unfolding *defined-lit-def* **apply** (*rule iffI*)
using *image-iff* **apply** *fastforce*
by (*fastforce simp add: atm-of-eq-atm-of dest: atm-imp-marked-or-proped*)

lemma *defined-lit-uminus[iff]*:
defined-lit I ($\neg L$) \longleftrightarrow *defined-lit* I L
unfolding *defined-lit-def* **by** *auto*

lemma *Marked-Propagated-in-iff-in-lits-of*:
defined-lit I L $\longleftrightarrow (L \in \text{lits-of } I \vee \neg L \in \text{lits-of } I)$
unfolding *lits-of-def* *defined-lit-def*
by (*auto simp: rev-image-eqI*) (*rename-tac* x , *case-tac* x , *auto*) $+$

lemma *consistent-add-undefined-lit-consistent[simp]*:
assumes
consistent-interp (*lits-of* 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 *defined-lit* [] L
unfolding *defined-lit-def* **by** *simp*

13.2 Backtracking

```

fun backtrack-split :: ('v, 'l, 'm) marked-lits
  ⇒ ('v, 'l, 'm) marked-lits × ('v, 'l, 'm) marked-lits where
backtrack-split [] = ([], []) |
backtrack-split (Propagated L P # mlits) = apfst ((op #) (Propagated L P)) (backtrack-split mlits) |
backtrack-split (Marked L l # mlits) = ([], Marked L l # mlits)

```

lemma *backtrack-split-fst-not-marked*: $a \in \text{set } (\text{fst } (\text{backtrack-split } l)) \implies \neg \text{is-marked } a$
by (induct l rule: marked-lit-list-induct) auto

lemma *backtrack-split-snd-hd-marked*:
 $\text{snd } (\text{backtrack-split } l) \neq [] \implies \text{is-marked } (\text{hd } (\text{snd } (\text{backtrack-split } l)))$
by (induct l rule: marked-lit-list-induct) auto

lemma *backtrack-split-list-eq[simp]*:
 $\text{fst } (\text{backtrack-split } l) @ (\text{snd } (\text{backtrack-split } l)) = l$
by (induct l rule: marked-lit-list-induct) auto

lemma *backtrack-snd-empty-not-marked*:
 $\text{backtrack-split } M = (M'', []) \implies \forall l \in \text{set } M. \neg \text{is-marked } l$
by (metis append-Nil2 backtrack-split-fst-not-marked backtrack-split-list-eq snd-conv)

lemma *backtrack-split-some-is-marked-then-snd-has-hd*:
 $\exists l \in \text{set } M. \text{is-marked } l \implies \exists M' L' M''. \text{backtrack-split } M = (M'', L' \# M')$
by (metis backtrack-snd-empty-not-marked list.exhaust prod.collapse)

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma *backtrack-split-takeWhile-dropWhile*:
 $\text{backtrack-split } M = (\text{takeWhile } (\text{Not } o \text{ is-marked}) M, \text{dropWhile } (\text{Not } o \text{ is-marked}) M)$
proof (induct M)
case Nil **show** ?case **by** simp
next
case (Cons L M) **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
  ⇒ (('a, 'l, 'm) marked-lits × ('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) (rename-tac a xs, 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 = []  $\vee$  (is-marked (hd a)  $\wedge$  hd a  $\in$  set M)
  using assms apply (induct M arbitrary: a b rule: marked-lit-list-induct)
  apply auto[2]
  by (metis UnCI backtrack-split-snd-hd-marked get-all-marked-decomposition-backtrack-split

```


get-all-marked-decomposition-decomp *hd-in-set* *list.sel(1)* *set-append* *snd-conv*)

lemma *get-all-marked-decomposition-snd-not-marked*:
assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
and $L \in \text{set } b$
shows $\neg \text{is-marked } L$
using *assms* **apply** (*induct* *M* *arbitrary*: *a* *b* *rule*: *marked-lit-list-induct*, *simp*)
by (*rename-tac* *L' l xs a b*, *case-tac* *get-all-marked-decomposition xs*; *fastforce*)**+**

lemma *tl-get-all-marked-decomposition-skip-some*:
assumes $x \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } M1))$
shows $x \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } (M0 @ M1)))$
using *assms*
by (*induct* *M0* *rule*: *marked-lit-list-induct*)
(auto simp add: list.set-sel(2))

lemma *hd-get-all-marked-decomposition-skip-some*:
assumes $(x, y) = \text{hd } (\text{get-all-marked-decomposition } M1)$
shows $(x, y) \in \text{set } (\text{get-all-marked-decomposition } (M0 @ \text{Marked } K \ i \ \# \ M1))$
using *assms*
proof (*induct* *M0*)
case *Nil*
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*
by (*cases* *get-all-marked-decomposition* (*M0 @ Marked K i # M1*))
(auto dest!: get-all-marked-decomposition-decomp
arg-cong[of get-all-marked-decomposition - - hd])
qed
qed

lemma *get-all-marked-decomposition-snd-union*:
 $\text{set } M = \bigcup (\text{set 'snd ' set } (\text{get-all-marked-decomposition } M)) \cup \{L \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$
(is ?M M = ?U M \cup ?Ls M)
proof (*induct M arbitrary*:)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons L M*)
show *?case*
proof (*cases L*)
case (*Marked a l*) **note** $L = \text{this}$
hence $L \in ?Ls (L \# M)$ **by** *auto*
moreover **have** $?U (L \# M) = ?U M$ **unfolding** *L* **by** *auto*
moreover **have** $?M M = ?U M \cup ?Ls M$ **using** *Cons.hyps* **by** *auto*
ultimately **show** *?thesis* **by** *auto*

```

next
  case (Propagated a P)
  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 (rename-tac L' m xs, case-tac get-all-marked-decomposition (xs @ M')) auto

lemma *get-all-marked-decomposition-remove-unmarked-length:*

assumes $\forall l \in \text{set } M'. \neg \text{is-marked } l$

shows $\text{length } (\text{get-all-marked-decomposition } (M' @ M''))$

$= \text{length } (\text{get-all-marked-decomposition } M'')$

using *assms* **by** (induct M' arbitrary: M'' rule: marked-lit-list-induct) auto

lemma *get-all-marked-decomposition-not-is-marked-length:*

assumes $\forall l \in \text{set } M'. \neg \text{is-marked } l$

shows $1 + \text{length } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M))$

$= \text{length } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M))$

using *assms* *get-all-marked-decomposition-remove-unmarked-length* **by** fastforce

lemma *get-all-marked-decomposition-last-choice:*

assumes $\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)) \neq []$

and $\forall l \in \text{set } M'. \neg \text{is-marked } l$

and $\text{hd } (\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M))) = (M0', M0)$

shows $\text{hd } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M)) = (M0', \text{Propagated } (-L) P \# M0)$

using *assms* **by** (induct M' rule: marked-lit-list-induct) auto

lemma *get-all-marked-decomposition-except-last-choice-equal:*

assumes $\forall l \in \text{set } M'. \neg \text{is-marked } l$

shows $\text{tl } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M))$

$= \text{tl } (\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L l \# M)))$

using *assms* **by** (induct M' rule: marked-lit-list-induct) auto

lemma *get-all-marked-decomposition-hd-hd:*

assumes $\text{get-all-marked-decomposition } Ls = (M, C) \# (M0, M0') \# l$

shows $\text{tl } M = M0' @ M0 \wedge \text{is-marked } (\text{hd } M)$

using *assms*

proof (induct Ls arbitrary: M C M0 M0' l)

case Nil

thus ?case **by** simp

next

case (Cons a Ls M C M0 M0' l) **note** IH = this(1) **and** g = this(2)

{ **fix** L level

assume a: a = Marked L level

have Ls = M0' @ M0

using g a **by** (force intro: get-all-marked-decomposition-decomp)

hence $\text{tl } M = M0' @ M0 \wedge \text{is-marked } (\text{hd } M)$ **using** g a **by** auto

}

```

moreover {
  fix L P
  assume a: a = Propagated L P
  have tl M = M0' @ M0  $\wedge$  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)  $\in$  set (get-all-marked-decomposition M)
  shows  $\exists$  c. M = c @ b @ a
  using assms apply (induct M rule: marked-lit-list-induct)
  apply simp
  by (rename-tac L' m xs, case-tac get-all-marked-decomposition xs;
    auto dest!: arg-cong[of get-all-marked-decomposition - - hd]
    get-all-marked-decomposition-decomp)+

```

```

lemma get-all-marked-decomposition-incl:
  assumes (a, b)  $\in$  set (get-all-marked-decomposition M)
  shows set b  $\subseteq$  set M and set a  $\subseteq$  set M
  using assms get-all-marked-decomposition-exists-prepend by fastforce+

```

```

lemma get-all-marked-decomposition-exists-prepend':
  assumes (a, b)  $\in$  set (get-all-marked-decomposition M)
  obtains c where M = c @ b @ a
  using assms apply (induct M rule: marked-lit-list-induct)
  apply auto[1]
  by (rename-tac L' m xs, case-tac hd (get-all-marked-decomposition xs),
    auto dest!: get-all-marked-decomposition-decomp simp add: list.set-sel(2))+

```

```

lemma union-in-get-all-marked-decomposition-is-subset:
  assumes (a, b)  $\in$  set (get-all-marked-decomposition M)
  shows set a  $\cup$  set b  $\subseteq$  set M
  using assms by force

```

definition *all-decomposition-implies* :: 'a literal multiset set
 $\Rightarrow ((\text{'a, 'l, 'm}) \text{ marked-lit list} \times (\text{'a, 'l, 'm}) \text{ marked-lit list}) \text{ list} \Rightarrow \text{bool})$ **where**
all-decomposition-implies N S
 $\longleftrightarrow (\forall (Ls, seen) \in \text{set } S. \text{unmark } Ls \cup N \models_{ps} \text{unmark } seen)$

```

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)]
   $\longleftrightarrow \text{unmark } Ls \cup N \models_{ps} \text{unmark } seen$ 
  unfolding all-decomposition-implies-def by auto

```

```

lemma all-decomposition-implies-append[iff]:
  all-decomposition-implies N (S @ S')
   $\longleftrightarrow (\text{all-decomposition-implies } N S \wedge \text{all-decomposition-implies } N S')$ 
  unfolding all-decomposition-implies-def by auto

```

lemma *all-decomposition-implies-cons-pair*[iff]:
all-decomposition-implies N $((Ls, seen) \# S')$
 $\longleftrightarrow (all-decomposition-implies\ N\ [(Ls, seen)] \wedge all-decomposition-implies\ N\ S')$
unfolding *all-decomposition-implies-def* **by** *auto*

lemma *all-decomposition-implies-cons-single*[iff]:
all-decomposition-implies N $(l \# S') \longleftrightarrow$
 $(unmark\ (fst\ l) \cup N \models_{ps} unmark\ (snd\ l) \wedge$
 $all-decomposition-implies\ N\ S')$
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 $N \cup \{\{\#lit-of\ L\# \mid L.\ is-marked\ L \wedge L \in set\ M\}\}$
 $\models_{ps} (\lambda a. \{\#lit-of\ a\# \}) \text{ ' } \bigcup (set \text{ ' } snd \text{ ' } set\ (get-all-marked-decomposition\ M))$
using *assms*
proof (*induct length (get-all-marked-decomposition M) arbitrary: M*)
case 0
thus ?case **by** *auto*
next
case (*Suc n*) **note** $IH = this(1)$ **and** $length = this(2)$
{
assume $length\ (get-all-marked-decomposition\ M) \leq 1$
then obtain $a\ b$ **where** $g: get-all-marked-decomposition\ M = (a, b) \# []$
by (*cases get-all-marked-decomposition M*) *auto*
moreover {
assume $a = []$
hence ?case **using** *Suc.prem*s g **by** *auto*
}
moreover {
assume $l: length\ a = 1$ **and** $m: is-marked\ (hd\ a)$ **and** $hd: hd\ a \in set\ M$
hence $(\lambda a. \{\#lit-of\ a\# \})\ (hd\ a) \in \{\{\#lit-of\ L\# \mid L.\ is-marked\ L \wedge L \in set\ M\}\}$ **by** *auto*
hence $H: unmark\ a \cup N \subseteq N \cup \{\{\#lit-of\ L\# \mid L.\ is-marked\ L \wedge L \in set\ M\}\}$
using l **by** (*cases a*) *auto*
have $f1: (\lambda m. \{\#lit-of\ m\# \}) \text{ ' } set\ a \cup N \models_{ps} (\lambda m. \{\#lit-of\ m\# \}) \text{ ' } set\ b$
using *Suc.prem*s **unfolding** *all-decomposition-implies-def* g **by** *simp*
have ?case
unfolding g **apply** (*rule true-clss-clss-subset*) **using** $f1\ H$ **by** *auto*
}
ultimately have ?case **using** *get-all-marked-decomposition-length-1-fst-empty-or-length-1* **by** *blast*
}
moreover {
assume $length\ (get-all-marked-decomposition\ M) > 1$
then obtain $Ls0\ seen0\ M'$ **where**
 $Ls0: get-all-marked-decomposition\ M = (Ls0, seen0) \# get-all-marked-decomposition\ M'$ **and**
 $length': length\ (get-all-marked-decomposition\ M') = n$ **and**
 $M'-in-M: set\ M' \subseteq set\ M$
using $length$ **apply** (*induct M*)
apply *simp*
by (*rename-tac a M, 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.prem*s **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) (rename-tac a xs, 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-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M' }
     $\models_{ps}$  ( $\lambda a. \{ \#lit-of a \# \}$ ) '  $\bigcup$  (set ' snd ' set (get-all-marked-decomposition M'))
    using IH length' by auto

  have l: N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M' }
     $\subseteq$  N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M }
    using M'-in-M by auto
  hence  $\Psi N$ : N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M }
     $\models_{ps}$  ( $\lambda a. \{ \#lit-of a \# \}$ ) '  $\bigcup$  (set ' snd ' set (get-all-marked-decomposition M'))
    using true-clss-clss-subset[OF l N] by auto
  have is-marked (hd Ls0) and LS: tl Ls0 = seen1 @ Ls1
    using get-all-marked-decomposition-hd-hd[of M] unfolding Ls0 Ls1 by auto

  have LSM: seen1 @ Ls1 = M' using get-all-marked-decomposition-decomp[of M] Ls1 by auto
  have M': set M' = Union (set ' snd ' set (get-all-marked-decomposition M'))
     $\cup$  { L | L. is-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-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M }  $\models_p$  ( $\lambda a. \{ \#lit-of a \# \}$ ) (hd Ls0)
      using (is-marked (hd Ls0)) by (metis (mono-tags, lifting) UnCI mem-Collect-eq
        true-clss-clss-in)
  } note hd-Ls0 = this

  have l: ( $\lambda a. \{ \#lit-of a \# \}$ ) ' ( $\bigcup$  (set ' snd ' set (get-all-marked-decomposition M'))
     $\cup$  { L | L. is-marked L  $\wedge$  L  $\in$  set M' })
    = ( $\lambda a. \{ \#lit-of a \# \}$ ) '
       $\bigcup$  (set ' snd ' set (get-all-marked-decomposition M'))
       $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M' }
    by auto
  have N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M' }  $\models_{ps}$ 
    ( $\lambda a. \{ \#lit-of a \# \}$ ) ' ( $\bigcup$  (set ' snd ' set (get-all-marked-decomposition M'))
     $\cup$  { L | L. is-marked L  $\wedge$  L  $\in$  set M' })
    unfolding l using N by (auto simp add: all-in-true-clss-clss)
  hence N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M' }  $\models_{ps}$  unmark (tl Ls0)
    using M' unfolding LS LSM by auto
  hence t: N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M' }
     $\models_{ps}$  unmark (tl Ls0)
    by (blast intro: all-in-true-clss-clss)
  hence N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M }
     $\models_{ps}$  unmark (tl Ls0)
    using M'-in-M true-clss-clss-subset[OF - t,
      of N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M' }]
    by auto
  hence N  $\cup$  { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set M }  $\models_{ps}$  unmark Ls0

```

```

using hd-Ls0 by (cases Ls0, auto)

moreover have unmark Ls0  $\cup N \models_{ps}$  unmark 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  $\Psi: N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{ is-marked } L \wedge L \in \text{set } M\} \models_{ps}$ 
  unmark seen0
  by (meson true-clss-clss-left-right true-clss-clss-union-and true-clss-clss-union-l-r)
have  $(\lambda a. \{\#lit\text{-of } a\# \}) '(\text{set seen0}$ 
   $\cup (\bigcup x \in \text{set } (get\text{-all-marked-decomposition } M'). \text{set } (snd\ x)))$ 
   $= \text{unmark seen0}$ 
   $\cup (\lambda a. \{\#lit\text{-of } a\# \}) '(\bigcup x \in \text{set } (get\text{-all-marked-decomposition } M'). \text{set } (snd\ x))$ 
  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  $N \cup \{\{\#lit\text{-of } L\# \} \mid L. \text{ is-marked } L \wedge L \in \text{set } M\} \models_{ps} \text{unmark } M$ 
  (is ?I  $\models_{ps}$  ?A)
proof –
  have ?I  $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) ' \{L \mid L. \text{ is-marked } L \wedge L \in \text{set } M\}$ 
  by (auto intro: all-in-true-clss-clss)
  moreover have ?I  $\models_{ps} (\lambda a. \{\#lit\text{-of } a\# \}) ' \bigcup (\text{set } 'snd ' \text{set } (get\text{-all-marked-decomposition } M))$ 
  using all-decomposition-implies-trail-is-implied assms by blast
  ultimately have  $N \cup \{\{\#lit\text{-of } m\# \} \mid m. \text{ is-marked } m \wedge m \in \text{set } M\}$ 
   $\models_{ps} (\lambda m. \{\#lit\text{-of } m\# \}) ' \bigcup (\text{set } 'snd ' \text{set } (get\text{-all-marked-decomposition } M))$ 
   $\cup (\lambda m. \{\#lit\text{-of } m\# \}) ' \{m \mid m. \text{ is-marked } m \wedge m \in \text{set } M\}$ 
  by blast
  thus ?thesis
  by (metis (no-types) get-all-marked-decomposition-snd-union[of M] image-Un)
qed

lemma all-decomposition-implies-insert-single:
  all-decomposition-implies N M  $\implies$  all-decomposition-implies (insert C N) M
  unfolding all-decomposition-implies-def by auto

```

13.4 Negation of Clauses

definition *CNot* :: *'v clause* \Rightarrow *'v clauses* **where**
CNot $\psi = \{ \{\#-L\# \} \mid L. L \in \# \psi \}$

lemma *in-CNot-uminus[iff]:*
shows $\{\#L\# \} \in CNot\ \psi \longleftrightarrow -L \in \# \psi$
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** $-L \in lits\text{-}of\ M$
using *assms* **by** (*auto simp add: true-annot-def true-annot-def CNot-def*)

lemma *CNot-remdups-mset[simp]*:
 $CNot\ (remdups\text{-}mset\ A) = CNot\ A$
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 {φ}* **and** $\neg I \models \varphi$
shows $I \models_s CNot\ \varphi$
using *assms* **unfolding** *total-over-m-def total-over-set-def true-clss-def true-clss-def CNot-def*
apply *clarify*
by (*rename-tac x L, case-tac L*) (*force intro: pos-lit-in-atms-of neg-lit-in-atms-of*)**+**

lemma *total-not-CNot*:
assumes *total-over-m I {φ}* **and** $\neg I \models_s CNot\ \varphi$
shows $I \models \varphi$
using *assms* *total-not-true-clss-true-clss-CNot* **by** *auto*

lemma *atms-of-ms-CNot-atms-of[simp]*:
 $atms\text{-}of\text{-}ms\ (CNot\ C) = atms\text{-}of\ C$
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 $atm\text{-}of\ L \in atm\text{-}of\ \text{'lits-of}\ M$
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 ∪ CNot {#L#}) and
  cons: consistent-interp I and
  I: I ⊨s B
have total-over-m I ({#L#} ∪ B) using tot by auto
hence ¬I ⊨s insert {#L#} B
  using assms cons unfolding true-clss-clss-def by simp
thus I ⊨s CNot {#L#}
  using tot I by (cases L) auto
qed

lemma true-annots-true-clss-def-iff-negation-in-model:
  M ⊨as CNot C ⟷ (∀ L ∈ # C. ¬L ∈ lits-of M)
  unfolding CNot-def true-annots-true-clss true-clss-def by auto

lemma consistent-CNot-not-tautology:
  consistent-interp M ⟹ M ⊨s CNot D ⟹ ¬tautology D
  by (metis atms-of-ms-CNot-atms-of consistent-CNot-not satisfiable-carac' satisfiable-def
    tautology-def total-over-m-def)

lemma atms-of-ms-CNot-atms-of-ms: atms-of-ms (CNot CC) = atms-of-ms {CC}
  by simp

lemma total-over-m-CNot-total-over-m[simp]:
  total-over-m I (CNot C) = total-over-set I (atms-of C)
  unfolding total-over-m-def total-over-set-def by auto

lemma uminus-lit-swap: ¬(a::'a literal) = i ⟷ a = -i
  by auto

lemma true-clss-clss-plus-CNot:
  assumes CC-L: A ⊨p CC + {#L#}
  and CNot-CC: A ⊨ps CNot CC
  shows A ⊨p {#L#}
  unfolding true-clss-clss-def true-clss-clss-def CNot-def total-over-m-def
proof (intro allI impI)
  fix I
  assume tot: total-over-set I (atms-of-ms (A ∪ {{#L#}}))
  and cons: consistent-interp I
  and I: I ⊨s A
  let ?I = I ∪ {Pos P | P. P ∈ atms-of CC ∧ P ∉ atm-of ' I}
  have cons': consistent-interp ?I
    using cons unfolding consistent-interp-def
    by (auto simp add: uminus-lit-swap atms-of-def rev-image-eqI)
  have I': ?I ⊨s A
    using I true-clss-union-increase by blast
  have tot-CNot: total-over-m ?I (A ∪ 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 ∪ {CC + {#L#}})
    using tot unfolding total-over-m-def total-over-set-atm-of by auto
  hence ?I ⊨ CC + {#L#} using CC-L cons' I' unfolding true-clss-clss-def by blast
  moreover
    have ?I ⊨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 $H: \forall x. x \in CNot A \longrightarrow L \# M \models_a x$ and $l: l \in CNot A$
 hence $L \# M \models_a l$ by auto
 thus $M \models_a l$ using LA l by (cases L) (auto simp add: *CNot-def*)
 qed

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:
 $A \cup \{B\} \models_{ps} \{\{\#\}\} \longleftrightarrow A \models_{ps} CNot B$
 using *total-not-CNot consistent-CNot-not* **unfolding** *total-over-m-def true-clss-clss-def*
 by fastforce

lemma *true-annot-remove-hd-if-notin-vars*:
 assumes $a \# M' \models_a D$
 and *atm-of* (*lit-of* a) \notin *atms-of* D
 shows $M' \models_a D$
 using *assms true-clss-remove-hd-if-notin-vars* **unfolding** *true-annot-def* by auto

lemma *true-annot-remove-if-notin-vars*:
 assumes $M @ M' \models_a D$
 and $\forall x \in \text{atms-of } D. x \notin \text{atm-of ' lits-of } M$
 shows $M' \models_a D$
 using *assms* **apply** (induct M , *simp*)
 using *true-annot-remove-hd-if-notin-vars* by force+

lemma *true-annots-remove-if-notin-vars*:
 assumes $M @ M' \models_{as} D$
 and $\forall x \in \text{atms-of-ms } D. x \notin \text{atm-of ' lits-of } M$
 shows $M' \models_{as} D$ **unfolding** *true-annots-def*
 using *assms true-annot-remove-if-notin-vars*[*of* M M']
unfolding *true-annots-def atms-of-ms-def* by force

lemma *all-variables-defined-not-imply-cnot*:
 assumes $\forall s \in \text{atms-of-ms } \{B\}. s \in \text{atm-of ' lits-of } A$
 and $\neg A \models_a B$
 shows $A \models_{as} CNot B$
unfolding *true-annot-def true-annots-def Ball-def CNot-def true-lit-def*
proof (clarify, rule ccontr)
 fix L
 assume $LB: L \in \# B$ and $\neg \text{lits-of } A \models_l \neg L$
 hence *atm-of* $L \in \text{atm-of ' lits-of } A$

```

    using assms(1) by (simp add: atm-of-lit-in-atms-of lits-of-def)
  hence  $L \in \text{lits-of } A \vee -L \in \text{lits-of } A$ 
    using atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set bymetis
  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]:
  no-dup (rev  $M$ )  $\longleftrightarrow$  no-dup  $M$ 
  by (auto simp: rev-map[symmetric])

```

```

lemma no-dup-length-eq-card-atm-of-lits-of:
  assumes 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:
  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:  $\text{atm-of } (\text{lit-of } L) \notin (\lambda l. \text{atm-of } (\text{lit-of } l)) ' \text{set } M$  using Cons.prem by auto
  have undefined-lit  $M$  (lit-of  $L$ )
    using a2 image-iff unfolding defined-lit-def by fastforce
  thus ?case
    using a1 by simp
qed

```

```

lemma distinct-get-all-marked-decomposition-no-dup:
  assumes ( $a, b$ )  $\in \text{set } (\text{get-all-marked-decomposition } M)$ 
  and no-dup  $M$ 
  shows no-dup ( $a @ b$ )
  using assms by force

```

```

lemma true-annot-lit-of-notin-skip:
  assumes  $L \# M \models_{\text{as}} \text{CNot } A$ 
  and  $-\text{lit-of } L \notin \# A$ 
  and no-dup ( $L \# M$ )
  shows  $M \models_{\text{as}} \text{CNot } A$ 
proof -
  have  $\forall l \in \# A. -l \in \text{lits-of } (L \# M)$ 
    using assms(1) in-CNot-implies-uminus(2) by blast
  moreover

```

have *atm-of* (*lit-of* *L*) \notin *atm-of* ' *lits-of* *M*
using *assms*(3) **unfolding** *lits-of-def* **by** *force*
hence \neg *lit-of* *L* \notin *lits-of* *M* **unfolding** *lits-of-def*
by (*metis* (*no-types*) *atm-of-uminus* *imageI*)
ultimately have $\forall l \in \# A. \neg l \in \text{lits-of } M$
using *assms*(2) **unfolding** *Ball-mset-def* **by** (*metis insertE lits-of-cons uminus-of-uminus-id*)
thus *?thesis* **by** (*auto simp add: true-annots-def*)
qed

type-synonym 'v *clauses* = 'v *clause multiset*

abbreviation *true-annots-mset* (**infix** \models_{asm} 50) **where**
 $I \models_{asm} C \equiv I \models_{as} (\text{set-mset } C)$

abbreviation *true-clss-clss-m*:: 'a *clauses* \Rightarrow 'a *clauses* \Rightarrow *bool* (**infix** \models_{psm} 50) **where**
 $I \models_{psm} C \equiv \text{set-mset } I \models_{ps} (\text{set-mset } C)$

Analog of $\llbracket ?N \models_{ps} ?B; ?A \subseteq ?B \rrbracket \Longrightarrow ?N \models_{ps} ?A$

lemma *true-clss-clssm-subsetE*: $N \models_{psm} B \Longrightarrow A \subseteq \# B \Longrightarrow N \models_{psm} A$
using *set-mset-mono true-clss-clss-subsetE* **by** *blast*

abbreviation *true-clss-clss-m*:: 'a *clauses* \Rightarrow 'a *clause* \Rightarrow *bool* (**infix** \models_{pm} 50) **where**
 $I \models_{pm} C \equiv \text{set-mset } I \models_p C$

abbreviation *distinct-mset-mset* :: 'a *multiset multiset* \Rightarrow *bool* **where**
 $\text{distinct-mset-mset } \Sigma \equiv \text{distinct-mset-set } (\text{set-mset } \Sigma)$

abbreviation *all-decomposition-implies-m* **where**
 $\text{all-decomposition-implies-m } A B \equiv \text{all-decomposition-implies } (\text{set-mset } A) B$

abbreviation *atms-of-msu* **where**
 $\text{atms-of-msu } U \equiv \text{atms-of-ms } (\text{set-mset } U)$

abbreviation *true-clss-m*:: 'a *interp* \Rightarrow 'a *clauses* \Rightarrow *bool* (**infix** \models_{sm} 50) **where**
 $I \models_{sm} C \equiv I \models_s \text{set-mset } C$

abbreviation *true-clss-ext-m* (**infix** \models_{sextm} 49) **where**
 $I \models_{sextm} C \equiv I \models_{sext} \text{set-mset } C$

end

theory *CDCL-NOT*

imports *Partial-Annotated-Clausal-Logic List-More Wellfounded-More Partial-Clausal-Logic*
begin

14 NOT's CDCL

sledgehammer-params[*verbose, prover=e spass z3 cvc4 verit remote-vampire*]

declare *set-mset-minus-replicate-mset*[*simp*]

14.1 Auxiliary Lemmas and Measure

lemma *no-dup-cannot-not-lit-and-uminus*:
 $\text{no-dup } M \Longrightarrow \neg \text{lit-of } xa = \text{lit-of } x \Longrightarrow x \in \text{set } M \Longrightarrow xa \notin \text{set } M$
by (*metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id'*)

lemma *true-clss-single-iff-incl*:

$I \models_s \text{single} \text{ ' } B \longleftrightarrow B \subseteq I$

unfolding *true-clss-def* **by** *auto*

lemma *atms-of-ms-single-atm-of[simp]*:

$\text{atms-of-ms } \{\{\# \text{lit-of } L\# \} \mid L. P L\} = \text{atm-of ' } \{\text{lit-of } L \mid L. P L\}$

unfolding *atms-of-ms-def* **by** *auto*

lemma *atms-of-uminus-lit-atm-of-lit-of*:

$\text{atms-of } \{\# - \text{lit-of } x. x \in \# A\# \} = \text{atm-of ' } (\text{lit-of ' } (\text{set-mset } A))$

unfolding *atms-of-def* **by** (*auto simp add: Fun.image-comp*)

lemma *atms-of-ms-single-image-atm-of-lit-of*:

$\text{atms-of-ms } ((\lambda x. \{\# \text{lit-of } x\# \}) \text{ ' } A) = \text{atm-of ' } (\text{lit-of ' } A)$

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 $\mu_C :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{nat}$ **where**

$\mu_C s b M \equiv (\sum i=0..<\text{length } M. M!i * b^\wedge (s + i - \text{length } M))$

lemma $\mu_C\text{-nil}[simp]$:

$\mu_C s b [] = 0$

unfolding $\mu_C\text{-def}$ **by** *auto*

lemma $\mu_C\text{-single}[simp]$:

$\mu_C s b [L] = L * b^\wedge (s - \text{Suc } 0)$

unfolding $\mu_C\text{-def}$ **by** *auto*

lemma *set-sum-atLeastLessThan-add*:

$(\sum i=k..<k+(b::\text{nat}). f i) = (\sum i=0..<b. f (k + i))$

by (*induction b*) *auto*

lemma *set-sum-atLeastLessThan-Suc*:

$(\sum i=1..<\text{Suc } j. f i) = (\sum i=0..<j. f (\text{Suc } i))$

using *set-sum-atLeastLessThan-add[of - 1 j]* **by** *force*

lemma $\mu_C\text{-cons}$:

$\mu_C s b (L \# M) = L * b^\wedge (s - 1 - \text{length } M) + \mu_C s b M$

proof –

have $\mu_C s b (L \# M) = (\sum i=0..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$

unfolding $\mu_C\text{-def}$ **by** *blast*

also have $\dots = (\sum i=0..<1. (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$

$+ (\sum i=1..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$

by (*rule setsum-add-nat-ivl[symmetric]*) *simp-all*

finally have $\mu_C s b (L \# M) = L * b^\wedge (s - 1 - \text{length } M)$

$+ (\sum i=1..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M)))$

by *auto*

moreover {

have $(\sum i=1..<\text{length } (L\#M). (L\#M)!i * b^\wedge (s + i - \text{length } (L\#M))) =$

$(\sum i=0..<\text{length } (M). (L\#M)!(\text{Suc } i) * b^\wedge (s + (\text{Suc } i) - \text{length } (L\#M)))$

unfolding *length-Cons set-sum-atLeastLessThan-Suc* **by** *blast*

also have $\dots = (\sum i=0..<\text{length } (M). M!i * b^\wedge (s + i - \text{length } M))$

by *auto*

finally have $(\sum_{i=1..<\text{length } (L\#M)}. (L\#M)!i * b^\wedge (s+i - \text{length } (L\#M))) = \mu_C s b M$
 unfolding $\mu_C\text{-def}$.
 }
 ultimately show ?thesis by presburger
 qed

lemma $\mu_C\text{-append}$:
 assumes $s \geq \text{length } (M@M')$
 shows $\mu_C s b (M@M') = \mu_C (s - \text{length } M') b M + \mu_C s b M'$
proof -
 have $\mu_C s b (M@M') = (\sum_{i=0..<\text{length } (M@M')}. (M@M')!i * b^\wedge (s+i - \text{length } (M@M')))$
 unfolding $\mu_C\text{-def}$ by blast
 moreover then have ... = $(\sum_{i=0..<\text{length } M}. (M@M')!i * b^\wedge (s+i - \text{length } (M@M')))$
 $+ (\sum_{i=\text{length } M..<\text{length } (M@M')}. (M@M')!i * b^\wedge (s+i - \text{length } (M@M')))$
 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 $\mu_C (s - \text{length } M') b M = (\sum_{i=0..<\text{length } M}. (M@M')!i * b^\wedge (s+i - \text{length } (M@M')))$
 (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 $b :: nat$
 assumes
 $b > 0$ and
 $M \neq []$ and
 $M\text{-le}: \forall i < \text{length } M. M!i < b$ and

```

  s ≥ length M
shows μC s b M < b^s
proof -
consider (b1) b = 1 | (b) b > 1 using ⟨b > 0⟩ by (cases b) auto
then show ?thesis
proof cases
case b1
then have ∀ i < length M. M!i = 0 using M-le by auto
then have μC s b M = 0 unfolding μC-def by auto
then show ?thesis using ⟨b > 0⟩ by auto
next
case b
have ∀ i ∈ {0..<length M}. M!i * b^(s + i - length M) ≤ (b-1) * b^(s + i - length M)
using M-le ⟨b > 1⟩ by auto
then have μC s b M ≤ (∑ i=0..<length M. (b-1) * b^(s + i - length M))
using ⟨M ≠ []⟩ ⟨b > 0⟩ unfolding μC-def by (auto intro: setsum-mono)
also
have ∀ i ∈ {0..<length M}. (b-1) * b^(s + i - length M) = (b-1) * b^i * b^(s - length M)
by (metis Nat.add-diff-assoc2 add.commute assms(4) mult.assoc power-add)
then have (∑ i=0..<length M. (b-1) * b^(s + i - length M))
= (∑ i=0..<length M. (b-1) * b^i * b^(s - length M))
by (auto simp add: ac-simps)
also have ... = (∑ i=0..<length M. b^i) * b^(s - length M) * (b-1)
by (simp add: setsum-left-distrib setsum-right-distrib ac-simps)
finally have μC s b M ≤ (∑ i=0..<length M. b^i) * (b-1) * b^(s - length M)
by (simp add: ac-simps)

also
have (∑ i=0..<length M. b^i) * (b-1) = b^(length M) - 1
using sum-of-powers[of b length M] ⟨b > 1⟩
by (auto simp add: ac-simps)
finally have μC s b M ≤ (b^(length M) - 1) * b^(s - length M)
by auto
also have ... < b^(length M) * b^(s - length M)
using ⟨b > 1⟩ by auto
also have ... = b^s
by (metis assms(4) le-add-diff-inverse power-add)
finally show ?thesis unfolding μC-def by (auto simp add: ac-simps)
qed
qed

```

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

lemma μ_C -bounded:

fixes $b :: \text{nat}$

assumes

$M\text{-le: } \forall i < \text{length } M. M!i < b$ **and**

$s \geq \text{length } M$

$b > 0$

shows $\mu_C s b M < b^{\wedge} s$

proof -

consider ($M0$) $M = []$ | (M) $b > 0$ **and** $M \neq []$

using $M\text{-le}$ **by** (cases b , cases M) *auto*

then show ?thesis

proof cases

```

    case M0
    then show ?thesis using M-le  $\langle b > 0 \rangle$  by auto
next
    case M
    show ?thesis using  $\mu_C$ -bounded-non-degenerated[OF M assms(1,2)] by arith
qed
qed

```

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

```

lemma  $\mu_C$ -base-0:
  assumes length M  $\leq s$ 
  shows  $\mu_C s 0 M \leq M!0$ 
proof -
  {
    assume s = length M
    moreover {
      fix n
      have  $(\sum_{i=0..<n}. M ! i * (0::nat) ^ i) \leq M ! 0$ 
      apply (induction n rule: nat-induct)
      by simp (rename-tac n, case-tac n, auto)
    }
    ultimately have ?thesis unfolding  $\mu_C$ -def by auto
  }
moreover
  {
    assume length M < s
    then have  $\mu_C s 0 M = 0$  unfolding  $\mu_C$ -def by auto
  }
ultimately show ?thesis using assms unfolding  $\mu_C$ -def by linarith
qed

```

14.2 Initial definitions

14.2.1 The state

We define here an abstraction over operation on the state we are manipulating.

```

locale dpll-state =
  fixes
    trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
    clauses :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
    tl-trail :: 'st  $\Rightarrow$  'st and
    add-clsNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    remove-clsNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st
  assumes
    trail-prepend-trail[simp]:
       $\bigwedge st L. \text{undefined-lit } (trail\ st) (lit-of\ L) \implies trail\ (prepend-trail\ L\ st) = L \# trail\ st$ 
    and
    tl-trail[simp]: trail (tl-trail S) = tl (trail S) and
    trail-add-clsNOT[simp]:  $\bigwedge st C. no-dup\ (trail\ st) \implies trail\ (add-cls_{NOT}\ C\ st) = trail\ st$  and
    trail-remove-clsNOT[simp]:  $\bigwedge st C. trail\ (remove-cls_{NOT}\ C\ st) = trail\ st$  and

    clauses-prepend-trail[simp]:
       $\bigwedge st L. \text{undefined-lit } (trail\ st) (lit-of\ L) \implies clauses\ (prepend-trail\ L\ st) = clauses\ st$ 
    and
    clauses-tl-trail[simp]:  $\bigwedge st. clauses\ (tl-trail\ st) = clauses\ st$  and

```

```

clauses-add-clNOT[simp]:
   $\bigwedge st\ C. \text{no-dup } (trail\ st) \implies \text{clauses } (add-cl_{NOT}\ C\ st) = \{\#C\# \} + \text{clauses } st$  and
clauses-remove-clNOT[simp]:  $\bigwedge st\ C. \text{clauses } (remove-cl_{NOT}\ C\ st) = \text{remove-mset } C\ (\text{clauses } st)$ 
begin

function reduce-trail-toNOT :: 'a list  $\Rightarrow$  'st  $\Rightarrow$  'st where
reduce-trail-toNOT F S =
  (if length (trail S) = length F  $\vee$  trail S = [] then S else reduce-trail-toNOT F (tl-trail S))
by fast+
termination by (relation measure ( $\lambda(-, S). \text{length } (trail\ S)$ )) auto
declare reduce-trail-toNOT.simps[simp del]

lemma
shows
reduce-trail-toNOT-nil[simp]: trail S = []  $\implies$  reduce-trail-toNOT F S = S and
reduce-trail-toNOT-eq-length[simp]: length (trail S) = length F  $\implies$  reduce-trail-toNOT F S = S
by (auto simp: reduce-trail-toNOT.simps)

lemma reduce-trail-toNOT-length-ne[simp]:
  length (trail S)  $\neq$  length F  $\implies$  trail S  $\neq$  []  $\implies$ 
    reduce-trail-toNOT F S = reduce-trail-toNOT F (tl-trail S)
by (auto simp: reduce-trail-toNOT.simps)

lemma trail-reduce-trail-toNOT-length-le:
assumes length F > length (trail S)
shows trail (reduce-trail-toNOT F S) = []
using assms by (induction F S rule: reduce-trail-toNOT.induct)
(simp add: less-imp-diff-less reduce-trail-toNOT.simps)

lemma trail-reduce-trail-toNOT-nil[simp]:
  trail (reduce-trail-toNOT [] S) = []
by (induction [] S rule: reduce-trail-toNOT.induct)
(simp add: less-imp-diff-less reduce-trail-toNOT.simps)

lemma clauses-reduce-trail-toNOT-nil:
  clauses (reduce-trail-toNOT [] S) = clauses S
by (induction [] S rule: reduce-trail-toNOT.induct)
(simp add: less-imp-diff-less reduce-trail-toNOT.simps)

lemma trail-reduce-trail-toNOT-drop:
  trail (reduce-trail-toNOT F S) =
    (if length (trail S)  $\geq$  length F
      then drop (length (trail S) - length F) (trail S)
      else [])
apply (induction F S rule: reduce-trail-toNOT.induct)
apply (rename-tac F S, case-tac trail S)
apply auto[]
apply (rename-tac list, case-tac Suc (length list) > length F)
prefer 2 apply simp
apply (subgoal-tac 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 ≡ map ((λl. 1 + length l) o snd) (get-all-marked-decomposition (trail S))

definition *state-eq_{NOT} :: 'st ⇒ 'st ⇒ bool (infix ~ 50) where*

S ~ T ⟷ trail S = trail T ∧ clauses S = clauses T

lemma *state-eq_{NOT}-ref[simp]*:

S ~ S

unfolding *state-eq_{NOT}-def by auto*

lemma *state-eq_{NOT}-sym*:

S ~ T ⟷ T ~ S

unfolding *state-eq_{NOT}-def by auto*

lemma *state-eq_{NOT}-trans*:

S ~ T ⟹ T ~ U ⟹ S ~ U

unfolding *state-eq_{NOT}-def by auto*

lemma

shows

state-eq_{NOT}-trail: S ~ T ⟹ trail S = trail T and

state-eq_{NOT}-clauses: S ~ T ⟹ 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 ⟹ 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 ~ T*

shows *reduce-trail-to_{NOT} F S ~ 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-cl_{NOT}[simp]*:

no-dup (trail S) ⟹

trail (reduce-trail-to_{NOT} F (add-cl_{NOT} C S)) = trail (reduce-trail-to_{NOT} F S)

by (rule trail-eq-reduce-trail-to_{NOT}-eq) simp

lemma reduce-trail-to_{NOT}-trail-tl-trail-decomp[simp]:

trail $S = F' @ \text{Marked } K () \# F \implies$

trail (reduce-trail-to_{NOT} F (tl-trail S)) = F

apply (rule reduce-trail-to_{NOT}-skip-beginning[of - tl ($F' @ \text{Marked } K () \# []$)])

by (cases F') (auto simp add:tl-append reduce-trail-to_{NOT}-skip-beginning)

end

14.2.2 Definition of the operation

locale propagate-ops =

dpll-state trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} **for**

trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**

clauses :: 'st \Rightarrow 'v clauses **and**

prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**

tl-trail :: 'st \Rightarrow 'st **and**

add-cl_{NOT} remove-cl_{NOT}:: 'v clause \Rightarrow 'st \Rightarrow 'st **and**

propagate-cond :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool

begin

inductive propagate_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool **where**

propagate_{NOT}[intro]: $C + \{\#L\} \in \# \text{ clauses } S \implies \text{trail } S \models_{\text{as}} C \text{Not } C$

$\implies \text{undefined-lit (trail } S) L$

$\implies \text{propagate-cond (Propagated } L ()) S$

$\implies T \sim \text{prepend-trail (Propagated } L ()) S$

$\implies \text{propagate}_{\text{NOT}} S T$

inductive-cases propagate_{NOT}E[elim]: propagate_{NOT} $S T$

end

locale decide-ops =

dpll-state trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} **for**

trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**

clauses :: 'st \Rightarrow 'v clauses **and**

prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**

tl-trail :: 'st \Rightarrow 'st **and**

add-cl_{NOT} remove-cl_{NOT}:: 'v clause \Rightarrow 'st \Rightarrow 'st

begin

inductive decide_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool **where**

decide_{NOT}[intro]: $\text{undefined-lit (trail } S) L \implies \text{atm-of } L \in \text{atms-of-msu (clauses } S)$

$\implies T \sim \text{prepend-trail (Marked } L ()) S$

$\implies \text{decide}_{\text{NOT}} S T$

inductive-cases decide_{NOT}E[elim]: decide_{NOT} $S S'$

end

locale backjumping-ops =

dpll-state trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}

for

trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**

clauses :: 'st \Rightarrow 'v clauses **and**

prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**

tl-trail :: 'st \Rightarrow 'st **and**

add-cl_{NOT} remove-cl_{NOT}:: 'v clause \Rightarrow 'st \Rightarrow 'st +

fixes

```

    backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive backjump where
trail S = F' @ Marked K () # F
 $\Rightarrow$  T  $\sim$  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 $atms-of C' \subseteq atms-of-ms N$ because to ensure that we can backjump even if the last decision variable has disappeared.

The part of the condition $atm-of L \in atm-of ' lits-of (F' @ Marked K () \# F)$ is important, otherwise you are not sure that you can backtrack.

14.3.1 Definition

We define dpll with backjumping:

inductive *dpll-bj* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**
*bj-decide*_{NOT}: *decide*_{NOT} *S S'* \Rightarrow *dpll-bj S S'* |
*bj-propagate*_{NOT}: *propagate*_{NOT} *S S'* \Rightarrow *dpll-bj S S'* |
bj-backjump: *backjump S S'* \Rightarrow *dpll-bj S S'*

lemmas *dpll-bj-induct* = *dpll-bj.induct[split-format(complete)]*
thm *dpll-bj-induct[OF dpll-with-backjumping-ops-axioms]*
lemma *dpll-bj-all-induct[consumes 2, case-names decide_{NOT} propagate_{NOT} backjump]*:
fixes *S T* :: 'st
assumes
dpll-bj S T and
inv S
 $\bigwedge L T. \text{undefined-lit } (\text{trail } S) L \Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S)$
 $\Rightarrow T \sim \text{prepend-trail } (\text{Marked } L ()) S$
 $\Rightarrow P S T$ **and**
 $\bigwedge C L T. C + \{\#L\} \in \# \text{ clauses } S \Rightarrow \text{trail } S \models_{\text{as}} C \text{Not } C \Rightarrow \text{undefined-lit } (\text{trail } S) L$
 $\Rightarrow T \sim \text{prepend-trail } (\text{Propagated } L ()) S$
 $\Rightarrow P S T$ **and**
 $\bigwedge C F' K F L C' T. C \in \# \text{ clauses } S \Rightarrow F' @ \text{Marked } K () \# F \models_{\text{as}} C \text{Not } C$
 $\Rightarrow \text{trail } S = F' @ \text{Marked } K () \# F$
 $\Rightarrow \text{undefined-lit } F L$
 $\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' (lits-of } (F' @ \text{Marked } K () \# F))$
 $\Rightarrow \text{clauses } S \models_{\text{pm}} C' + \{\#L\}$
 $\Rightarrow F \models_{\text{as}} C \text{Not } C'$
 $\Rightarrow T \sim \text{prepend-trail } (\text{Propagated } L ()) (\text{reduce-trail-to}_{\text{NOT}} F S)$
 $\Rightarrow P S T$
shows *P S T*
apply (*induct T rule: dpll-bj-induct[OF local.dpll-with-backjumping-ops-axioms]*)
apply (*rule assms(1)*)
using *assms(3)* **apply** *blast*
apply (*elim propagate_{NOT}E*) **using** *assms(4)* **apply** *blast*
apply (*elim backjumpE*) **using** *assms(5) <inv S>* **by** *simp*

14.3.2 Basic properties

First, some better suited induction principle **lemma** *dpll-bj-clauses*:

assumes *dpll-bj S T and inv S*
shows *clauses S = clauses T*
using *assms* **by** (*induction rule: dpll-bj-all-induct*) *auto*

No duplicates in the trail **lemma** *dpll-bj-no-dup*:

assumes *dpll-bj S T and inv S*
and *no-dup (trail S)*
shows *no-dup (trail T)*
using *assms* **by** (*induction rule: dpll-bj-all-induct*)
(auto simp add: defined-lit-map reduce-trail-to_{NOT}-skip-beginning)

Valuations **lemma** *dpll-bj-sat-iff*:

assumes *dpll-bj S T and inv S*
shows $I \models_{\text{sm}} \text{clauses } S \longleftrightarrow I \models_{\text{sm}} \text{clauses } T$
using *assms* **by** (*induction rule: dpll-bj-all-induct*) *auto*

Clauses **lemma** *dpll-bj-atms-of-ms-clauses-inv*:

assumes
dpll-bj S T and

$inv\ S$
shows $atms-of-msu\ (clauses\ S) = atms-of-msu\ (clauses\ T)$
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 y_0 : $y = (Propagated\ L\ ()) \# (tl\ y)$
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 $arg-cong[OF\ M', of\ tl]\ y_0\ M\ get-all-marked-decomposition-decomp$ **by** *force*
have $a-Un-N-M$: $unmark\ a \cup set-mset\ ?N \models_{ps}\ unmark\ (tl\ y)$
using $decomp\ ay$ **unfolding** $all-decomposition-implies-def$ **by** (simp add: M) +
moreover have $unmark\ a \cup set-mset\ ?N \models_p\ \{\#L\# \}$ (is $?I \models_p -$)
proof (rule $true-clss-clss-plus-CNot$)
show $?I \models_p\ C + \{\#L\# \}$
using $propa\ propagate_{NOT}.prems$ **by** (auto dest!: $true-clss-clss-in-imp-true-clss-clss$)
next

```

have (λm. {#lit-of m#}) ‘ set ?M' ⊨ps CNot C
  using ⟨trail S ⊨as CNot C⟩ undef by (auto simp add: true-annots-true-clss-clss)
have a1: (λm. {#lit-of m#}) ‘ set a ∪ (λm. {#lit-of m#}) ‘ set (tl y) ⊨ps CNot C
  using propagateNOT.hyps(2) tr-S true-annots-true-clss-clss
  by (force simp add: image-Un sup-commute)
have a2: set-mset (clauses S) ∪ unmark a
  ⊨ps unmark (tl y)
  using calculation by (auto simp add: sup-commute)
show (λm. {#lit-of m#}) ‘ set a ∪ set-mset (clauses S) ⊨ps CNot C
  proof –
    have set-mset (clauses S) ∪ (λm. {#lit-of m#}) ‘ set a ⊨ps
      (λm. {#lit-of m#}) ‘ set a ∪ (λm. {#lit-of m#}) ‘ set (tl y)
      using a2 true-clss-clss-def by blast
    then show (λm. {#lit-of m#}) ‘ set a ∪ set-mset (clauses S) ⊨ps CNot C
      using a1 unfolding sup-commute by (meson true-clss-clss-left-right
        true-clss-clss-union-and true-clss-clss-union-l-r )
  qed
qed

ultimately have unmark a ∪ set-mset ?N ⊨ps unmark ?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 unmark (fst (hd (get-all-marked-decomposition F)))
  ∪ set-mset (clauses S)
  ⊨ps unmark (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 ⊆ atm-of ‘ lits-of F
    using ⟨F ⊨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: unmark a ∪ set-mset (clauses S) ⊨ps unmark b
  using decomp unfolding all-decomposition-implies-def by (auto simp add: F)

have F-D: unmark F ⊨ps CNot D
  using ⟨F ⊨as CNot D⟩ by (simp add: true-annots-true-clss-clss)
then have unmark a ∪ unmark b ⊨ps CNot D
  unfolding ⟨F = b @ a⟩ by (simp add: image-Un sup-commute)
have a-N-CNot-D: unmark a ∪ set-mset (clauses S)

```

```

 $\models_{ps} CNot\ D \cup unmark\ b$ 
apply (rule true-clss-clss-left-right)
using a-N-b F-D unfolding  $\langle F = b @ a \rangle$  by (auto simp add: image-Un ac-simps)

have a-N-D-L:  $unmark\ a \cup set-mset\ (clauses\ S) \models_p D + \{\#L\# \}$ 
by (simp add: N-C)
have  $unmark\ a \cup set-mset\ (clauses\ S) \models_p \{\#L\# \}$ 
using a-N-D-L a-N-CNot-D by (blast intro: true-clss-clss-plus-CNot)
then show ?case
using decomp T tr undef unfolding all-decomposition-implies-def by (auto simp add: F)
qed

```

14.3.3 Termination

Using a proper measure lemma *length-get-all-marked-decomposition-append-Marked*:

```

length (get-all-marked-decomposition (F' @ Marked K () # F)) =
  length (get-all-marked-decomposition F')
+ length (get-all-marked-decomposition (Marked K () # F))
- 1
by (induction F' rule: marked-lit-list-induct) auto

```

lemma *take-length-get-all-marked-decomposition-marked-sandwich*:

```

take (length (get-all-marked-decomposition F))
  (map (f o snd) (rev (get-all-marked-decomposition (F' @ Marked K () # F))))
=
  map (f o snd) (rev (get-all-marked-decomposition F))

```

proof (induction F' rule: marked-lit-list-induct)

```

case nil
then show ?case by auto
next
case (marked K)
then show ?case by (simp add: length-get-all-marked-decomposition-append-Marked)
next
case (proped L m F') note IH = this(1)
obtain a b l where F':  $get-all-marked-decomposition\ (F' @ Marked\ K\ () \# F) = (a, b) \# l$ 
by (cases get-all-marked-decomposition (F' @ Marked K () # F)) auto
have  $length\ (get-all-marked-decomposition\ F) - length\ l = 0$ 
using length-get-all-marked-decomposition-append-Marked[of F' K F]
unfolding F' by (cases get-all-marked-decomposition F') auto
then show ?case
using IH by (simp add: F')
qed

```

lemma *length-get-all-marked-decomposition-length*:

```

length (get-all-marked-decomposition M)  $\leq 1 + length\ M$ 
by (induction M rule: marked-lit-list-induct) auto

```

lemma *length-in-get-all-marked-decomposition-bounded*:

```

assumes  $i:i \in set\ (trail-weight\ S)$ 
shows  $i \leq Suc\ (length\ (trail\ S))$ 
proof -
obtain a b where
   $(a, b) \in set\ (get-all-marked-decomposition\ (trail\ S))$  and
   $ib: i = Suc\ (length\ b)$ 
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 *unassigned-lit* :: '*b literal multiset set* \Rightarrow '*a list* \Rightarrow *nat* where
unassigned-lit $N \ M \equiv \text{card } (\text{atms-of-ms } N) - \text{length } M$

lemma *dpll-bj-trail-mes-increasing-prop*:

fixes $M :: ('v, \text{unit}, \text{unit}) \text{ marked-lits}$ **and** $N :: 'v \text{ clauses}$

assumes

dpll-bj $S \ T$ **and**

inv S **and**

NA : $\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

MA : $\text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d}$: $\text{no-dup } (\text{trail } S)$ **and**

finite: $\text{finite } A$

shows $\mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$

$> \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$

using *assms*(1,2)

proof (*induction rule*: *dpll-bj-all-induct*)

case (*propagate*_{NOT} $C \ L$) **note** $CLN = \text{this}(1)$ **and** $MC = \text{this}(2)$ **and** $\text{undef-L} = \text{this}(3)$ **and** $T = \text{this}(4)$

have *incl*: $\text{atm-of } ' \text{ lits-of } (\text{Propagated } L \ ()) \# \text{trail } S \subseteq \text{atms-of-ms } A$

using *propagate*_{NOT}.*hyps* *propagate-ops.propagate*_{NOT} *dpll-bj-atms-in-trail-in-set* *bj-propagate*_{NOT}

$NA \ MA \ CLN$ **by** (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

have *no-dup*: $\text{no-dup } (\text{Propagated } L \ ()) \# \text{trail } S$

using *defined-lit-map* *n-d* *undef-L* **by** *auto*

obtain $a \ b \ l$ **where** M : $\text{get-all-marked-decomposition } (\text{trail } S) = (a, b) \# l$

by (*cases* *get-all-marked-decomposition* $(\text{trail } S)$) *auto*

have *b-le-M*: $\text{length } b \leq \text{length } (\text{trail } S)$

using *get-all-marked-decomposition-decomp*[*of trail S*] **by** (*simp add: M*)

have *finite* $(\text{atms-of-ms } A)$ **using** *finite* **by** *simp*

then have $\text{length } (\text{Propagated } L \ ()) \# \text{trail } S \leq \text{card } (\text{atms-of-ms } A)$

using *incl* *finite* **unfolding** *no-dup-length-eq-card-atm-of-lits-of*[*OF no-dup*]

by (*simp add: card-mono*)

then have *latm*: $\text{unassigned-lit } A \ b = \text{Suc } (\text{unassigned-lit } A \ (\text{Propagated } L \ d \ \# \ b))$

using *b-le-M* **by** *auto*

then show *?case* **using** $T \ \text{undef-L}$ **by** (*auto simp: latm M* $\mu_C\text{-cons}$)

next

case (*decide*_{NOT} L) **note** $\text{undef-L} = \text{this}(1)$ **and** $MC = \text{this}(2)$ **and** $T = \text{this}(3)$

have *incl*: $\text{atm-of } ' \text{ lits-of } (\text{Marked } L \ ()) \# (\text{trail } S) \subseteq \text{atms-of-ms } A$

using *dpll-bj-atms-in-trail-in-set* *bj-decide*_{NOT} *decide*_{NOT}.*decide*_{NOT}[*OF decide*_{NOT}.*hyps*] $NA \ MA$

MC


```

by auto

have no-dup: no-dup (Marked L () # (trail S))
  using defined-lit-map n-d undef-L by auto
obtain a b l where M: get-all-marked-decomposition (trail S) = (a, b) # l
  by (cases get-all-marked-decomposition (trail S)) auto

then have length (Marked L () # (trail S)) ≤ 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 μC-cons)
next
case (backjump C F' K F L C' T) note undef-L = this(4) and MC = this(1) and tr-S = this(3)
and
  L = this(5) and T = this(8)
have incl: atm-of ' lits-of (Propagated L () # F) ⊆ atms-of-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 ≤ 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) ≤ 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) ≤ (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 (λa. Suc (length (snd a))) (rev (get-all-marked-decomposition (F' @ Marked K () # F)))
  = map (λa. Suc (length (snd a))) (rev (get-all-marked-decomposition F)) @ rem
  using take-length-get-all-marked-decomposition-marked-sandwich[of F λa. Suc (length a) F' K]
  unfolding o-def by (metis append-take-drop-id)
then have rem: map (λa. Suc (length (snd a)))
  (get-all-marked-decomposition (F' @ Marked K () # F))
  = rev rem @ map (λa. Suc (length (snd a))) ((get-all-marked-decomposition F))
  by (simp add: rev-map[symmetric] rev-swap)
have length (rev rem @ map (λa. Suc (length (snd a))) (get-all-marked-decomposition F))
  ≤ Suc (card (atms-of-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  $i < \text{length } xs \implies \text{length } xs - \text{Suc } i < \text{length } xs$ 
    by auto
  then have  $H: i < \text{length } xs \implies \text{rev } xs ! i \in \text{set } xs$ 
    using  $\text{rev-nth}[of\ i\ xs]$  unfolding  $\text{in-set-conv-nth}$  by  $(\text{force simp add: in-set-conv-nth})$ 
} note  $H = \text{this}$ 
have  $\forall i < \text{length } \text{rem}. \text{rev } \text{rem} ! i < \text{card } (\text{atms-of-ms } A) + 2$ 
  using  $\text{tr-S-le-A length-in-get-all-marked-decomposition-bounded}[of\ -\ S]$  unfolding  $\text{tr-S}$ 
  by  $(\text{force simp add: o-def rem dest!: } H \text{ intro: length-get-all-marked-decomposition-length})$ 
ultimately show  $?case$ 
  using  $\mu_C\text{-bounded}[of\ \text{rev } \text{rem } \text{card } (\text{atms-of-ms } A) + 2\ \text{unassigned-lit } A\ l]\ T\ \text{undef-L}$ 
  by  $(\text{simp add: rem } \mu_C\text{-append } \mu_C\text{-cons } F\ \text{tr-S})$ 
qed

```

lemma $\text{dpll-bj-trail-mes-decreasing-prop}$:

```

assumes  $\text{dpll: dpll-bj } S\ T$  and  $\text{inv: inv } S$  and
 $N\text{-A: atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$  and
 $M\text{-A: atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$  and
 $\text{nd: no-dup } (\text{trail } S)$  and
 $\text{fin-A: finite } A$ 
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  $?b = 2 + \text{card } (\text{atms-of-ms } A)$ 
let  $?s = 1 + \text{card } (\text{atms-of-ms } A)$ 
let  $?μ = \mu_C\ ?s\ ?b$ 
have  $M'\text{-A: atm-of ' lits-of } (\text{trail } T) \subseteq \text{atms-of-ms } A$ 
  by  $(\text{meson } M\text{-A } N\text{-A } \text{dpll } \text{dpll-bj-atms-in-trail-in-set } \text{inv})$ 
have  $\text{nd': no-dup } (\text{trail } T)$ 
  using  $\langle \text{dpll-bj } S\ T \rangle\ \text{dpll-bj-no-dup } \text{nd } \text{inv}$  by  $\text{blast}$ 
{ fix } i :: nat and xs :: 'a list
  have  $i < \text{length } xs \implies \text{length } xs - \text{Suc } i < \text{length } xs$ 
    by  $\text{auto}$ 
  then have  $H: i < \text{length } xs \implies xs ! i \in \text{set } xs$ 
    using  $\text{rev-nth}[of\ i\ xs]$  unfolding  $\text{in-set-conv-nth}$  by  $(\text{force simp add: in-set-conv-nth})$ 
} note } H = \text{this}

```

```

have  $l\text{-M-A: length } (\text{trail } S) \leq \text{card } (\text{atms-of-ms } A)$ 
  by  $(\text{simp add: fin-A } M\text{-A } \text{card-mono no-dup-length-eq-card-atm-of-lits-of } \text{nd})$ 
have  $l\text{-M'-A: length } (\text{trail } T) \leq \text{card } (\text{atms-of-ms } A)$ 
  by  $(\text{simp add: fin-A } M'\text{-A } \text{card-mono no-dup-length-eq-card-atm-of-lits-of } \text{nd'})$ 
have  $l\text{-trail-weight-M: length } (\text{trail-weight } T) \leq 1 + \text{card } (\text{atms-of-ms } A)$ 
  using  $l\text{-M'-A length-get-all-marked-decomposition-length}[of\ \text{trail } T]$  by  $\text{auto}$ 
have  $\text{bounded-M: } \forall i < \text{length } (\text{trail-weight } T). (\text{trail-weight } T) ! i < \text{card } (\text{atms-of-ms } A) + 2$ 
  using  $\text{length-in-get-all-marked-decomposition-bounded}[of\ -\ T]\ l\text{-M'-A}$ 
  by  $(\text{metis } (\text{no-types, lifting})\ \text{Nat.le-trans One-nat-def Suc-1 add.right-neutral add-Suc-right le-imp-less-Suc less-eq-Suc-le nth-mem})$ 

```

```

from  $\text{dpll-bj-trail-mes-increasing-prop}[OF\ \text{dpll } \text{inv } N\text{-A } M\text{-A } \text{nd } \text{fin-A}]$ 
have  $\mu_C\ ?s\ ?b (\text{trail-weight } S) < \mu_C\ ?s\ ?b (\text{trail-weight } T)$  by  $\text{simp}$ 
moreover from  $\mu_C\text{-bounded}[OF\ \text{bounded-M } l\text{-trail-weight-M}]$ 
  have  $\mu_C\ ?s\ ?b (\text{trail-weight } T) \leq ?b \wedge ?s$  by  $\text{auto}$ 
ultimately show  $?thesis$  by  $\text{linarith}$ 

```

qed

lemma *wf-dpll-bj*:

assumes *fin*: *finite A*

shows *wf* $\{(T, S). \text{dpll-bj } S \text{ } T$
 $\wedge \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$
(is wf ?A)

proof (*rule wf-bounded-measure*[*of -*
 $\lambda -. (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\lambda S. \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)]$)

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* = $\mu_C \text{ ?s ?b}$

assume *ab*: $(b, a) \in \{(T, S). \text{dpll-bj } S \text{ } T$
 $\wedge \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$

have *fin-A*: *finite (atms-of-ms A)*

using *fin* **by** *auto*

have

dpll-bj: *dpll-bj a b and*

N-A: *atms-of-msu (clauses a) ⊆ atms-of-ms A and*

M-A: *atm-of ' lits-of (trail a) ⊆ atms-of-ms A and*

nd: *no-dup (trail a) and*

inv: *inv a*

using *ab* **by** *auto*

have *M'-A*: *atm-of ' lits-of (trail b) ⊆ atms-of-ms A*

by (*meson M-A N-A (dpll-bj a b) dpll-bj-atms-in-trail-in-set inv*)

have *nd'*: *no-dup (trail b)*

using *(dpll-bj a b) dpll-bj-no-dup nd inv* **by** *blast*

{ fix *i* :: *nat* **and** *xs* :: '*a list*

have $i < \text{length } xs \implies \text{length } xs - \text{Suc } i < \text{length } xs$

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 $\mu_C \text{ ?s ?b } (\text{trail-weight } a) < \mu_C \text{ ?s ?b } (\text{trail-weight } b)$ **by** *simp*

moreover from $\mu_C\text{-bounded}$ [*OF bounded-M l-trail-weight-M*]

have $\mu_C \text{ ?s ?b } (\text{trail-weight } b) \leq ?b \wedge ?s$ **by** *auto*

ultimately show $?b \wedge ?s \leq ?b \wedge ?s \wedge$

$\mu_C \text{ ?s ?b (trail-weight } b) \leq \text{?b} \wedge \text{?s} \wedge$
 $\mu_C \text{ ?s ?b (trail-weight } a) < \mu_C \text{ ?s ?b (trail-weight } b)$
 by blast
 qed

14.3.4 Normal Forms

We prove that given a normal form of DPLL, with some invariants, the either N is satisfiable and the built valuation M is a model; or N is unsatisfiable.

Idea of the proof: We have to prove that *satisfiable* N , $\neg M \models_{as} N$ and there is no remaining step is incompatible.

1. The *decide* rules tells us that every variable in N has a value.
2. $\neg M \models_{as} N$ tells us that there is conflict.
3. There is at least one decision in the trail (otherwise, M is a model of N).
4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

The assumption are saying that we have a finite upper bound A for the literals, that we cannot do any step *no-step dpll-bj* S

theorem *dpll-backjump-final-state*:

fixes $A :: \text{'v literal multiset set}$ **and** $S \ T :: \text{'st}$

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 $\text{?N} = \text{set-mset}$ (*clauses* S)

let $\text{?M} = \text{trail}$ S

consider

(*sat*) *satisfiable* ?N **and** $\text{?M} \models_{as} \text{?N}$

| (*sat'*) *satisfiable* ?N **and** $\neg \text{?M} \models_{as} \text{?N}$

| (*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** $C \in \text{?N}$ **and** $\neg \text{?M} \models_a C$ **using** M **unfolding** *true-annots-def* **by** *auto*

obtain $I :: \text{'v literal set}$ **where**

$I \models_s \text{?N}$ **and**

cons: *consistent-interp* I **and**

tot: *total-over-m* I ?N **and**

atm-I-N: *atm-of* ' $I \subseteq$ *atms-of-ms* ?N

using *sat* **unfolding** *satisfiable-def-min* **by** *auto*

let $\text{?I} = I \cup \{P \mid P \in \text{lits-of } \text{?M} \wedge \text{atm-of } P \notin \text{atm-of ' } I\}$

```

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 ∪ unmark ?M)
  using tot atms-of-s-def unfolding total-over-m-def total-over-set-def
  by fastforce
have {P | P. P ∈ lits-of ?M ∧ atm-of P ∉ atm-of 'I} ⊨s ?O
  using ⟨I ⊨s ?N⟩ atm-I-N by (auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def)
then have I'-N: ?I ⊨s ?N ∪ ?O
  using ⟨I ⊨s ?N⟩ true-clss-union-increase by force
have tot': total-over-m ?I (?N ∪ ?O)
  using atm-I-N tot unfolding total-over-m-def total-over-set-def
  by (force simp: image-iff lits-of-def dest!: is-marked-ex-Marked)

have atms-N-M: atms-of-ms ?N ⊆ atm-of 'lits-of ?M
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain l :: 'v where
    l-N: l ∈ atms-of-ms ?N and
    l-M: l ∉ atm-of 'lits-of ?M
  by auto
  have undefined-lit ?M (Pos l)
    using l-M by (metis Marked-Propagated-in-iff-in-lits-of
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
  from bj-decideNOT[OF decideNOT[OF this]] show False
    using l-N n-s by (metis literal.sel(1) state-eqNOT-ref)
qed

have ?M ⊨as CNot C
  by (metis ⟨C ∈ set-mset (clauses S)⟩ ⟨¬ trail S ⊨a C⟩ all-variables-defined-not-imply-cnot
    atms-N-M atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of atms-of-ms-CNot-atms-of-ms
    subset-eq)
have ∃ l ∈ set ?M. is-marked l
proof (rule ccontr)
  let ?O = { {#lit-of L#} | L. is-marked L ∧ L ∈ set ?M ∧ atm-of (lit-of L) ∉ atms-of-ms ?N }
  have ∅[iff]: ∧ I. total-over-m I (?N ∪ ?O ∪ unmark ?M)
    ↔ total-over-m I (?N ∪ unmark ?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 unmark ?M
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] by auto

  then have ?I ⊨s unmark ?M
    using cons-I' I'-N tot-I' ⟨?I ⊨s ?N ∪ ?O⟩ unfolding ∅ true-clss-clss-def by blast
  then have lits-of ?M ⊆ ?I
    unfolding true-clss-def lits-of-def by auto
  then have ?M ⊨as ?N
    using I'-N ⟨C ∈ ?N⟩ ⟨¬ ?M ⊨a C⟩ cons-I' atms-N-M
    by (meson ⟨trail S ⊨as CNot C⟩ consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
      true-annots-def true-clss-mono-set-mset-l true-clss-def)

```

```

    then show False using M by fast
  qed
from List.split-list-first-propE[OF this] obtain K :: 'v literal and
  F F' :: ('v, unit, unit) marked-lit list where
  M-K: ?M = F' @ Marked K () # F and
  nm:  $\forall f \in \text{set } F'. \neg \text{is-marked } f$ 
  unfolding is-marked-def by (metis (full-types) old.unit.exhaust)
let ?K = Marked K ()::('v, unit, unit) marked-lit
have ?K ∈ set ?M
  unfolding M-K by auto
let ?C = image-mset lit-of {#L ∈ #mset ?M. is-marked L ∧ L ≠ ?K #} :: 'v literal multiset
let ?C' = set-mset (image-mset (λL::'v literal. {#L #}) (?C + {#lit-of ?K #}))
have ?N ∪ {#{#lit-of L #} | L. is-marked L ∧ L ∈ set ?M} ⊢ps unmark ?M
  using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have C': ?C' = {#{#lit-of L #} | L. is-marked L ∧ L ∈ set ?M}
  unfolding M-K apply standard
  apply force
  using IntI by auto
ultimately have N-C-M: ?N ∪ ?C' ⊢ps unmark ?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-annots-def Ball-def
  true-annot-def by (metis consistent-CNot-not sup.orderE sup-commute true-clss-def
    true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have undefined-lit F K using ⟨no-dup ?M⟩ unfolding M-K by (simp add: defined-lit-map)
moreover
  have ?N ∪ ?C' ⊢ps {#{#}}
  proof -
    have A: ?N ∪ ?C' ∪ unmark ?M =
      ?N ∪ unmark ?M
    unfolding M-K by auto
    show ?thesis
      using true-clss-clss-left-right[OF N-C-M, of {#{#}}] N-M-False unfolding A by auto
  qed
have ?N ⊢p image-mset uminus ?C + {#-K #}
  unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
  proof (intro allI impI)
    fix I
    assume
      tot: total-over-set I (atms-of-ms (?N ∪ {image-mset uminus ?C + {#-K #}))) and
      cons: consistent-interp I and
      I ⊢s ?N
    have (K ∈ I ∧ -K ∉ I) ∨ (-K ∈ I ∧ K ∉ I)
      using cons tot unfolding consistent-interp-def by (cases K) auto
    have tot': total-over-set I
      (atm-of 'lit-of ' (set ?M ∩ {L. is-marked L ∧ L ≠ Marked K ()}))
    using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
    { fix x :: ('v, unit, unit) marked-lit
      assume
        a3: lit-of x ∉ I and
        a1: x ∈ set ?M and
        a4: is-marked x and
        a5: x ≠ Marked K ()
      then have Pos (atm-of (lit-of x)) ∈ I ∨ Neg (atm-of (lit-of x)) ∈ I

```

```

    using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast
  moreover have f6: Neg (atm-of (lit-of x)) = - Pos (atm-of (lit-of x))
    by simp
  ultimately have - lit-of x ∈ I
    using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      literal.sel(1))
} note H = this

have ¬I ⊨s ?C'
  using ⟨?N ∪ ?C' ⊨ps {{#}}⟩ tot cons ⟨I ⊨s ?N⟩
  unfolding true-clss-clss-def total-over-m-def
  by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of)
then show I ⊨ image-mset uminus ?C + {#- K#}
  unfolding true-clss-def true-cl-def Bex-mset-def
  using ⟨(K ∈ I ∧ -K ∉ I) ∨ (-K ∈ I ∧ K ∉ I)⟩
  by (auto dest!: H)
qed
moreover have F ⊨as CNot (image-mset uminus ?C)
  using nm unfolding true-annots-def CNot-def M-K by (auto simp add: lits-of-def)
ultimately have False
  using bj-can-jump[of S F' K F C -K
    image-mset uminus (image-mset lit-of {# L :# mset ?M. is-marked L ∧ L ≠ Marked K ()#})]
    ⟨C ∈ ?N⟩ n-s ⟨?M ⊨as CNot C⟩ bj-backjump inv ⟨no-dup (trail S)⟩ unfolding M-K by auto
  then show ?thesis by fast
qed auto
qed

end

locale dp11-with-backjumping =
  dp11-with-backjumping-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv backjump-conds
for
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and tl-trail :: 'st ⇒ 'st and
  add-clNOT remove-clNOT :: '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
+
assumes dp11-bj-inv: ∧ S T. dp11-bj S T ⇒ inv S ⇒ inv T
begin

lemma rtranclp-dp11-bj-inv:
assumes dp11-bj* S T and inv S
shows inv T
using assms by (induction rule: rtranclp-induct)
  (auto simp add: dp11-bj-no-dup intro: dp11-bj-inv)

lemma rtranclp-dp11-bj-no-dup:
assumes dp11-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: rtrancpl-dpll-bj-inv dpll-bj-inv)

lemma rtrancpl-dpll-bj-atms-of-ms-clauses-inv:

assumes

dpll-bj** S T **and** inv S

shows atms-of-msu (clauses S) = atms-of-msu (clauses T)

using assms **by** (induction rule: rtrancpl-induct)

(auto dest: rtrancpl-dpll-bj-inv dpll-bj-atms-of-ms-clauses-inv)

lemma rtrancpl-dpll-bj-atms-in-trail:

assumes

dpll-bj** S T **and**

inv S **and**

atm-of ' (lits-of (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: rtrancpl-induct)

using dpll-bj-atms-in-trail dpll-bj-atms-of-ms-clauses-inv rtrancpl-dpll-bj-inv **by** auto

lemma rtrancpl-dpll-bj-sat-iff:

assumes dpll-bj** S T **and** inv S

shows $I \models_{sm} \text{clauses } S \longleftrightarrow I \models_{sm} \text{clauses } T$

using assms **by** (induction rule: rtrancpl-induct)

(auto dest!: dpll-bj-sat-iff simp: rtrancpl-dpll-bj-inv)

lemma rtrancpl-dpll-bj-atms-in-trail-in-set:

assumes

dpll-bj** S T **and**

inv S

atms-of-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: rtrancpl-induct)

(auto dest: rtrancpl-dpll-bj-inv

simp add: dpll-bj-atms-in-trail-in-set rtrancpl-dpll-bj-atms-of-ms-clauses-inv

rtrancpl-dpll-bj-inv)

lemma rtrancpl-dpll-bj-all-decomposition-implies-inv:

assumes

dpll-bj** S T **and**

inv S

all-decomposition-implies-m (clauses 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: rtrancpl-induct)

(auto intro: dpll-bj-all-decomposition-implies-inv simp: rtrancpl-dpll-bj-inv)

lemma rtrancpl-dpll-bj-inv-incl-dpll-bj-inv-trancpl:

$\{(T, S). \text{dpll-bj}^{++} S T$

\wedge atms-of-msu (clauses S) \subseteq atms-of-ms $A \wedge$ atm-of ' lits-of (trail S) \subseteq atms-of-ms A

\wedge no-dup (trail S) \wedge inv $S\}$

$\subseteq \{(T, S). \text{dpll-bj } S T \wedge$ atms-of-msu (clauses S) \subseteq atms-of-ms A

\wedge atm-of ' lits-of (trail S) \subseteq atms-of-ms $A \wedge$ no-dup (trail S) \wedge inv $S\}^+$

(is $?A \subseteq ?B^+$)

proof standard

fix x

assume $x-A: x \in ?A$
obtain $S T :: 'st$ **where**
 $x[simp]: x = (T, S)$ **by** $(cases\ x)\ auto$
have
 $dpll-bj^{++}\ S\ T$ **and**
 $atms-of-msu\ (clauses\ S) \subseteq atms-of-ms\ A$ **and**
 $atm-of\ ' lits-of\ (trail\ S) \subseteq atms-of-ms\ A$ **and**
 $no-dup\ (trail\ S)$ **and**
 $inv\ S$
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 $(step\ T\ U)$ **note** $step = this(1)$ **and** $ST = this(2)$ **and** $IH = this(3)[OF\ this(4-7)]$
and $N-A = this(4)$ **and** $M-A = this(5)$ **and** $nd = this(6)$ **and** $inv = this(7)$

have $[simp]: atms-of-msu\ (clauses\ S) = atms-of-msu\ (clauses\ T)$
using $step\ rtranclp-dpll-bj-atms-of-ms-clauses-inv\ tranclp-into-rtranclp\ inv$ **by** $fastforce$
have $no-dup\ (trail\ T)$
using $local.step\ nd\ rtranclp-dpll-bj-no-dup\ tranclp-into-rtranclp\ inv$ **by** $fastforce$
moreover have $atm-of\ ' (lits-of\ (trail\ T)) \subseteq atms-of-ms\ A$
by $(metis\ inv\ M-A\ N-A\ local.step\ rtranclp-dpll-bj-atms-in-trail-in-set\ tranclp-into-rtranclp)$
moreover have $inv\ T$
using $inv\ local.step\ rtranclp-dpll-bj-inv\ tranclp-into-rtranclp$ **by** $fastforce$
ultimately have $(U, T) \in ?B$ **using** $ST\ N-A\ M-A\ inv$ **by** $auto$
then show $?case$ **using** IH **by** $(rule\ trancl-into-trancl2)$
qed
qed

lemma $wf-tranclp-dpll-bj$:

assumes $fin: finite\ A$
shows $wf\ \{(T, S). dpll-bj^{++}\ S\ T$
 $\wedge atms-of-msu\ (clauses\ S) \subseteq atms-of-ms\ A \wedge atm-of\ ' lits-of\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S) \wedge inv\ S\}$
using $wf-trancl[OF\ wf-dpll-bj[OF\ fin]]\ rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl$
by $(rule\ wf-subset)$

lemma $dpll-bj-sat-ext-iff$:

$dpll-bj\ S\ T \implies inv\ S \implies I \models_{sextm}\ clauses\ S \longleftrightarrow I \models_{sextm}\ clauses\ T$
by $(simp\ add: dpll-bj-clauses)$

lemma $rtranclp-dpll-bj-sat-ext-iff$:

$dpll-bj^{**}\ S\ T \implies inv\ S \implies I \models_{sextm}\ clauses\ S \longleftrightarrow I \models_{sextm}\ clauses\ T$
by $(induction\ rule:\ rtranclp-induct)\ (simp-all\ add: rtranclp-dpll-bj-inv\ dpll-bj-sat-ext-iff)$

theorem $full-dpll-backjump-final-state$:

fixes $A :: 'v\ literal\ multiset\ set$ **and** $S\ T :: 'st$
assumes
 $full: full\ dpll-bj\ S\ T$ **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 *⟨inv S⟩ 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 *⟨finite A⟩ dpll-backjump-final-state by force*
then show *?thesis*
by (*meson ⟨inv S⟩ rtranclp-dpll-bj-sat-iff satisfiable-carac st true-annots-true-cls*)
qed

corollary *full-dpll-backjump-final-state-from-init-state:*

fixes *A :: ‘v literal multiset set and S T :: ‘st*

assumes

full: full dpll-bj S T and

trail S = [] and

clauses S = N and

inv S

shows *unsatisfiable (set-mset N) \vee (trail T \models_{asm} N \wedge satisfiable (set-mset N))*

using *assms full-dpll-backjump-final-state[of S T set-mset N] by auto*

lemma *tranclp-dpll-bj-trail-mes-decreasing-prop:*

assumes *dpll: dpll-bj⁺⁺ S T and inv: inv S and*

N-A: atms-of-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 *rtranclp-dpll-bj-atms-of-ms-clauses-inv by (metis dpll-bj-clauses dpll-bj-inv inv st*

```

    trancplD)
  then have  $N-A'$ :  $\text{atms-of-msu } (\text{clauses } T) \subseteq \text{atms-of-ms } A$ 
    using  $N-A$  by auto
  moreover have  $M-A'$ :  $\text{atm-of } ' \text{ lits-of } (\text{trail } T) \subseteq \text{atms-of-ms } A$ 
    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-clNOT remove-clNOT
  for
    trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
    clauses :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and tl-trail :: 'st  $\Rightarrow$  'st and
    add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
  fixes
    learn-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool

begin
  inductive learn :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
    clauses  $S \models_{pm} C \implies \text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$ 
       $\implies \text{learn-cond } C$ 
       $\implies T \sim \text{add-cl}_{NOT} C S$ 
       $\implies \text{learn } S T$ 
  inductive-cases learnNOT $E$ : learn  $S T$ 

lemma learn- $\mu_C$ -stable:
  assumes learn  $S T$  and no-dup (trail  $S$ )
  shows  $\mu_C A B (\text{trail-weight } S) = \mu_C A B (\text{trail-weight } T)$ 
  using assms by (auto elim: learnNOT $E$ )
end

```

```

locale forget-ops =
  dpll-state trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  for
    trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
    clauses :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and tl-trail :: 'st  $\Rightarrow$  'st and
    add-clNOT remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
  fixes
    forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool

begin
  inductive forgetNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
    forgetNOT: clauses  $S - \text{replicate-mset } (\text{count } (\text{clauses } S) C) C \models_{pm} C$ 

```

```

 $\Rightarrow$  forget-cond  $C\ S$ 
 $\Rightarrow C \in \# \text{ clauses } S$ 
 $\Rightarrow T \sim \text{remove-cl}_{NOT} C\ S$ 
 $\Rightarrow \text{forget}_{NOT} S\ T$ 
inductive-cases forgetNOT $E$ : forgetNOT  $S\ T$ 

lemma forget- $\mu_C$ -stable:
  assumes forgetNOT  $S\ T$ 
  shows  $\mu_C\ A\ B\ (\text{trail-weight } S) = \mu_C\ A\ B\ (\text{trail-weight } T)$ 
  using assms by (auto elim!: forgetNOT $E$ )
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 cdclNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for  $S :: 'st$  where
  c-dpll-bj: dpll-bj  $S\ S' \Rightarrow$  cdclNOT  $S\ S'$  |
  c-learn: learn  $S\ S' \Rightarrow$  cdclNOT  $S\ S'$  |
  c-forgetNOT: forgetNOT  $S\ S' \Rightarrow$  cdclNOT  $S\ S'$ 

```

```

lemma cdclNOT-all-induct[consumes 1, case-names dpll-bj learn forgetNOT]:
  fixes  $S\ T :: 'st$ 
  assumes cdclNOT  $S\ T$  and

```

dpll: $\bigwedge T. \text{dpll-bj } S \ T \implies P \ S \ T$ **and**
learning:
 $\bigwedge C \ T. \text{clauses } S \models_{pm} C \implies$
 $\text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S)) \implies$
 $T \sim \text{add-cl}_\text{NOT} \ C \ S \implies$
 $P \ S \ T$ **and**
forgetting: $\bigwedge C \ T. \text{clauses } S - \text{replicate-mset } (\text{count } (\text{clauses } S) \ C) \ C \models_{pm} C \implies$
 $C \in \# \text{clauses } S \implies$
 $T \sim \text{remove-cl}_\text{NOT} \ C \ S \implies$
 $P \ S \ T$
shows $P \ S \ T$
using *assms*(1) **by** (*induction rule*: *cdcl_{NOT}.induct*)
(auto intro: assms(2, 3, 4) *elim!:* *learn_{NOT}E* *forget_{NOT}E*)**+**

lemma *cdcl_{NOT}-no-dup*:

assumes
 $\text{cdcl}_\text{NOT} \ S \ T$ **and**
 $\text{inv } S$ **and**
 $\text{no-dup } (\text{trail } S)$
shows $\text{no-dup } (\text{trail } T)$
using *assms* **by** (*induction rule*: *cdcl_{NOT}-all-induct*) (*auto intro: dpll-bj-no-dup*)

Consistency of the trail lemma *cdcl_{NOT}-consistent*:

assumes
 $\text{cdcl}_\text{NOT} \ S \ T$ **and**
 $\text{inv } S$ **and**
 $\text{no-dup } (\text{trail } S)$
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 $\text{cdcl}_\text{NOT} \ S \ T$ **and** $\text{inv } S$ **and** $\text{no-dup } (\text{trail } S)$
shows $\text{atms-of-msu } (\text{clauses } T) \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$
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 $\text{cdcl}_\text{NOT} \ S \ T$ **and** $\text{inv } S$ **and** $\text{no-dup } (\text{trail } S)$
and $\text{atm-of } ' (\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-msu } (\text{clauses } S)$
shows $\text{atm-of } ' (\text{lits-of } (\text{trail } T)) \subseteq \text{atms-of-msu } (\text{clauses } S)$
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
 $\text{cdcl}_\text{NOT} \ S \ T$ **and** $\text{inv } S$ **and** $\text{no-dup } (\text{trail } S)$ **and**
 $\text{atms-of-msu } (\text{clauses } S) \subseteq A$ **and**
 $\text{atm-of } ' (\text{lits-of } (\text{trail } S)) \subseteq A$
shows $\text{atm-of } ' (\text{lits-of } (\text{trail } T)) \subseteq A$
using *assms*
by (*induction rule*: *cdcl_{NOT}-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 unmark a ∪ set-mset (clauses S) ⊨ps unmark 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 unmark a ∪ set-mset (clauses T)
    ⊨ps unmark 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 ⊨sextm clauses S and
      I ⊆ J and
      tot: total-over-m J (set-mset ({#C#} + (clauses S))) and
      cons: consistent-interp J
    then have J ⊨sm clauses S unfolding true-clss-ext-def by auto

    moreover

```

```

    with (clauses  $S \models_{pm} C$ ) have  $J \models C$ 
      using tot cons unfolding true-clss-cl-def by auto
    ultimately have  $J \models_{sm} \{\#C\# \} + \text{clauses } S$  by auto
  }
then have  $H: I \models_{sextm} (\text{clauses } S) \implies I \models_{sext} \text{insert } C (\text{set-mset } (\text{clauses } S))$ 
  unfolding true-clss-ext-def by auto
show ?case
  apply standard
    using T n-d apply (auto simp add: H)[]
  using T n-d apply simp
  by (metis Diff-insert-absorb insert-subset subsetI subset-antisym
    true-clss-ext-decrease-right-remove-r)
next
case (forgetNOT C T) note cls-C = this(1) and T = this(3)
{ fix J
  assume
     $I \models_{sext} \text{set-mset } (\text{clauses } S) - \{C\}$  and
     $I \subseteq J$  and
    tot: total-over-m J (set-mset (clauses S)) and
    cons: consistent-interp J
  then have  $J \models_s \text{set-mset } (\text{clauses } S) - \{C\}$ 
    unfolding true-clss-ext-def by (meson Diff-subset total-over-m-subset)

  moreover
    with cls-C have  $J \models C$ 
      using tot cons unfolding true-clss-cl-def
      by (metis Un-commute forgetNOT.hypos(2) insert-Diff insert-is-Un mem-set-mset-iff order-refl
        set-mset-minus-replicate-mset(1))
    ultimately have  $J \models_{sm} (\text{clauses } S)$  by (metis insert-Diff-single true-clss-insert)
  }
then have  $H: I \models_{sext} \text{set-mset } (\text{clauses } S) - \{C\} \implies I \models_{sextm} (\text{clauses } S)$ 
  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 cdclNOT-inv:  $\bigwedge S T. \text{cdcl}_{NOT} S T \implies \text{inv } S \implies \text{inv } T$ 
begin
sublocale dpll-with-backjumping
  apply unfold-locales
  using cdclNOT.simps cdclNOT-inv by auto

lemma rtranclp-cdclNOT-inv:
  cdclNOT** S T  $\implies \text{inv } S \implies \text{inv } T$ 
  by (induction rule: rtranclp-induct) (auto simp add: cdclNOT-inv)

lemma rtranclp-cdclNOT-no-dup:
  assumes cdclNOT** S T and inv S
  and no-dup (trail S)
  shows no-dup (trail T)
  using assms by (induction rule: rtranclp-induct) (auto intro: cdclNOT-no-dup rtranclp-cdclNOT-inv)

```

lemma *rtrancpl-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) ⊆ A* **and**

atms-trail-S: *atm-of (lits-of (trail S)) ⊆ A*

shows *atm-of (lits-of (trail T)) ⊆ A ∧ atms-of-msu (clauses T) ⊆ A*

using *cdcl*

proof (*induction rule: rtrancpl-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 rtrancpl-cdcl_{NOT}-inv* **by** *blast*

have *no-dup (trail T)*

using *rtrancpl-cdcl_{NOT}-no-dup[of S T]* *st cdcl_{NOT} inv n-d* **by** *blast*

then have *atms-of-msu (clauses U) ⊆ A*

using *cdcl_{NOT}-atms-of-ms-clauses-decreasing[OF cdcl_{NOT}] IH n-d (inv T)* **by** *auto*

moreover

have *atm-of (lits-of (trail U)) ⊆ A*

using *cdcl_{NOT}-atms-in-trail-in-set[OF cdcl_{NOT}, of A] (no-dup (trail T))*

by (*meson atms-trail-S atms-clauses-S IH (inv T) cdcl_{NOT}*)

ultimately show ?*case* **by** *fast*

qed

lemma *rtrancpl-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: rtrancpl-cdcl_{NOT}-inv cdcl_{NOT}-all-decomposition-implies rtrancpl-cdcl_{NOT}-no-dup*)

lemma *rtrancpl-cdcl_{NOT}-bj-sat-ext-iff*:

assumes *cdcl_{NOT}** S T* **and** *inv S* **and** *no-dup (trail S)*

shows $I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$

using *assms* **apply** (*induction rule: rtrancpl-induct*)

using *cdcl_{NOT}-bj-sat-ext-iff* **by** (*auto intro: rtrancpl-cdcl_{NOT}-inv rtrancpl-cdcl_{NOT}-no-dup*)

definition *cdcl_{NOT}-NOT-all-inv* **where**

cdcl_{NOT}-NOT-all-inv A S \longleftrightarrow (*finite A ∧ inv S ∧ atms-of-msu (clauses S) ⊆ atms-of-ms A*

$\wedge \text{atm-of (lits-of (trail S))} \subseteq \text{atms-of-ms A} \wedge \text{no-dup (trail S)}$)

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: rtrancpl-cdcl_{NOT}-inv rtrancpl-cdcl_{NOT}-no-dup rtrancpl-cdcl_{NOT}-trail-clauses-bound*)

abbreviation *learn-or-forget* **where**

learn-or-forget S T $\equiv (\lambda S T. \text{learn } S T \vee \text{forget}_{\text{NOT}} S T) S T$

lemma *rtrancpl-learn-or-forget-cdcl_{NOT}*:
*learn-or-forget** S T \implies cdcl_{NOT}** S T*
using *rtrancpl-mono*[of *learn-or-forget cdcl_{NOT}*] *cdcl_{NOT}.c-learn cdcl_{NOT}.c-forget_{NOT}* **by** *blast*

lemma *learn-or-forget-dpll- μ_C* :

assumes

l-f: *learn-or-forget** S T* **and**

dpll: *dpll-bj T U* **and**

inv: *cdcl_{NOT}-NOT-all-inv A S*

shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } U)$
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$
(is $?_{\mu} U < ?_{\mu} S$ **)**

proof $-$

have $?_{\mu} S = ?_{\mu} T$

using *l-f*

proof (*induction*)

case *base*

then show *?case* **by** *simp*

next

case (*step T U*)

moreover then have *no-dup (trail T)*

using *rtrancpl-cdcl_{NOT}-no-dup*[of *S T*] *cdcl_{NOT}-NOT-all-inv-def inv*

rtrancpl-learn-or-forget-cdcl_{NOT} **by** *auto*

ultimately show *?case*

using *forget- μ_C -stable learn- μ_C -stable inv* **unfolding** *cdcl_{NOT}-NOT-all-inv-def* **by** *presburger*

qed

moreover have *cdcl_{NOT}-NOT-all-inv A T*

using *rtrancpl-learn-or-forget-cdcl_{NOT}* *cdcl_{NOT}-NOT-all-inv l-f inv* **by** *blast*

ultimately show *?thesis*

using *dpll-bj-trail-mes-decreasing-prop*[of *T U A, OF dpll*] *finite*

unfolding *cdcl_{NOT}-NOT-all-inv-def* **by** *linarith*

qed

lemma *infinite-cdcl_{NOT}-exists-learn-and-forget-infinite-chain*:

assumes

$\bigwedge i. \text{cdcl}_{NOT} (f i) (f (\text{Suc } i))$ **and**

inv: *cdcl_{NOT}-NOT-all-inv A (f 0)*

shows $\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (\text{Suc } i))$

using *assms*

proof (*induction* $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (f 0))$
arbitrary: *f*
rule: *nat-less-induct-case*)

case (*Suc n*) **note** *IH = this(1)* **and** $\mu = \text{this}(2)$ **and** *cdcl_{NOT} = this(3)* **and** *inv = this(4)*

consider

(*dpll-end*) $\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (\text{Suc } i))$

| (*dpll-more*) $\neg(\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (\text{Suc } i)))$

by *blast*

then show *?case*

proof *cases*

case *dpll-end*

then show *?thesis* **by** *auto*

next

```

case dpll-more
then have  $j: \exists i. \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ 
  by blast
obtain i where
   $\neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$  and
   $\forall k < i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
  proof –
    obtain  $i_0$  where  $\neg \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f \ i_0) \ (f \ (\text{Suc } i_0))$ 
      using j by auto
    then have  $\{i. i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))\} \neq \{\}$ 
      by auto
    let  $?I = \{i. i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))\}$ 
    let  $?i = \text{Min } ?I$ 
    have finite  $?I$ 
      by auto
    have  $\neg \text{learn } (f \ ?i) \ (f \ (\text{Suc } ?i)) \wedge \neg \text{forget}_{NOT} (f \ ?i) \ (f \ (\text{Suc } ?i))$ 
      using Min-in[OF ‹finite ?I› ‹?I ≠ {}›] by auto
    moreover have  $\forall k < ?i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
      using Min.coboundedI[of ‹{i. i ≤ i_0 ∧ ¬ learn (f i) (f (Suc i)) ∧ ¬ forget_{NOT} (f i) (f (Suc i))}›, simplified]
      by (meson  $\hookrightarrow \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f \ i_0) \ (f \ (\text{Suc } i_0))$ ) 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  $\hookrightarrow \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$  cdcl_{NOT} cdcl_{NOT}.cases
  g-def by auto
{
  fix j
  assume  $j \leq i$ 
  then have learn-or-forget**  $(f \ 0) \ (f \ j)$ 
    apply (induction j)
    apply simp
    by (metis (no-types, lifting) Suc-leD Suc-le-lessD 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-μC[of f 0 f i g 0 A] inv ‹dpll-bj (f i) (g 0)›
  unfolding cdcl_{NOT}-NOT-all-inv-def by linarith

moreover have cdcl_{NOT}-i: cdcl_{NOT}**  $(f \ 0) \ (g \ 0)$ 
  using rtranclp-learn-or-forget-cdcl_{NOT}[of f 0 f i] ‹learn-or-forget** (f 0) (f i)›
  cdcl_{NOT}[of i] unfolding g-def by auto
moreover have  $\bigwedge i. \text{cdcl}_{NOT} (g \ i) \ (g \ (\text{Suc } i))$ 
  using cdcl_{NOT} g-def by auto
moreover have cdcl_{NOT}-NOT-all-inv A  $(g \ 0)$ 
  using inv cdcl_{NOT}-i rtranclp-cdcl_{NOT}-trail-clauses-bound g-def cdcl_{NOT}-NOT-all-inv by auto
ultimately obtain j where  $j: \bigwedge i. i \geq j \implies \text{learn-or-forget } (g \ i) \ (g \ (\text{Suc } i))$ 
  using IH unfolding  $\mu[\text{symmetric}]$  by presburger
show ?thesis

```

```

proof
  {
    fix  $k$ 
    assume  $k \geq j + \text{Suc } i$ 
    then have  $\text{learn-or-forget } (f\ k) (f\ (\text{Suc } k))$ 
      using  $j[\text{of } k - \text{Suc } i]$  unfolding  $g\text{-def}$  by  $\text{auto}$ 
  }
  then show  $\forall k \geq j + \text{Suc } i. \text{learn-or-forget } (f\ k) (f\ (\text{Suc } k))$ 
    by  $\text{auto}$ 
qed
qed
next
case 0 note  $H = \text{this}(1)$  and  $\text{cdcl}_{\text{NOT}} = \text{this}(2)$  and  $\text{inv} = \text{this}(3)$ 
show ?case
proof (rule ccontr)
  assume  $\neg ?\text{case}$ 
  then have  $j: \exists i. \neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i) (f\ (\text{Suc } i))$ 
    by  $\text{blast}$ 
  obtain  $i$  where
     $\neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i) (f\ (\text{Suc } i))$  and
     $\forall k < i. \text{learn-or-forget } (f\ k) (f\ (\text{Suc } k))$ 
  proof -
    obtain  $i_0$  where  $\neg \text{learn } (f\ i_0) (f\ (\text{Suc } i_0)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i_0) (f\ (\text{Suc } i_0))$ 
      using  $j$  by  $\text{auto}$ 
    then have  $\{i. i \leq i_0 \wedge \neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i) (f\ (\text{Suc } i))\} \neq \{\}$ 
      by  $\text{auto}$ 
    let  $?I = \{i. i \leq i_0 \wedge \neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i) (f\ (\text{Suc } i))\}$ 
    let  $?i = \text{Min } ?I$ 
    have  $\text{finite } ?I$ 
      by  $\text{auto}$ 
    have  $\neg \text{learn } (f\ ?i) (f\ (\text{Suc } ?i)) \wedge \neg \text{forget}_{\text{NOT}} (f\ ?i) (f\ (\text{Suc } ?i))$ 
      using  $\text{Min-in}[OF\ \langle \text{finite } ?I \rangle\ \langle ?I \neq \{\} \rangle]$  by  $\text{auto}$ 
    moreover have  $\forall k < ?i. \text{learn-or-forget } (f\ k) (f\ (\text{Suc } k))$ 
      using  $\text{Min.coboundedI}[\text{of } \{i. i \leq i_0 \wedge \neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i) (f\ (\text{Suc } i))\}, \text{simplified}]$ 
      by  $(\text{meson } (\neg \text{learn } (f\ i_0) (f\ (\text{Suc } i_0)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i_0) (f\ (\text{Suc } i_0))) \text{ less-imp-le dual-order.trans not-le})$ 
    ultimately show ?thesis using that by  $\text{blast}$ 
  qed
have  $\text{dpll-bj } (f\ i) (f\ (\text{Suc } i))$ 
  using  $(\neg \text{learn } (f\ i) (f\ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f\ i) (f\ (\text{Suc } i))) \text{ cdcl}_{\text{NOT}} \text{ cdcl}_{\text{NOT}}.\text{cases}$ 
  by  $\text{blast}$ 
  {
    fix  $j$ 
    assume  $j \leq i$ 
    then have  $\text{learn-or-forget}^{**} (f\ 0) (f\ j)$ 
      apply (induction  $j$ )
      apply  $\text{simp}$ 
      by (metis (no-types, lifting)  $\text{Suc-leD Suc-le-lessD rtrancIp.simps}$ 
         $\langle \forall k < i. \text{learn } (f\ k) (f\ (\text{Suc } k)) \vee \text{forget}_{\text{NOT}} (f\ k) (f\ (\text{Suc } k)) \rangle$ )
  }
  then have  $\text{learn-or-forget}^{**} (f\ 0) (f\ i)$  by  $\text{blast}$ 

then show False
  using  $\text{learn-or-forget-dpll-}\mu_C[\text{of } f\ 0\ f\ i\ f\ (\text{Suc } i)\ A]\ \text{inv } 0$ 

```

$\langle \text{dpll-bj } (f \ i) \ (f \ (\text{Suc } i)) \rangle$ **unfolding** $\text{cdcl}_{\text{NOT-} \text{NOT-all-inv-def}}$ **by** *linarith*
qed
qed

lemma $\text{wf-cdcl}_{\text{NOT-no-learn-and-forget-infinite-chain}}$:

assumes

$\text{no-infinite-lf}: \bigwedge f \ j. \neg (\forall i \geq j. \text{learn-or-forget } (f \ i) \ (f \ (\text{Suc } i)))$

shows $\text{wf } \{(T, S). \text{cdcl}_{\text{NOT}} S \ T \wedge \text{cdcl}_{\text{NOT-} \text{NOT-all-inv}} A \ S\}$ **(is** $\text{wf } \{(T, S). \text{cdcl}_{\text{NOT}} S \ T$
 $\wedge ?\text{inv } S\})$

unfolding $\text{wf-iff-no-infinite-down-chain}$

proof (*rule ccontr*)

assume $\neg \neg (\exists f. \forall i. (f \ (\text{Suc } i), f \ i) \in \{(T, S). \text{cdcl}_{\text{NOT}} S \ T \wedge ?\text{inv } S\})$

then obtain f **where**

$\forall i. \text{cdcl}_{\text{NOT}} (f \ i) \ (f \ (\text{Suc } i)) \wedge ?\text{inv } (f \ i)$

by *fast*

then have $\exists j. \forall i \geq j. \text{learn-or-forget } (f \ i) \ (f \ (\text{Suc } i))$

using $\text{infinite-cdcl}_{\text{NOT-exists-learn-and-forget-infinite-chain}}[\text{of } f]$ **by** *meson*

then show *False* **using** no-infinite-lf **by** *blast*

qed

lemma $\text{inv-and-tranclp-cdcl}_{\text{NOT-tranclp-cdcl}_{\text{NOT-and-inv}}}$:

$\text{cdcl}_{\text{NOT}}^{++} S \ T \wedge \text{cdcl}_{\text{NOT-} \text{NOT-all-inv}} A \ S \longleftrightarrow (\lambda S \ T. \text{cdcl}_{\text{NOT}} S \ T \wedge \text{cdcl}_{\text{NOT-} \text{NOT-all-inv}} A \ S)$
 $\text{cdcl}_{\text{NOT}}^{++} S \ T$

(is $?A \wedge ?I \longleftrightarrow ?B$)

proof

assume $?A \wedge ?I$

then have $?A$ **and** $?I$ **by** *blast+*

then show $?B$

apply *induction*

apply (*simp add: tranclp.r-into-trancl*)

by (*metis (no-types, lifting) cdcl_{\text{NOT-} \text{NOT-all-inv}} tranclp.simps tranclp-into-rtranclp*)

next

assume $?B$

then have $?A$ **by** *induction auto*

moreover have $?I$ **using** $\langle ?B \rangle \text{tranclpD}$ **by** *fastforce*

ultimately show $?A \wedge ?I$ **by** *blast*

qed

lemma $\text{wf-tranclp-cdcl}_{\text{NOT-no-learn-and-forget-infinite-chain}}$:

assumes

$\text{no-infinite-lf}: \bigwedge f \ j. \neg (\forall i \geq j. \text{learn-or-forget } (f \ i) \ (f \ (\text{Suc } i)))$

shows $\text{wf } \{(T, S). \text{cdcl}_{\text{NOT}}^{++} S \ T \wedge \text{cdcl}_{\text{NOT-} \text{NOT-all-inv}} A \ S\}$

using $\text{wf-tranclp}[\text{OF } \text{wf-cdcl}_{\text{NOT-no-learn-and-forget-infinite-chain}}[\text{OF } \text{no-infinite-lf}]]$

apply (*rule wf-subset*)

by (*auto simp: trancl-set-tranclp inv-and-tranclp-cdcl_{\text{NOT-tranclp-cdcl}_{\text{NOT-and-inv}}}*)

lemma $\text{cdcl}_{\text{NOT-final-state}}$:

assumes

$n\text{-s}$: *no-step* $\text{cdcl}_{\text{NOT}} S$ **and**

inv : $\text{cdcl}_{\text{NOT-} \text{NOT-all-inv}} A \ S$ **and**

decomp : *all-decomposition-implies-m* (*clauses* S) (*get-all-marked-decomposition* (*trail* S))

shows *unsatisfiable* (*set-mset* (*clauses* S))

$\vee (\text{trail } S \models_{\text{asm}} \text{clauses } S \wedge \text{satisfiable } (\text{set-mset } (\text{clauses } S)))$

proof —

have $n\text{-s}'$: *no-step* $\text{dpll-bj } S$

```

    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))
    ∨ (trail T ⊨asm clauses T ∧ 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
λC S. distinct-mset C ∧ ¬tautology C ∧ learn-restrictions C S ∧
  (∃ F K d F' C' L. trail S = F' @ Marked K () # F ∧ C = C' + {#L#} ∧ F ⊨as CNot C'
    ∧ C' + {#L#} ⊄ clauses S)
λC S. ¬(∃ F' F K d L. trail S = F' @ Marked K () # F ∧ F ⊨as CNot (C - {#L#}))
  ∧ forget-restrictions C S
for
  trail :: 'st ⇒ ('v, unit, unit) marked-lits and
  clauses :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clsNOT remove-clsNOT:: 'v clause ⇒ 'st ⇒ 'st and
  propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
  inv :: 'st ⇒ bool and
  backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
  learn-restrictions forget-restrictions :: 'v clause ⇒ 'st ⇒ 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: ∧T. dpll-bj S T ⇒ P S T and
  learning:

```

```

 $\bigwedge C F K F' C' L T. \text{ clauses } S \models_{pm} C$ 
 $\implies \text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' (\text{lits-of } (\text{trail } S))$ 
 $\implies \text{distinct-mset } C \implies \neg \text{tautology } C \implies \text{learn-restrictions } C S$ 
 $\implies \text{trail } S = F' @ \text{Marked } K () \# F \implies C = C' + \{\#L\# \} \implies F \models_{as} CNot C'$ 
 $\implies C' + \{\#L\# \} \notin \text{clauses } S \implies T \sim \text{add-cl}_{NOT} C S$ 
 $\implies P S T$  and
forgetting:  $\bigwedge C T. \text{ clauses } S - \text{replicate-mset } (\text{count } (\text{clauses } S) C) C \models_{pm} C$ 
 $\implies C \in \# \text{ clauses } S$ 
 $\implies \neg (\exists F' F K L. \text{ trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} CNot (C - \{\#L\# \}))$ 
 $\implies T \sim \text{remove-cl}_{NOT} C S$ 
 $\implies \text{forget-restrictions } C S \implies P S T$ 
shows  $P S T$ 
using assms(1)
apply (induction rule: cdclNOT.induct)
  apply (auto dest: assms(2) simp add: learn-ops-axioms)[]
  apply (auto elim!: learn-ops.learn.cases[OF learn-ops-axioms] dest: assms(3))[]
apply (auto elim!: forget-ops.forgetNOT.cases[OF forget-ops-axioms] dest!: assms(4))
done

lemma rtranclp-cdclNOT-inv:
 $cdcl_{NOT}^{**} S T \implies inv S \implies inv T$ 
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)
 $\subseteq \text{simple-clss } (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' \text{ lits-of } (\text{trail } S))$ 
proof
fix  $C$  assume  $C: C \in \text{set-mset } (\text{clauses } T - \text{clauses } S)$ 
have distinct-mset C  $\neg \text{tautology } C$  using learn C n-d by (elim learnNOTE; auto)+
then have  $C \in \text{simple-clss } (\text{atms-of } C)$ 
  using distinct-mset-not-tautology-implies-in-simple-clss by blast
moreover have  $\text{atms-of } C \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' \text{ lits-of } (\text{trail } S)$ 
  using learn C n-d by (elim learnNOTE) (auto simp: atms-of-ms-def atms-of-def image-Un
true-annots-CNot-all-atms-defined)
moreover have  $\text{finite } (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' \text{ lits-of } (\text{trail } S))$ 
  by auto
ultimately show  $C \in \text{simple-clss } (\text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of } ' \text{ lits-of } (\text{trail } S))$ 
  using simple-clss-mono by (metis (no-types) insert-subset mk-disjoint-insert)
qed

definition conflicting-bj-clss S  $\equiv$ 
 $\{C + \{\#L\#\} \mid C L. C + \{\#L\#\} \in \# \text{ clauses } S \wedge \text{distinct-mset } (C + \{\#L\#\}) \wedge \neg \text{tautology } (C + \{\#L\#\})$ 
 $\wedge (\exists F' K F. \text{ trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} CNot C)\}$ 

lemma conflicting-bj-clss-remove-clNOT[simp]:
 $\text{conflicting-bj-clss } (\text{remove-cl}_{NOT} C S) = \text{conflicting-bj-clss } S - \{C\}$ 
unfolding conflicting-bj-clss-def by fastforce

lemma conflicting-bj-clss-add-clNOT-state-eq:

```

$T \sim \text{add-cl}_{\text{NOT}} C' S \implies \text{no-dup} (\text{trail } S) \implies \text{conflicting-bj-clss } T$
 $= \text{conflicting-bj-clss } S$
 $\cup (\text{if } \exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset} (C + \{\#L\# \}) \wedge \neg \text{tautology} (C + \{\#L\# \})$
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} C \text{Not } C)$
 $\text{then } \{C'\} \text{ else } \{\})$
unfolding *conflicting-bj-clss-def* **by** *auto metis+*

lemma *conflicting-bj-clss-add-cl_{NOT}*:
 $\text{no-dup} (\text{trail } S) \implies$
 $\text{conflicting-bj-clss} (\text{add-cl}_{\text{NOT}} C' S)$
 $= \text{conflicting-bj-clss } S$
 $\cup (\text{if } \exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset} (C + \{\#L\# \}) \wedge \neg \text{tautology} (C + \{\#L\# \})$
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{\text{as}} C \text{Not } C)$
 $\text{then } \{C'\} \text{ else } \{\})$
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 learn_{NOT}E*)
by (*subst conflicting-bj-clss-add-cl_{NOT}-state-eq[of T]*) *auto*

abbreviation *conflicting-bj-clss-yet b S* \equiv
 $\exists \wedge b - \text{card} (\text{conflicting-bj-clss } S)$

abbreviation $\mu_L :: \text{nat} \Rightarrow 'st \Rightarrow \text{nat} \times \text{nat}$ **where**
 $\mu_L b S \equiv (\text{conflicting-bj-clss-yet } b S, \text{card} (\text{set-mset} (\text{clauses } S)))$

lemma *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*:
assumes *forget_{NOT} S T*
shows $\text{conflicting-bj-clss } S = \text{conflicting-bj-clss } T$
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 (*metis card-Diff1-less clauses-remove-cl_{NOT} finite-set-mset mem-set-mset-iff order-refl*
set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses)
then show *?thesis*
unfolding *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched[OF forget_{NOT}]*
by *auto*

qed

lemma *set-condition-or-split*:

$\{a. (a = b \vee Q a) \wedge S a\} = (\text{if } S b \text{ then } \{b\} \text{ else } \{\}) \cup \{a. Q a \wedge S a\}$
 by *auto*

lemma *set-insert-neg*:

$A \neq \text{insert } a A \longleftrightarrow a \notin A$
 by *auto*

lemma *learn- μ_L -decrease*:

assumes *learnST*: *learn* $S T$ **and** *n-d*: *no-dup* (*trail* S) **and**
A: *atms-of-msu* (*clauses* S) \cup *atm-of* ‘*lits-of* (*trail* S)’ $\subseteq A$ **and**
fin-A: *finite* A
shows $(\mu_L (\text{card } A) T, \mu_L (\text{card } A) S) \in \text{less-than} <*\text{lex}*> \text{less-than}$

proof –

have [*simp*]: $(\text{atms-of-msu} (\text{clauses } T) \cup \text{atm-of} \text{ ‘lits-of } (\text{trail } T))$
 $= (\text{atms-of-msu} (\text{clauses } S) \cup \text{atm-of} \text{ ‘lits-of } (\text{trail } S))$
using *learnST* *n-d* **by** (*elim learn_{NOT}E*) *auto*

then have $\text{card } (\text{atms-of-msu} (\text{clauses } T) \cup \text{atm-of} \text{ ‘lits-of } (\text{trail } T))$
 $= \text{card } (\text{atms-of-msu} (\text{clauses } S) \cup \text{atm-of} \text{ ‘lits-of } (\text{trail } S))$
by (*auto intro!*: *card-mono*)

then have $3: (3::\text{nat}) \wedge \text{card } (\text{atms-of-msu} (\text{clauses } T) \cup \text{atm-of} \text{ ‘lits-of } (\text{trail } T))$
 $= 3 \wedge \text{card } (\text{atms-of-msu} (\text{clauses } S) \cup \text{atm-of} \text{ ‘lits-of } (\text{trail } S))$
by (*auto intro*: *power-mono*)

moreover have *conflicting-bj-clss* $S \subseteq \text{conflicting-bj-clss } T$
using *learnST* *n-d* **by** (*simp add*: *learn-conflicting-increasing*)

moreover have *conflicting-bj-clss* $S \neq \text{conflicting-bj-clss } T$
using *learnST*

proof (*elim learn_{NOT}E*, *goal-cases*)

case (*1 C*) **note** *clss-S* = *this*(*1*) **and** *atms-C* = *this*(*2*) **and** *inv* = *this*(*3*) **and** $T = \text{this}(4)$

then obtain $F K F' C' L$ **where**

tr-S: *trail* $S = F' @ \text{Marked } K () \# F$ **and**

$C: C = C' + \{\#L\# \}$ **and**

$F: F \models_{\text{as}} C \text{Not } C'$ **and**

$C-S: C' + \{\#L\# \} \notin \text{clauses } S$

by *blast*

moreover have *distinct-mset* $C \neg \text{tautology } C$ **using** *inv* **by** *blast+*

ultimately have $C' + \{\#L\# \} \in \text{conflicting-bj-clss } T$

using T *n-d* **unfolding** *conflicting-bj-clss-def* **by** *fastforce*

moreover have $C' + \{\#L\# \} \notin \text{conflicting-bj-clss } S$

using $C-S$ **unfolding** *conflicting-bj-clss-def* **by** *auto*

ultimately show *?case* **by** *blast*

qed

moreover have *fin-T*: *finite* (*conflicting-bj-clss* T)

using *learnST* **by** *induction* (*auto simp add*: *conflicting-bj-clss-add-clss_{NOT}*)

ultimately have $\text{card } (\text{conflicting-bj-clss } T) \geq \text{card } (\text{conflicting-bj-clss } S)$

using *card-mono* **by** *blast*

moreover

have *fin'*: *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: conflicting-bj-clss T
 $\subseteq$  simple-clss (atms-of-msu (clauses T)  $\cup$  atm-of ' lits-of (trail T))
  by standard (meson 1 2 fin'  $\langle$ finite (conflicting-bj-clss T) $\rangle$  simple-clss-mono
    distinct-mset-set-def simplified-in-simple-clss subsetCE sup.coboundedI1)
moreover
then have #: 3  $\wedge$  card (atms-of-msu (clauses T)  $\cup$  atm-of ' lits-of (trail T))
   $\geq$  card (conflicting-bj-clss T)
  by (meson Nat.le-trans simple-clss-card simple-clss-finite card-mono fin')
have atms-of-msu (clauses T)  $\cup$  atm-of ' lits-of (trail T)  $\subseteq$  A
  using learnNOTE[OF learnST] A by simp
then have 3  $\wedge$  (card A)  $\geq$  card (conflicting-bj-clss T)
  using # fin-A by (meson simple-clss-card simple-clss-finite
    simple-clss-mono calculation(2) card-mono dual-order.trans)
ultimately show ?thesis
  using psubset-card-mono[OF fin-T]
  unfolding less-than-iff lex-prod-def by clarify
  (meson  $\langle$ conflicting-bj-clss S  $\neq$  conflicting-bj-clss T $\rangle$ 
     $\langle$ conflicting-bj-clss S  $\subseteq$  conflicting-bj-clss T $\rangle$ 
    diff-less-mono2 le-less-trans not-le psubsetI)
qed

```

We have to assume the following:

- *inv* S: the invariant holds in the initial state.
- A is a (finite *finite* A) superset of the literals in the trail *atm-of ' lits-of (trail S)* \subseteq *atms-of-ms* A and in the clauses *atms-of-msu (clauses S)* \subseteq *atms-of-ms* A. This can be the set of all the literals in the starting set of clauses.
- *no-dup (trail S)*: no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} where

$\mu_{CDCL} A T \equiv ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T),$
 $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T, \text{card } (\text{set-mset } (\text{clauses } T)))$

lemma *cdcl_{NOT}-decreasing-measure*:

```

assumes
  cdclNOT S T and
  inv: inv S and
  atm-clss: atms-of-msu (clauses S)  $\subseteq$  atms-of-ms A and
  atm-lits: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-ms A and
  n-d: no-dup (trail S) and
  fin-A: finite A
shows ( $\mu_{CDCL} A T, \mu_{CDCL} A S$ )
   $\in$  less-than <*lex*> (less-than <*lex*> less-than)
using assms(1)
proof induction
  case (c-dpll-bj T)
  from dpll-bj-trail-mes-decreasing-prop[OF this(1) inv atm-clss atm-lits n-d fin-A]
  show ?case unfolding  $\mu_{CDCL}$ -def
    by (meson in-lex-prod less-than-iff)
next
  case (c-learn T) note learn = this(1)

```

then have S : $\text{trail } S = \text{trail } T$
using $\text{inv atm-clss atm-lits n-d fin-A}$
by $(\text{elim learn}_{NOT} E)$ **auto**
show $?case$
using $\text{learn-}\mu_L\text{-decrease}[OF \text{ learn - }] \text{ atm-clss atm-lits fin-A n-d unfolding } S \mu_{CDCL}\text{-def}$ **by** auto
next
case $(c\text{-forget}_{NOT} T)$ **note** $\text{forget}_{NOT} = \text{this}(1)$
have $\text{trail } S = \text{trail } T$ **using** forget_{NOT} **by** induction auto
then show $?case$
using $\text{forget-}\mu_L\text{-decrease}[OF \text{ forget}_{NOT}]$ **unfolding** $\mu_{CDCL}\text{-def}$ **by** auto
qed

lemma $\text{wf-cdcl}_{NOT}\text{-restricted-learning}$:

assumes $\text{finite } A$
shows $\text{wf } \{(T, S).$
 $(\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S)$
 $\wedge \text{inv } S)$
 $\wedge \text{cdcl}_{NOT} S T \}$
by $(\text{rule wf-wf-if-measure'}[\text{of less-than } <*lex*> (\text{less-than } <*lex*> \text{less-than})])$
 $(\text{auto intro: cdcl}_{NOT}\text{-decreasing-measure}[OF \text{ - - - - assms}])$

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 $\text{cdcl}_{NOT}\text{-decreasing-measure'}$:

assumes
 $\text{cdcl}_{NOT} S T$ **and**
 $\text{inv: inv } S$ **and**
 $\text{atms-clss: atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atms-trail: atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{n-d: no-dup } (\text{trail } S)$ **and**
 $\text{fin-A: finite } A$

shows $\mu_{CDCL}' A T < \mu_{CDCL}' A S$

using $\text{assms}(1)$

proof $(\text{induction rule: cdcl}_{NOT}\text{-learn-all-induct})$

case $(\text{dpll-bj } T)$

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 $\text{dpll-bj-trail-mes-decreasing-prop fin-A inv n-d atms-clss atms-trail}$

unfolding $\mu_C'\text{-def}$ **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 $\text{mult-le-mono1}[OF \text{ this, of } (1 + 3^{\text{card } (\text{atms-of-ms } A)})]$

have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T) *$

$(1 + 3^{\text{card } (\text{atms-of-ms } A)}) + (1 + 3^{\text{card } (\text{atms-of-ms } A)})$

$\leq ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A S)$

```

    * (1 + 3 ^ card (atms-of-ms A))
  unfolding Nat.add-mult-distrib
  by presburger
moreover
  have cl-T-S: clauses T = clauses S
    using dpll-bj.hyps inv dpll-bj-clauses by auto
  have conflicting-bj-clss-yet (card (atms-of-ms A)) S < 1 + 3 ^ card (atms-of-ms A)
    by simp
ultimately have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A T)
  * (1 + 3 ^ card (atms-of-ms A)) + conflicting-bj-clss-yet (card (atms-of-ms A)) T
  < ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A S) * (1 + 3 ^ card (atms-of-ms
A))
  by linarith
then have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A T)
  * (1 + 3 ^ card (atms-of-ms A))
  + conflicting-bj-clss-yet (card (atms-of-ms A)) T
  < ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A S)
  * (1 + 3 ^ card (atms-of-ms A))
  + conflicting-bj-clss-yet (card (atms-of-ms A)) S
  by linarith
then have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A T)
  * (1 + 3 ^ card (atms-of-ms A)) * 2
  + conflicting-bj-clss-yet (card (atms-of-ms A)) T * 2
  < ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A S)
  * (1 + 3 ^ card (atms-of-ms A)) * 2
  + conflicting-bj-clss-yet (card (atms-of-ms A)) S * 2
  by linarith
then show ?case unfolding μCDCL'-def cl-T-S by presburger
next
case (learn C F' K F C' L T) note clss-S-C = this(1) and atms-C = this(2) and dist = this(3)
  and tauto = this(4) and learn-restr = this(5) and tr-S = this(6) and C' = this(7) and
  F-C = this(8) and C-new = this(9) and T = this(10)
have insert C (conflicting-bj-clss S) ⊆ simple-clss (atms-of-ms A)
proof -
  have C ∈ simple-clss (atms-of-ms A)
  by (metis (no-types, hide-lams) Un-subset-iff atms-of-ms-finite simple-clss-mono
    contra-subsetD dist distinct-mset-not-tautology-implies-in-simple-clss
    dual-order.trans fin-A atms-C atms-clss atms-trail tauto)
moreover have conflicting-bj-clss S ⊆ simple-clss (atms-of-ms A)
  unfolding conflicting-bj-clss-def
  proof
    fix x :: 'v literal multiset
    assume x ∈ {C + {#L#} | C L. C + {#L#} ∈ # clauses S
      ∧ distinct-mset (C + {#L#}) ∧ ¬ tautology (C + {#L#})
      ∧ (∃ F' K F. trail S = F' @ Marked K () # F ∧ F ⊢as CNot C)}
    then have ∃ m l. x = m + {#l#} ∧ m + {#l#} ∈ # clauses S
      ∧ distinct-mset (m + {#l#}) ∧ ¬ tautology (m + {#l#})
      ∧ (∃ ms l msa. trail S = ms @ Marked l () # msa ∧ msa ⊢as CNot m)
    by blast
    then show x ∈ simple-clss (atms-of-ms A)
    by (meson atms-clss atms-of-atms-of-ms-mono atms-of-ms-finite simple-clss-mono
      distinct-mset-not-tautology-implies-in-simple-clss fin-A finite-subset
      mem-set-mset-iff set-rev-mp)
  qed
ultimately show ?thesis

```

```

    by auto
  qed
then have card (insert C (conflicting-bj-clss S)) ≤ 3 ^ (card (atms-of-ms A))
  by (meson Nat.le-trans atms-of-ms-finite simple-clss-card simple-clss-finite
      card-mono fin-A)
moreover have [simp]: card (insert C (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]: conflicting-bj-clss (add-clssNOT C S) = conflicting-bj-clss S ∪ {C}
  using dist tauto F-C n-d by (subst conflicting-bj-clss-add-clssNOT)
  (force simp add: ac-simps C' tr-S)+
ultimately have [simp]: conflicting-bj-clss-yet (card (atms-of-ms A)) S
  = Suc (conflicting-bj-clss-yet (card (atms-of-ms A)) (add-clssNOT C S))
  by simp
have 1: clauses T = clauses (add-clssNOT C S) using T by auto
have 2: conflicting-bj-clss-yet (card (atms-of-ms A)) T
  = conflicting-bj-clss-yet (card (atms-of-ms A)) (add-clssNOT C S)
  using T unfolding conflicting-bj-clss-def by auto
have 3: μC' A T = μC' A (add-clssNOT C S)
  using T unfolding μC'-def by auto
have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A (add-clssNOT C S))
  * (1 + 3 ^ card (atms-of-ms A)) * 2
  = ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A)) - μC' A S)
  * (1 + 3 ^ card (atms-of-ms A)) * 2
  using n-d unfolding μC'-def by auto
moreover
  have conflicting-bj-clss-yet (card (atms-of-ms A)) (add-clssNOT C S)
    * 2
    + card (set-mset (clauses (add-clssNOT C S)))
    < conflicting-bj-clss-yet (card (atms-of-ms A)) S * 2
    + card (set-mset (clauses S))
  by (simp add: C' C-new n-d)
ultimately show ?case unfolding μCDCL'-def 1 2 3 by presburger
next
case (forgetNOT C T) note T = this(4)
have [simp]: μC' A (remove-clssNOT C S) = μC' A S
  unfolding μC'-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 card (set-mset (clauses T)) < card (set-mset (clauses S))
  by (metis T card-Diff1-less clauses-remove-clssNOT 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 μCDCL'-def
  by (metis (no-types) T μC' A (remove-clssNOT C S) = μC' A S add-le-cancel-left
      μC'-def not-le state-eqNOT-trail)
qed

```

lemma *cdcl_{NOT}-clauses-bound*:
assumes
cdcl_{NOT} S T and
inv S and
atms-of-msu (clauses S) ⊆ A and

$atm\text{-}of \text{ '}(lits\text{-}of \text{ (trail } S)) \subseteq A$ and
 $n\text{-}d: no\text{-}dup \text{ (trail } S)$ and
 $fin\text{-}A[simp]: finite \ A$
shows $set\text{-}mset \text{ (clauses } T) \subseteq set\text{-}mset \text{ (clauses } S) \cup simple\text{-}clss \ A$
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** $atms\text{-}C = this(2)$ **and** $dist = this(3)$ **and** $tauto = this(4)$ **and**
 $T = this(10)$ **and** $atms\text{-}clss\text{-}S = this(12)$ **and** $atms\text{-}trail\text{-}S = this(13)$
have $atms\text{-}of \ C \subseteq A$
using $atms\text{-}C \ atms\text{-}clss\text{-}S \ atms\text{-}trail\text{-}S$ **by** *auto*
then have $simple\text{-}clss \ (atms\text{-}of \ C) \subseteq simple\text{-}clss \ A$
by (*simp add: simple-clss-mono*)
then have $C \in simple\text{-}clss \ A$
using *finite dist tauto*
by (*auto dest: distinct-mset-not-tautology-implies-in-simple-clss*)
then show *?case using T n-d by auto*
qed

lemma *rtrancpl-cdcl_{NOT}-clauses-bound:*

assumes
 $cdcl_{NOT}^{**} \ S \ T$ **and**
 $inv \ S$ **and**
 $atms\text{-}of\text{-}msu \text{ (clauses } S) \subseteq A$ **and**
 $atm\text{-}of \text{ '}(lits\text{-}of \text{ (trail } S)) \subseteq A$ **and**
 $n\text{-}d: no\text{-}dup \text{ (trail } S)$ **and**
 $finite: finite \ A$
shows $set\text{-}mset \text{ (clauses } T) \subseteq set\text{-}mset \text{ (clauses } S) \cup simple\text{-}clss \ A$
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** $atms\text{-}clss\text{-}S = this(5)$ **and** $atms\text{-}trail\text{-}S = this(6)$ **and** $finite\text{-}cls\text{-}S = this(7)$
have $inv \ T$
using *rtrancpl-cdcl_{NOT}-inv st inv by blast*
moreover have $atms\text{-}of\text{-}msu \text{ (clauses } T) \subseteq A$ **and** $atm\text{-}of \text{ '}(lits\text{-}of \text{ (trail } T)) \subseteq A$
using *rtrancpl-cdcl_{NOT}-trail-clauses-bound[OF st] inv atms-clss-S atms-trail-S n-d by blast+*
moreover have $no\text{-}dup \text{ (trail } T)$
using *rtrancpl-cdcl_{NOT}-no-dup[OF st (inv S) n-d] by simp*
ultimately have $set\text{-}mset \text{ (clauses } U) \subseteq set\text{-}mset \text{ (clauses } T) \cup simple\text{-}clss \ A$
using $cdcl_{NOT} \ finite \ n\text{-}d$ **by** (*auto simp: cdcl_{NOT}-clauses-bound*)
then show *?case using IH by auto*
qed

lemma *rtrancpl-cdcl_{NOT}-card-clauses-bound:*

assumes
 $cdcl_{NOT}^{**} \ S \ T$ **and**

inv S and
atms-of-msu (clauses S) \subseteq A and
atm-of '(lits-of (trail S)) \subseteq A and
n-d: no-dup (trail S) and
finite: finite A
shows *card (set-mset (clauses T)) \leq card (set-mset (clauses S)) + 3 \wedge (card A)*
using *rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] finite by (meson Nat.le-trans*
simple-clss-card simple-clss-finite card-Un-le card-mono finite-UnI
finite-set-mset nat-add-left-cancel-le)

lemma *rtrancpl-cdcl_{NOT}-card-clauses-bound':*

assumes
*cdcl_{NOT}** S T and*
inv S and
atms-of-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 *card {C | C. C \in # clauses T \wedge (tautology C \vee \neg distinct-mset C)}*
 \leq card {C | C. C \in # clauses S \wedge (tautology C \vee \neg distinct-mset C)} + 3 \wedge (card A)
(is card ?T \leq card ?S + -)
using *rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] finite*
proof –
have *?T \subseteq ?S \cup simple-clss A*
using *rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] by force*
then have *card ?T \leq card (?S \cup simple-clss A)*
using *finite by (simp add: assms(5) simple-clss-finite card-mono)*
then show *?thesis*
by *(meson le-trans simple-clss-card card-Un-le local.finite nat-add-left-cancel-le)*
qed

lemma *rtrancpl-cdcl_{NOT}-card-simple-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 *card (set-mset (clauses T))*
 \leq card {C. C \in # clauses S \wedge (tautology C \vee \neg distinct-mset C)} + 3 \wedge (card A)
(is card ?T \leq card ?S + -)
using *rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] finite*
proof –
have *$\bigwedge x. x \in \# \text{ clauses } T \implies \neg \text{tautology } x \implies \text{distinct-mset } x \implies x \in \text{simple-clss } A$*
using *rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] by (metis (no-types, hide-lams) Un-iff assms(3)*
atms-of-atms-of-ms-mono simple-clss-mono contra-subsetD
distinct-mset-not-tautology-implies-in-simple-clss local.finite mem-set-mset-iff
subset-trans)
then have *set-mset (clauses T) \subseteq ?S \cup simple-clss A*
using *rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] by auto*
then have *card(set-mset (clauses T)) \leq card (?S \cup simple-clss A)*
using *finite by (simp add: assms(5) simple-clss-finite card-mono)*
then show *?thesis*
by *(meson le-trans simple-clss-card card-Un-le local.finite nat-add-left-cancel-le)*

qed

definition $\mu_{CDCL}'\text{-bound} :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow nat$ **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 $\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}[\text{simp}]$:

$\mu_{CDCL}'\text{-bound } A \ (\text{reduce-trail-to}_{NOT} \ M \ S) = \mu_{CDCL}'\text{-bound } A \ S$

unfolding $\mu_{CDCL}'\text{-bound-def}$ **by** *auto*

lemma $rtrancpl\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}$:

assumes

$cdcl_{NOT}^{**} \ S \ T$ **and**

$inv \ S$ **and**

$\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

$\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d: no-dup } (\text{trail } S)$ **and**

$\text{finite: finite } (\text{atms-of-ms } A)$ **and**

$U: U \sim \text{reduce-trail-to}_{NOT} \ M \ T$

shows $\mu_{CDCL}' \ A \ U \leq \mu_{CDCL}'\text{-bound } A \ S$

proof –

have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' \ A \ U)$

$\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$

by *auto*

then have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' \ A \ U)$

$* (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$

$\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$

using *mult-le-mono1* **by** *blast*

moreover

have $\text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) \ T * 2 \leq 2 * 3 \wedge \text{card } (\text{atms-of-ms } A)$

by *linarith*

moreover have $\text{card } (\text{set-mset } (\text{clauses } U))$

$\leq \text{card } \{C. C \in \# \text{ clauses } S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge \text{card } (\text{atms-of-ms } A)$

using $rtrancpl\text{-}cdcl_{NOT}\text{-card-simple-clauses-bound}[OF \ \text{assms}(1-6)] \ U$ **by** *auto*

ultimately show *?thesis*

unfolding $\mu_{CDCL}'\text{-def}$ $\mu_{CDCL}'\text{-bound-def}$ **by** *linarith*

qed

lemma $rtrancpl\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-bound}$:

assumes

$cdcl_{NOT}^{**} \ S \ T$ **and**

$inv \ S$ **and**

$\text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

$\text{atm-of } '(\text{lits-of } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d: no-dup } (\text{trail } S)$ **and**

$\text{finite: finite } (\text{atms-of-ms } A)$

shows $\mu_{CDCL}' \ A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$

proof –

have $\mu_{CDCL}' \ A \ (\text{reduce-trail-to}_{NOT} \ (\text{trail } T) \ T) = \mu_{CDCL}' \ A \ T$

unfolding $\mu_{CDCL}'\text{-def}$ $\mu_C'\text{-def}$ $\text{conflicting-bj-clss-def}$ **by** *auto*

then show *?thesis* **using** $rtrancpl\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}[OF \ \text{assms}, \text{ of - trail } T]$

state-eq_{NOT}\text{-ref} **by** *fastforce*

qed

lemma *rtrancp- μ_{CDCL} '-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 $C \notin \text{simple-clss (atms-of-ms A)}$

by (*auto dest: simple-clssE*)

then show $C \in ?S$

using *C-T rtrancp-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-learn*

14.7 CDCL with restarts

14.7.1 Definition

locale *restart-ops =*

fixes

cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool and

restart :: 'st \Rightarrow 'st \Rightarrow bool

begin

inductive *cdcl_{NOT}-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-cl_{NOT} remove-cl_{NOT} :: 'v clause \Rightarrow 'st \Rightarrow 'st and

propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool and


```

    inv :: 'st ⇒ bool and
    backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
    learn-cond forget-cond :: 'v clause ⇒ 'st ⇒ bool
begin

lemma cdclNOT-iff-cdclNOT-raw-restart-no-restarts:
  cdclNOT S T ⟷ restart-ops.cdclNOT-raw-restart cdclNOT (λ-. False) S T
  (is ?C S T ⟷ ?R S T)
proof
  fix S T
  assume ?C S T
  then show ?R S T by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
next
  fix S T
  assume ?R S T
  then show ?C S T
    apply (cases rule: restart-ops.cdclNOT-raw-restart.cases)
    using ⟨?R S T⟩ by fast+
qed

lemma cdclNOT-cdclNOT-raw-restart:
  cdclNOT S T ⟹ restart-ops.cdclNOT-raw-restart cdclNOT restart S T
  by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
end

```

14.7.2 Increasing restarts

To add restarts we need some assumptions on the predicate (called *cdcl_{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 ⇒ 'st ⇒ bool and
  cdclNOT :: 'st ⇒ 'st ⇒ bool +
fixes
  f :: nat ⇒ nat and
  bound-inv :: 'bound ⇒ 'st ⇒ bool and

```

$\mu :: 'bound \Rightarrow 'st \Rightarrow nat$ **and**
 $cdcl_{NOT-inv} :: 'st \Rightarrow bool$ **and**
 $\mu-bound :: 'bound \Rightarrow 'st \Rightarrow nat$
assumes
 $f: unbounded\ f$ **and**
 $f-ge-1: \bigwedge n. n \geq 1 \Rightarrow f\ n \neq 0$ **and**
 $bound-inv: \bigwedge A\ S\ T. cdcl_{NOT-inv}\ S \Rightarrow bound-inv\ A\ S \Rightarrow cdcl_{NOT}\ S\ T \Rightarrow bound-inv\ A\ T$ **and**
 $cdcl_{NOT-measure}: \bigwedge A\ S\ T. cdcl_{NOT-inv}\ S \Rightarrow bound-inv\ A\ S \Rightarrow cdcl_{NOT}\ S\ T \Rightarrow \mu\ A\ T < \mu$
 $A\ S$ **and**
 $measure-bound2: \bigwedge A\ T\ U. cdcl_{NOT-inv}\ T \Rightarrow bound-inv\ A\ T \Rightarrow cdcl_{NOT}^{**}\ T\ U$
 $\Rightarrow \mu\ A\ U \leq \mu-bound\ A\ T$ **and**
 $measure-bound4: \bigwedge A\ T\ U. cdcl_{NOT-inv}\ T \Rightarrow bound-inv\ A\ T \Rightarrow cdcl_{NOT}^{**}\ T\ U$
 $\Rightarrow \mu-bound\ A\ U \leq \mu-bound\ A\ T$ **and**
 $cdcl_{NOT-restart-inv}: \bigwedge A\ U\ V. cdcl_{NOT-inv}\ U \Rightarrow restart\ U\ V \Rightarrow bound-inv\ A\ U \Rightarrow bound-inv$
 $A\ V$
and
 $exists-bound: \bigwedge R\ S. cdcl_{NOT-inv}\ R \Rightarrow restart\ R\ S \Rightarrow \exists A. bound-inv\ A\ S$ **and**
 $cdcl_{NOT-inv}: \bigwedge S\ T. cdcl_{NOT-inv}\ S \Rightarrow cdcl_{NOT}\ S\ T \Rightarrow cdcl_{NOT-inv}\ T$ **and**
 $cdcl_{NOT-inv-restart}: \bigwedge S\ T. cdcl_{NOT-inv}\ S \Rightarrow restart\ S\ T \Rightarrow cdcl_{NOT-inv}\ T$
begin

lemma $cdcl_{NOT-cdcl_{NOT-inv}}$:
assumes
 $(cdcl_{NOT} \sim^n) S\ T$ **and**
 $cdcl_{NOT-inv}\ S$
shows $cdcl_{NOT-inv}\ T$
using *assms* **by** (*induction* n *arbitrary: T*) (*auto intro: bound-inv cdcl_{NOT-inv}*)

lemma $cdcl_{NOT-bound-inv}$:
assumes
 $(cdcl_{NOT} \sim^n) S\ T$ **and**
 $cdcl_{NOT-inv}\ S$
 $bound-inv\ A\ S$
shows $bound-inv\ A\ T$
using *assms* **by** (*induction* n *arbitrary: T*) (*auto intro: bound-inv cdcl_{NOT-cdcl_{NOT-inv}}*)

lemma $rtrancpl-cdcl_{NOT-cdcl_{NOT-inv}}$:
assumes
 $cdcl_{NOT}^{**}\ S\ T$ **and**
 $cdcl_{NOT-inv}\ S$
shows $cdcl_{NOT-inv}\ T$
using *assms* **by** *induction* (*auto intro: cdcl_{NOT-inv}*)

lemma $rtrancpl-cdcl_{NOT-bound-inv}$:
assumes
 $cdcl_{NOT}^{**}\ S\ T$ **and**
 $bound-inv\ A\ S$ **and**
 $cdcl_{NOT-inv}\ S$
shows $bound-inv\ A\ T$
using *assms* **by** *induction* (*auto intro: bound-inv rtrancpl-cdcl_{NOT-cdcl_{NOT-inv}}*)

lemma $cdcl_{NOT-comp-n-le}$:
assumes
 $(cdcl_{NOT} \sim^n (Suc\ n)) S\ T$ **and**
 $bound-inv\ A\ S$

$cdcl_{NOT-inv} S$
shows $\mu A T < \mu A S - n$
using *assms*
proof (*induction n arbitrary: T*)
case 0
then show ?case **using** $cdcl_{NOT-measure}$ **by** *auto*
next
case (*Suc n*) **note** $IH = this(1)[OF - this(3) this(4)]$ **and** $S-T = this(2)$ **and** $b-inv = this(3)$ **and** $c-inv = this(4)$
obtain $U :: 'st$ **where** $S-U: (cdcl_{NOT} \rightsquigarrow (Suc n)) S U$ **and** $U-T: cdcl_{NOT} U T$ **using** $S-T$ **by** *auto*
then have $\mu A U < \mu A S - n$ **using** $IH[of U]$ **by** *simp*
moreover
have $bound-inv A U$
using $S-U b-inv cdcl_{NOT-bound-inv} c-inv$ **by** *blast*
then have $\mu A T < \mu A U$ **using** $cdcl_{NOT-measure}[OF - - U-T] S-U c-inv cdcl_{NOT-cdcl_{NOT-inv}}$
by *auto*
ultimately show ?case **by** *linarith*
qed

lemma *wf-cdcl_{NOT}*:
 $wf \{(T, S). cdcl_{NOT} S T \wedge cdcl_{NOT-inv} S \wedge bound-inv A S\}$ (**is** $wf ?A$)
apply (*rule wfP-if-measure2[of - - μA]*)
using $cdcl_{NOT-comp-n-le}[of 0 - - A]$ **by** *auto*

lemma *rtranclp-cdcl_{NOT-measure}*:
assumes
 $cdcl_{NOT}^{**} S T$ **and**
 $bound-inv A S$ **and**
 $cdcl_{NOT-inv} S$
shows $\mu A T \leq \mu A S$
using *assms*
proof (*induction rule: rtranclp-induct*)
case *base*
then show ?case **by** *auto*
next
case (*step T U*) **note** $IH = this(3)[OF this(4) this(5)]$ **and** $st = this(1)$ **and** $cdcl_{NOT} = this(2)$ **and** $b-inv = this(4)$ **and** $c-inv = this(5)$
have $bound-inv A T$
by (*meson cdcl_{NOT-bound-inv} rtranclp-imp-relpoup st step.prem*s)
moreover have $cdcl_{NOT-inv} T$
using $c-inv rtranclp-cdcl_{NOT-cdcl_{NOT-inv} st}$ **by** *blast*
ultimately have $\mu A U < \mu A T$ **using** $cdcl_{NOT-measure}[OF - - cdcl_{NOT}]$ **by** *auto*
then show ?case **using** IH **by** *linarith*
qed

lemma *cdcl_{NOT-comp-bounded}*:
assumes
 $bound-inv A S$ **and** $cdcl_{NOT-inv} S$ **and** $m \geq 1 + \mu A S$
shows $\neg(cdcl_{NOT} \rightsquigarrow m) S T$
using *assms cdcl_{NOT-comp-n-le}[of m-1 S T A]* **by** *fastforce*

- $f n < m$ ensures that at least one step has been done.

inductive *cdcl_{NOT-restart}* **where**
restart-step: $(cdcl_{NOT} \rightsquigarrow m) S T \implies m \geq f n \implies restart T U$

$\implies \text{cdcl}_{NOT}\text{-restart } (S, n) (U, \text{Suc } n) \mid$
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 $S T \implies \text{cdcl}_{NOT}\text{-raw-restart}^{**} (\text{fst } S) (\text{fst } T)$

proof (*induction rule*: *cdcl_{NOT}-restart.induct*)

case (*restart-step* $m S T n U$)

then have *cdcl_{NOT}^{**} S T* **by** (*meson relpowp-imp-rtrancpl*)

then have *cdcl_{NOT}-raw-restart^{**} S T* **using** *cdcl_{NOT}-raw-restart.intros(1)*

rtrancpl-mono[*of cdcl_{NOT} cdcl_{NOT}-raw-restart*] **by blast**

moreover have *cdcl_{NOT}-raw-restart T U*

using (*restart T U*) *cdcl_{NOT}-raw-restart.intros(2)* **by blast**

ultimately show *?case* **by auto**

next

case (*restart-full* $S T$)

then have *cdcl_{NOT}^{**} S T* **unfolding** *full1-def* **by auto**

then show *?case* **using** *cdcl_{NOT}-raw-restart.intros(1)*

rtrancpl-mono[*of cdcl_{NOT} cdcl_{NOT}-raw-restart*] **by auto**

qed

lemma *cdcl_{NOT}-with-restart-bound-inv*:

assumes

cdcl_{NOT}-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 rtrancpl-unfold fstI full1-def rtrancpl-cdcl_{NOT}-bound-inv*)

by (*metis cdcl_{NOT}-bound-inv cdcl_{NOT}-cdcl_{NOT}-inv cdcl_{NOT}-restart-inv fst-conv*)

lemma *cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*:

assumes

cdcl_{NOT}-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 rtrancpl-cdcl_{NOT}-cdcl_{NOT}-inv*)

done

lemma *rtrancpl-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 rtranclp-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-cls_{NOT} remove-cls_{NOT} :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**
f :: nat \Rightarrow nat **and**
restart :: 'st \Rightarrow 'st \Rightarrow bool **and**
bound-inv :: 'bound \Rightarrow 'st \Rightarrow bool **and**
 μ :: 'bound \Rightarrow 'st \Rightarrow nat **and**
cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool **and**
cdcl_{NOT}-inv :: 'st \Rightarrow bool **and**
 μ -bound :: 'bound \Rightarrow 'st \Rightarrow nat +
assumes
measure-bound: $\bigwedge A T V n. cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A T$
 $\implies cdcl_{NOT}\text{-restart } (T, n) (V, Suc\ n) \implies \mu\ A\ V \leq \mu\text{-bound } A\ T$ **and**
cdcl_{NOT}-raw-restart- μ -bound:
cdcl_{NOT}-restart (T, a) (V, b) $\implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A\ T$
 $\implies \mu\text{-bound } A\ V \leq \mu\text{-bound } A\ T$
begin

lemma *rtranclp-cdcl_{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: rtranclp-induct2*)
apply *simp*
by (*metis cdcl_{NOT}-raw-restart- μ -bound dual-order.trans fst-conv*
rtranclp-cdcl_{NOT}-with-restart-bound-inv rtranclp-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-rtranclp* **apply** *fastforce*
by (*metis full-def full-unfold measure-bound2 prod.inject*)

lemma *rtranclp-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 (*induction rule: rtranclp-induct2*)
apply (*simp add: measure-bound2*)
by (*metis dual-order.trans fst-conv measure-bound2 r-into-rtranclp rtranclp.rtrancl-refl*)

rtrancp-cdcl_{NOT}-with-restart-bound-inv *rtrancp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv*
rtrancp-cdcl_{NOT}-raw-restart-μ-bound)

lemma *wf-cdcl_{NOT}-restart*:

wf $\{(T, S). \text{cdcl}_{NOT}\text{-restart } S \ T \wedge \text{cdcl}_{NOT}\text{-inv } (\text{fst } S)\}$ (**is** *wf* ?*A*)

proof (*rule ccontr*)

assume $\neg ?thesis$

then obtain *g* **where**

g: $\bigwedge i. \text{cdcl}_{NOT}\text{-restart } (g \ i) \ (g \ (\text{Suc } i))$ **and**

cdcl_{NOT}-inv-g: $\bigwedge i. \text{cdcl}_{NOT}\text{-inv } (\text{fst } (g \ i))$

unfolding *wf-iff-no-infinite-down-chain* **by** *fast*

have *snd-g*: $\bigwedge i. \text{snd } (g \ i) = i + \text{snd } (g \ 0)$

apply (*induct-tac i*)

apply *simp*

by (*metis Suc-eq-plus1-left add.commute add.left-commute*
cdcl_{NOT}-with-restart-increasing-number g)

then have *snd-g-0*: $\bigwedge i. i > 0 \implies \text{snd } (g \ i) = i + \text{snd } (g \ 0)$

by *blast*

have *unbounded-f-g*: *unbounded* $(\lambda i. f \ (\text{snd } (g \ i)))$

using *f* **unfolding** *bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g*
not-bounded-nat-exists-larger not-le le-iff-add)

{ fix *i*

have *H*: $\bigwedge T \ \text{Ta} \ m. (\text{cdcl}_{NOT} \rightsquigarrow m) \ T \ \text{Ta} \implies \text{no-step } \text{cdcl}_{NOT} \ T \implies m = 0$

apply (*case-tac m*) **by** *simp* (*meson relpowp-E2*)

have $\exists \ T \ m. (\text{cdcl}_{NOT} \rightsquigarrow m) \ (\text{fst } (g \ i)) \ T \wedge m \geq f \ (\text{snd } (g \ i))$

using *g[of i]* **apply** (*cases rule: cdcl_{NOT}-restart.cases*)

apply *auto*

using *g[of Suc i]* *f-ge-1* **apply** (*cases rule: cdcl_{NOT}-restart.cases*)

apply (*auto simp add: full1-def full-def dest: H dest: trancpD*)

using *H Suc-leI leD* **by** *blast*

} note *H = this*

obtain *A* **where** *bound-inv A* $(\text{fst } (g \ 1))$

using *g[of 0]* *cdcl_{NOT}-inv-g[of 0]* **apply** (*cases rule: cdcl_{NOT}-restart.cases*)

apply (*metis One-nat-def cdcl_{NOT}-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* = $\mu\text{-bound } A \ (\text{fst } (g \ 1)) + 1$

obtain *j* **where**

j: $f \ (\text{snd } (g \ j)) > ?j$ **and** $j > 1$

using *unbounded-f-g not-bounded-nat-exists-larger* **by** *blast*

{

fix *i j*

have *cdcl_{NOT}-with-restart*: $j \geq i \implies \text{cdcl}_{NOT}\text{-restart}^{**} \ (g \ i) \ (g \ j)$

apply (*induction j*)

apply *simp*

by (*metis g le-Suc-eq rtrancp.rtrancp-into-rtrancp rtrancp.rtrancp-refl*)

} note *cdcl_{NOT}-restart = this*

have *cdcl_{NOT}-inv* $(\text{fst } (g \ (\text{Suc } 0)))$

by (*simp add: cdcl_{NOT}-inv-g*)

have *cdcl_{NOT}-restart^{**}* $(\text{fst } (g \ 1), \text{snd } (g \ 1)) \ (\text{fst } (g \ j), \text{snd } (g \ j))$

using $j > 1$ **by** (*simp add: cdcl_{NOT}-restart*)

have $\mu \ A \ (\text{fst } (g \ j)) \leq \mu\text{-bound } A \ (\text{fst } (g \ 1))$

```

apply (rule rtrancpl-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  $\mu A (fst (g j)) \leq ?j$ 
  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))⟩
  rtrancpl-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))  $\leq$  m
  using H[of j] by blast
have ?j < m
  using f-m j Nat.le-trans by linarith

then show False
  using ⟨ $\mu A (fst (g j)) \leq \mu$ -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) >  $\mu$ -bound A (fst S)
shows full1 cdclNOT (fst S) (fst T)
using assms
proof (induction rule: cdclNOT-restart.induct)
case restart-full
then show ?case by auto
next
case (restart-step m S T n U) note st = this(1) and f = this(2) and bound-inv = this(4) and
  cdclNOT-inv = this(5) and  $\mu$  = this(6)
then obtain m' where m: m = Suc m' by (cases m) auto
have  $\mu A S - m' = 0$ 
  using f bound-inv cdclNOT-inv  $\mu$  m rtrancpl-cdclNOT-raw-restart-measure-bound by fastforce
then have False using cdclNOT-comp-n-le[of m' S T A] restart-step unfolding m by simp
then show ?case by fast
qed

lemma rtrancpl-cdclNOT-with-inv-inv-rtrancpl-cdclNOT:
assumes
  inv: cdclNOT-inv S and
  binv: bound-inv A S
shows ( $\lambda S T. cdcl_{NOT} S T \wedge cdcl_{NOT}\text{-inv } S \wedge bound\text{-inv } A S$ )** S T  $\longleftrightarrow$  cdclNOT** S T
  (is ?A** S T  $\longleftrightarrow$  ?B** S T)
apply (rule iffI)
  using rtrancpl-mono[of ?A ?B] apply blast
apply (induction rule: rtrancpl-induct)
  using inv binv apply simp

```

by (metis (mono-tags, lifting) binv inv rtrancpl.simps rtrancpl-cdcl_{NOT}-bound-inv
rtrancpl-cdcl_{NOT}-cdcl_{NOT}-inv)

lemma no-step-cdcl_{NOT}-restart-no-step-cdcl_{NOT}:

assumes

n-s: no-step cdcl_{NOT}-restart *S* **and**

inv: cdcl_{NOT}-inv (fst *S*) **and**

binv: bound-inv *A* (fst *S*)

shows no-step cdcl_{NOT} (fst *S*)

proof (rule ccontr)

assume \neg ?thesis

then obtain *T* **where** *T*: cdcl_{NOT} (fst *S*) *T*

by blast

then obtain *U* **where** *U*: full ($\lambda S T. \text{cdcl}_{\text{NOT}} S T \wedge \text{cdcl}_{\text{NOT}}\text{-inv } S \wedge \text{bound-inv } A S$) *T* *U*

using wf-exists-normal-form-full[OF wf-cdcl_{NOT}, of *A* *T*] **by** auto

moreover have inv-*T*: cdcl_{NOT}-inv *T*

using $\langle \text{cdcl}_{\text{NOT}} (\text{fst } S) T \rangle \text{cdcl}_{\text{NOT}}\text{-inv inv}$ **by** blast

moreover have b-inv-*T*: bound-inv *A* *T*

using $\langle \text{cdcl}_{\text{NOT}} (\text{fst } S) T \rangle \text{binv bound-inv inv}$ **by** blast

ultimately have full cdcl_{NOT} *T* *U*

using rtrancpl-cdcl_{NOT}-with-inv-inv-rtrancpl-cdcl_{NOT} rtrancpl-cdcl_{NOT}-bound-inv

rtrancpl-cdcl_{NOT}-cdcl_{NOT}-inv **unfolding** full-def **by** blast

then have full1 cdcl_{NOT} (fst *S*) *U*

using *T* full-full1 **by** metis

then show False **by** (metis *n-s* prod.collapse restart-full)

qed

end

14.8 Merging backjump and learning

locale cdcl_{NOT}-merge-bj-learn-ops =

dpll-state trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} +

decide-ops trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} +

forget-ops trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} forget-cond +

propagate-ops trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT} propagate-conds

for

trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**

clauses :: 'st \Rightarrow 'v clauses **and**

prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**

tl-trail :: 'st \Rightarrow 'st **and**

add-cl_{NOT} remove-cl_{NOT} :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**

propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool **and**

forget-cond :: 'v clause \Rightarrow 'st \Rightarrow bool +

fixes backjump-l-cond :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow bool

begin

inductive backjump-l **where**

backjump-l: trail *S* = *F'* @ Marked *K* () # *F*

\Rightarrow no-dup (trail *S*)

$\Rightarrow T \sim \text{prepend-trail } (\text{Propagated } L \ ()) (\text{reduce-trail-to}_{\text{NOT}} F (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) S))$

$\Rightarrow C \in \# \text{ clauses } S$

$\Rightarrow \text{trail } S \models_{\text{as}} C \text{Not } C$

$\Rightarrow \text{undefined-lit } F L$

$\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' (lits-of (trail } S))$

$\Rightarrow \text{clauses } S \models_{\text{pm}} C' + \{\#L\# \}$

$\Rightarrow F \models_{\text{as}} C \text{Not } C'$


```

     $\Rightarrow$  backjump-l-cond  $C$   $C'$   $L$   $T$ 
     $\Rightarrow$  backjump-l  $S$   $T$ 
inductive-cases backjump-lE: backjump-l  $S$   $T$ 

inductive cdclNOT-merged-bj-learn :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for  $S$  :: 'st where
cdclNOT-merged-bj-learn-decideNOT: decideNOT  $S$   $S' \Rightarrow$  cdclNOT-merged-bj-learn  $S$   $S'$  |
cdclNOT-merged-bj-learn-propagateNOT: propagateNOT  $S$   $S' \Rightarrow$  cdclNOT-merged-bj-learn  $S$   $S'$  |
cdclNOT-merged-bj-learn-backjump-l: backjump-l  $S$   $S' \Rightarrow$  cdclNOT-merged-bj-learn  $S$   $S'$  |
cdclNOT-merged-bj-learn-forgetNOT: forgetNOT  $S$   $S' \Rightarrow$  cdclNOT-merged-bj-learn  $S$   $S'$ 

lemma cdclNOT-merged-bj-learn-no-dup-inv:
cdclNOT-merged-bj-learn  $S$   $T \Rightarrow$  no-dup (trail  $S$ )  $\Rightarrow$  no-dup (trail  $T$ )
apply (induction rule: cdclNOT-merged-bj-learn.induct)
  using defined-lit-map apply fastforce
  using defined-lit-map apply fastforce
  apply (force simp: defined-lit-map elim!: backjump-lE)[]
using forgetNOT.simps apply auto[1]
done
end

locale cdclNOT-merge-bj-learn-proxy =
cdclNOT-merge-bj-learn-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
propagate-conds forget-conds  $\lambda C$   $C'$   $L'$   $S$ . backjump-l-cond  $C$   $C'$   $L'$   $S$ 
 $\wedge$  distinct-mset ( $C' + \{\#L'\#\}$ )  $\wedge$   $\neg$ 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-clNOT remove-clNOT :: '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-merge-can-jump:
   $\bigwedge S$   $C$   $F'$   $K$   $F$   $L$ .
    inv  $S$ 
     $\Rightarrow$  trail  $S = F' @ \text{Marked } K () \# F$ 
     $\Rightarrow$   $C \in \#$  clauses  $S$ 
     $\Rightarrow$  trail  $S \models_{as} C \text{Not } C$ 
     $\Rightarrow$  undefined-lit  $F$   $L$ 
     $\Rightarrow$  atm-of  $L \in \text{atms-of-msu} (\text{clauses } S) \cup \text{atm-of } ( \text{ lits-of } (F' @ \text{Marked } K () \# F) )$ 
     $\Rightarrow$  clauses  $S \models_{pm} C' + \{\#L'\#\}$ 
     $\Rightarrow$   $F \models_{as} C \text{Not } C'$ 
     $\Rightarrow$   $\neg$ no-step backjump-l  $S$  and
  cdcl-merged-inv:  $\bigwedge S$   $T$ . cdclNOT-merged-bj-learn  $S$   $T \Rightarrow$  inv  $S \Rightarrow$  inv  $T$ 
begin
abbreviation backjump-conds where
backjump-conds  $\equiv \lambda$ -  $C$   $L$  -. distinct-mset ( $C + \{\#L'\#\}$ )  $\wedge$   $\neg$ 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
     $S'$ :  $S' \sim \text{prepend-trail } (\text{Propagated } L \ ()) \text{ (reduce-trail-to}_{NOT} F$ 
      (tl-trail(add-clNOT ( $C' + \{\#L\# \}$ )  $S$ )))
    and
    tr-S: trail  $S = F' @ \text{Marked } K \ () \# F$  and
     $C$ :  $C \in \# \text{ clauses } S$  and
    tr-S-C: trail  $S \models_{as} CNot C$  and
    undef-L: undefined-lit  $F L$  and
    atm-L: atm-of  $L \in \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' lits-of } (\text{trail } S)$  and
    cls-S-C': clauses  $S \models_{pm} C' + \{\#L\# \}$  and
     $F-C'$ :  $F \models_{as} CNot C'$  and
    dist: distinct-mset ( $C' + \{\#L\# \}$ ) and
    not-tauto:  $\neg \text{tautology } (C' + \{\#L\# \})$ 
  by (elim backjump-lE) simp

  have  $\exists S'$ . 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  $H = \text{this}(1)$ 
then show ?case using 1 bj-merge-can-jump by meson
qed

end

locale cdclNOT-merge-bj-learn-proxy2 =
  cdclNOT-merge-bj-learn-proxy 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  $\lambda C \neg. \text{distinct-mset } C \wedge \neg \text{tautology } C$ 

```

forget-conds
 by unfold-locales
 end

locale *cdcl_{NOT}-merge-bj-learn* =
cdcl_{NOT}-merge-bj-learn-proxy2 trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
 propagate-conds inv forget-conds backjump-l-cond
for
 trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**
 clauses :: 'st \Rightarrow 'v clauses **and**
 prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
 tl-trail :: 'st \Rightarrow 'st **and**
 add-cl_{NOT} remove-cl_{NOT}:: 'v clause \Rightarrow 'st \Rightarrow 'st **and**
 propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool **and**
 inv :: 'st \Rightarrow bool **and**
 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: $\bigwedge S T. \text{dpll-bj } S T \Rightarrow \text{inv } S \Rightarrow \text{inv } T$ **and**
 learn-inv: $\bigwedge S T. \text{learn } S T \Rightarrow \text{inv } S \Rightarrow \text{inv } T$
begin

interpretation *cdcl_{NOT}*:
conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
 propagate-conds inv backjump-conds $\lambda C. \text{distinct-mset } C \wedge \neg \text{tautology } C$ forget-conds
apply unfold-locales
apply (simp only: *cdcl_{NOT}.simps*)
using *cdcl_{NOT}-merged-bj-learn-forget_{NOT}* *cdcl-merged-inv* learn-inv
by (auto simp add: *cdcl_{NOT}.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 $\exists C' L. \text{learn } S (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) S)$
 $\wedge \text{backjump } (\text{add-cl}_{\text{NOT}} (C' + \{\#L\# \}) S) T$
 $\wedge \text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-msu } (\text{clauses } S) \cup \text{atm-of ' (lits-of (trail } S))$

proof –

obtain *C F' K F L l C'* **where**
tr-S: trail *S* = *F'* @ Marked *K* () # *F* **and**
T: *T* \sim prepend-trail (Propagated *L l*) (reduce-trail-to_{NOT} *F* (add-cl_{NOT} (*C'* + {#*L*#}) *S*)) **and**
C-clS: *C* \in # clauses *S* **and**
tr-S-CNot-C: trail *S* \models_{as} CNot *C* **and**
undef: undefined-lit *F L* **and**
atm-L: atm-of *L* \in atms-of-msu (clauses *S*) \cup atm-of ' (lits-of (trail *S*)) **and**
clss-C: clauses *S* \models_{pm} *C'* + {#*L*#} **and**
F \models_{as} CNot *C'* **and**
distinct: distinct-mset (*C'* + {#*L*#}) **and**
not-tauto: \neg tautology (*C'* + {#*L*#})
using *bt* *inv* **by** (elim backjump-lE) simp
have *atms-C'*: atms-of *C'* \subseteq atm-of ' (lits-of *F*)
proof –
obtain *ll* :: 'v \Rightarrow ('v literal \Rightarrow 'v) \Rightarrow 'v literal set \Rightarrow 'v literal **where**
 $\forall v f L. v \notin f \text{ ' } L \vee v = f (ll \ v \ f \ L) \wedge ll \ v \ f \ L \in L$
by moura
then show ?thesis **unfolding** *tr-S*
by (metis (no-types) $\langle F \models_{\text{as}}$ CNot *C'* \rangle atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set

$atms\text{-}of\text{-}def\ in\ CNot\text{-}implies\text{-}uminus(2)\ mem\text{-}set\text{-}mset\text{-}iff\ subsetI)$
qed
then have $atms\text{-}of\ (C' + \{\#L\#\}) \subseteq atms\text{-}of\text{-}msu\ (clauses\ S) \cup atm\text{-}of\ (lits\text{-}of\ (trail\ S))$
using $atm\text{-}L\ tr\text{-}S$ **by** $auto$
moreover have $learn: learn\ S\ (add\text{-}cls_{NOT}\ (C' + \{\#L\#\})\ S)$
apply $(rule\ learn.intros)$
apply $(rule\ cls\text{-}C)$
using $atms\text{-}C'\ atm\text{-}L$ **apply** $(fastforce\ simp\ add: tr\text{-}S\ in\text{-}plus\text{-}implies\text{-}atm\text{-}of\text{-}on\text{-}atms\text{-}of\text{-}ms)$
apply $standard$
apply $(rule\ distinct)$
apply $(rule\ not\text{-}tauto)$
apply $simp$
done
moreover have $bj: backjump\ (add\text{-}cls_{NOT}\ (C' + \{\#L\#\})\ S)\ T$
apply $(rule\ backjump.intros)$
using $\langle F \models_{as}\ CNot\ C' \rangle\ C\text{-}cls\text{-}S\ tr\text{-}S\ CNot\text{-}C\ undef\ T\ distinct\ not\text{-}tauto\ n\text{-}d$
by $(auto\ simp: tr\text{-}S\ state\text{-}eq_{NOT}\text{-}def\ simp\ del: state\text{-}simp_{NOT})$
ultimately show $?thesis$ **by** $auto$
qed

lemma $cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}is\text{-}trancpl\text{-}cdcl_{NOT}$:
 $cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\ S\ T \implies inv\ S \implies no\text{-}dup\ (trail\ S) \implies cdcl_{NOT}^{++}\ S\ T$
proof $(induction\ rule: cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn.induct)$
case $(cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}decide_{NOT}\ T)$
then have $cdcl_{NOT}\ S\ T$
using $bj\text{-}decide_{NOT}\ cdcl_{NOT}.simps$ **by** $fastforce$
then show $?case$ **by** $auto$
next
case $(cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}propagate_{NOT}\ T)$
then have $cdcl_{NOT}\ S\ T$
using $bj\text{-}propagate_{NOT}\ cdcl_{NOT}.simps$ **by** $fastforce$
then show $?case$ **by** $auto$
next
case $(cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}forget_{NOT}\ T)$
then have $cdcl_{NOT}\ S\ T$
using $c\text{-}forget_{NOT}$ **by** $blast$
then show $?case$ **by** $auto$
next
case $(cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}backjump\text{-}l\ T)$ **note** $bt = this(1)$ **and** $inv = this(2)$ **and** $n\text{-}d = this(3)$
obtain $C' :: 'v\ literal\ multiset$ **and** $L :: 'v\ literal$ **where**
 $f3: learn\ S\ (add\text{-}cls_{NOT}\ (C' + \{\#L\#\})\ S) \wedge$
 $backjump\ (add\text{-}cls_{NOT}\ (C' + \{\#L\#\})\ S)\ T \wedge$
 $atms\text{-}of\ (C' + \{\#L\#\}) \subseteq atms\text{-}of\text{-}msu\ (clauses\ S) \cup atm\text{-}of\ (lits\text{-}of\ (trail\ S))$
using $n\text{-}d\ backjump\text{-}l\text{-}learn\text{-}backjump[OF\ bt\ inv]$ **by** $blast$
then have $f4: cdcl_{NOT}\ S\ (add\text{-}cls_{NOT}\ (C' + \{\#L\#\})\ S)$
using $n\text{-}d\ c\text{-}learn$ **by** $blast$
have $cdcl_{NOT}\ (add\text{-}cls_{NOT}\ (C' + \{\#L\#\})\ S)\ T$
using $f3\ n\text{-}d\ bj\text{-}backjump\ c\text{-}dpll\text{-}bj$ **by** $blast$
then show $?case$
using $f4$ **by** $(meson\ trancpl.r\text{-}into\text{-}trancpl\ trancpl.trancpl\text{-}into\text{-}trancpl)$
qed

lemma $rtrancpl\text{-}cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn\text{-}is\text{-}rtrancpl\text{-}cdcl_{NOT}\text{-}and\text{-}inv$:
 $cdcl_{NOT}\text{-}merged\text{-}bj\text{-}learn^{**}\ S\ T \implies inv\ S \implies no\text{-}dup\ (trail\ S) \implies cdcl_{NOT}^{**}\ S\ T \wedge inv\ T$

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**
 $atm-clss: atms-of-msu\ (clauses\ S) \subseteq atms-of-ms\ A$ **and**
 $atm-trail: atm-of\ ' lits-of\ (trail\ S) \subseteq atms-of-ms\ A$ **and**
 $n-d: no-dup\ (trail\ S)$ **and**
 $fin-A: finite\ A$
shows μ_{CDCL}' -merged $A\ T < \mu_{CDCL}'$ -merged $A\ S$
using *assms(1)*

proof *induction*
case ($cdcl_{NOT}$ -merged-bj-learn-decide $_{NOT}\ T$)
have $clauses\ S = clauses\ T$
using $cdcl_{NOT}$ -merged-bj-learn-decide $_{NOT}$.*hyps* **by** *auto*
moreover have
 $(2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $- \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ T)$
 $< (2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $- \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ S)$
apply (*rule* $dp11$ -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 *$\mu_{CDCL'}$ -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 *⟨forget_{NOT} S T⟩ by (metis card-Diff1-less*
cdcl_{NOT}-merged-bj-learn-forget_{NOT}.hyps clauses-remove-cls_{NOT} finite-set-mset forget_{NOT}E
mem-set-mset-iff order-refl set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses)
moreover
have *trail S = trail T*
using *⟨forget_{NOT} S T⟩ by (auto elim: forget_{NOT}E)*
then have
 $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $= (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$
by *auto*
ultimately show *?case*
unfolding *$\mu_{CDCL'}$ -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-cls_{NOT} apply simp apply fast*
using *atm-trail n-d apply simp*
apply (*simp add: n-d*)
using *fin-A apply simp*
done
then have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T))$

$< ((2 + \text{card}(\text{atms-of-ms } A)) \wedge (1 + \text{card}(\text{atms-of-ms } A)))$
 $- \mu_C (1 + \text{card}(\text{atms-of-ms } A)) (2 + \text{card}(\text{atms-of-ms } A)) (\text{trail-weight } S))$
using $n\text{-d}$ **by** *auto*
then show $?case$
using card-T-S **unfolding** $\mu_{CDCL}'\text{-merged-def}$ $\mu_C'\text{-def}$ **by** *linarith*
qed

lemma $\text{wf-cdcl}_{NOT}\text{-merged-bj-learn}$:

assumes

fin-A : *finite* A

shows $\text{wf } \{(T, S)\}$.

$(\text{inv } S \wedge \text{atms-of-msu}(\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup}(\text{trail } S))$

$\wedge \text{cdcl}_{NOT}\text{-merged-bj-learn } S \ T\}$

apply (*rule* $\text{wfP-if-measure}[\text{of } - - \mu_{CDCL}'\text{-merged } A]$)

using $\text{cdcl}_{NOT}\text{-decreasing-measure}'$ fin-A **by** *simp*

lemma $\text{trancpl-cdcl}_{NOT}\text{-cdcl}_{NOT}\text{-trancpl}$:

assumes

$\text{cdcl}_{NOT}\text{-merged-bj-learn}^{++} \ S \ T$ **and**

inv : $\text{inv } S$ **and**

atm-clss : $\text{atms-of-msu}(\text{clauses } S) \subseteq \text{atms-of-ms } A$ **and**

atm-trail : $\text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d}$: $\text{no-dup}(\text{trail } S)$ **and**

$\text{fin-A}[\text{simp}]$: *finite* A

shows $(T, S) \in \{(T, S)\}$.

$(\text{inv } S \wedge \text{atms-of-msu}(\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup}(\text{trail } S))$

$\wedge \text{cdcl}_{NOT}\text{-merged-bj-learn } S \ T\}^+ (\text{is } - \in ?P^+)$

using *assms*(1)

proof (*induction rule*: trancpl-induct)

case *base*

then show $?case$ **using** $n\text{-d}$ atm-clss atm-trail inv **by** *auto*

next

case (*step* $T \ U$) **note** $st = \text{this}(1)$ **and** $\text{cdcl}_{NOT} = \text{this}(2)$ **and** $IH = \text{this}(3)$

have $\text{cdcl}_{NOT}^{**} \ S \ T$

apply (*rule* $\text{rtrancpl-cdcl}_{NOT}\text{-merged-bj-learn-is-rtrancpl-cdcl}_{NOT}$)

using st cdcl_{NOT} inv $n\text{-d}$ atm-clss atm-trail inv **by** *auto*

have $\text{inv } T$

apply (*rule* $\text{rtrancpl-cdcl}_{NOT}\text{-merged-bj-learn-inv}$)

using inv st cdcl_{NOT} $n\text{-d}$ atm-clss atm-trail inv **by** *auto*

moreover have $\text{atms-of-msu}(\text{clauses } T) \subseteq \text{atms-of-ms } A$

using $\text{cdcl}_{NOT}.\text{rtrancpl-cdcl}_{NOT}\text{-trail-clauses-bound}[OF \ \langle \text{cdcl}_{NOT}^{**} \ S \ T \rangle \ \text{inv } n\text{-d } \text{atm-clss } \text{atm-trail}]$
by *fast*

moreover have $\text{atm-of } ' (\text{lits-of } (\text{trail } T)) \subseteq \text{atms-of-ms } A$

using $\text{cdcl}_{NOT}.\text{rtrancpl-cdcl}_{NOT}\text{-trail-clauses-bound}[OF \ \langle \text{cdcl}_{NOT}^{**} \ S \ T \rangle \ \text{inv } n\text{-d } \text{atm-clss } \text{atm-trail}]$
by *fast*

moreover have $\text{no-dup}(\text{trail } T)$

using $\text{cdcl}_{NOT}.\text{rtrancpl-cdcl}_{NOT}\text{-no-dup}[OF \ \langle \text{cdcl}_{NOT}^{**} \ S \ T \rangle \ \text{inv } n\text{-d}]$ **by** *fast*

ultimately have $(U, T) \in ?P$

using cdcl_{NOT} **by** *auto*

then show $?case$ **using** IH **by** (*simp add*: $\text{trancpl-into-trancpl2}$)

qed

lemma $\text{wf-trancpl-cdcl}_{NOT}\text{-merged-bj-learn}$:

```

assumes finite A
shows wf {(T, S).
  (inv S  $\wedge$  atms-of-msu (clauses S)  $\subseteq$  atms-of-ms A  $\wedge$  atm-of ' lits-of (trail S)  $\subseteq$  atms-of-ms A
   $\wedge$  no-dup (trail S))
   $\wedge$  cdclNOT-merged-bj-learn++ S T
apply (rule wf-subset)
apply (rule wf-trancl[OF wf-cdclNOT-merged-bj-learn])
using assms apply simp
using tranclp-cdclNOT-cdclNOT-tranclp[OF - - - - (finite A)] by auto

lemma backjump-no-step-backjump-l:
  backjump S T  $\implies$  inv S  $\implies$   $\neg$ no-step backjump-l S
apply (elim backjumpE)
apply (rule bj-merge-can-jump)
  apply auto[7]
by blast

lemma cdclNOT-merged-bj-learn-final-state:
  fixes A :: 'v literal multiset set and S T :: 'st
assumes
  n-s: no-step cdclNOT-merged-bj-learn S and
  atms-S: atms-of-msu (clauses S)  $\subseteq$  atms-of-ms A and
  atms-trail: atm-of ' lits-of (trail S)  $\subseteq$  atms-of-ms A and
  n-d: no-dup (trail S) and
  finite A and
  inv: inv S and
  decomp: all-decomposition-implies-m (clauses S) (get-all-marked-decomposition (trail S))
shows unsatisfiable (set-mset (clauses S))
   $\vee$  (trail S  $\models_{asm}$  clauses S  $\wedge$  satisfiable (set-mset (clauses S)))

proof -
let ?N = set-mset (clauses S)
let ?M = trail S
consider
  (sat) satisfiable ?N and ?M  $\models_{as}$  ?N
  | (sat') satisfiable ?N and  $\neg$  ?M  $\models_{as}$  ?N
  | (unsat) unsatisfiable ?N
by auto
then show ?thesis
proof cases
  case sat' note sat = this(1) and M = this(2)
obtain C where C  $\in$  ?N and  $\neg$ ?M  $\models_a$  C using M unfolding true-annots-def by auto
obtain I :: 'v literal set where
  I  $\models_s$  ?N and
  cons: consistent-interp I and
  tot: total-over-m I ?N and
  atm-I-N: atm-of 'I  $\subseteq$  atms-of-ms ?N
using sat unfolding satisfiable-def-min by auto
let ?I = I  $\cup$  {P | P. P  $\in$  lits-of ?M  $\wedge$  atm-of P  $\notin$  atm-of ' I}
let ?O = { {#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set ?M  $\wedge$  atm-of (lit-of L)  $\notin$  atms-of-ms ?N }
have cons-I': consistent-interp ?I
using cons using (no-dup ?M) unfolding consistent-interp-def
by (auto simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def
  dest!: no-dup-cannot-not-lit-and-uminus)
have tot-I': total-over-m ?I (?N  $\cup$  unmark ?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 ∪ unmark ?M)
    ⟷ total-over-m I (?N ∪ unmark ?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 unmark ?M
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] by auto

  then have ?I ⊨s unmark ?M
    using cons-I' I'-N tot-I' ⟨?I ⊨s ?N ∪ ?O⟩ unfolding ∅ true-clss-clss-def by blast
  then have lits-of ?M ⊆ ?I
    unfolding true-clss-def lits-of-def by auto
  then have ?M ⊨as ?N
    using I'-N ⟨C ∈ ?N⟩ ⟨¬ ?M ⊨a C⟩ cons-I' atms-N-M
    by (meson ⟨trail S ⊨as CNot C⟩ consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
      true-annots-def true-clss-mono-set-mset-l true-clss-def)
  then show False using M by fast
qed
from List.split-list-first-propE[OF this] obtain K :: 'v literal and d :: unit and
  F F' :: ('v, unit, unit) marked-lit list where
  M-K: ?M = F' @ Marked K () # F and

```

```

nm:  $\forall f \in \text{set } F'. \neg \text{is-marked } f$ 
unfolding is-marked-def by (metis (full-types) old.unit.exhaust)
let ?K = Marked K () :: ('v, unit, unit) marked-lit
have ?K  $\in$  set ?M
unfolding M-K by auto
let ?C = image-mset lit-of {#L  $\in$  #mset ?M. is-marked L  $\wedge$  L  $\neq$  ?K#} :: 'v literal multiset
let ?C' = set-mset (image-mset ( $\lambda L :: 'v$  literal. {#L#}) (?C + {#lit-of ?K#}))
have ?N  $\cup$  {#{#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set ?M}  $\models_{ps}$  unmark ?M
using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have C': ?C' = {#{#lit-of L#} | L. is-marked L  $\wedge$  L  $\in$  set ?M}
unfolding M-K apply standard
apply force
using IntI by auto
ultimately have N-C-M: ?N  $\cup$  ?C'  $\models_{ps}$  unmark ?M
by auto
have N-M-False: ?N  $\cup$  ( $\lambda L. \{ \# \text{lit-of } L \# \}$ ) ' (set ?M)  $\models_{ps}$  {#{#}}
using M < ?M  $\models_{as}$  CNot C' < C  $\in$  ?N unfolding true-clss-clss-def true-annots-def Ball-def
true-annot-def by (metis consistent-CNot-not sup.orderE sup-commute true-clss-def
true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have undefined-lit F K using <no-dup ?M> unfolding M-K by (simp add: defined-lit-map)
moreover
have ?N  $\cup$  ?C'  $\models_{ps}$  {#{#}}
proof -
have A: ?N  $\cup$  ?C'  $\cup$  unmark ?M =
?N  $\cup$  unmark ?M
unfolding M-K by auto
show ?thesis
using true-clss-clss-left-right[OF N-C-M, of {#{#}}] N-M-False unfolding A by auto
qed
have ?N  $\models_p$  image-mset uminus ?C + {#-K#}
unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
proof (intro allI impI)
fix I
assume
tot: total-over-set I (atms-of-ms (?N  $\cup$  {image-mset uminus ?C + {#-K#}})) and
cons: consistent-interp I and
I  $\models_s$  ?N
have (K  $\in$  I  $\wedge$  -K  $\notin$  I)  $\vee$  (-K  $\in$  I  $\wedge$  K  $\notin$  I)
using cons tot unfolding consistent-interp-def by (cases K) auto
have tot': total-over-set I
(atm-of 'lit-of ' (set ?M  $\cap$  {L. is-marked L  $\wedge$  L  $\neq$  Marked K ()}))
using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
{ fix x :: ('v, unit, unit) marked-lit
assume
a3: lit-of x  $\notin$  I and
a1: x  $\in$  set ?M and
a4: is-marked x and
a5: x  $\neq$  Marked K ()
then have Pos (atm-of (lit-of x))  $\in$  I  $\vee$  Neg (atm-of (lit-of x))  $\in$  I
using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast
moreover have f6: Neg (atm-of (lit-of x)) = - Pos (atm-of (lit-of x))
by simp
ultimately have - lit-of x  $\in$  I
using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
```

```

    literal.sel(1))
  } note H = 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-merge-can-jump[of S F' K F C -K
      image-mset uminus (image-mset lit-of  $\{\# L : \# \text{ mset } ?M. \text{ is-marked } L \wedge L \neq \text{Marked } K ()\# \}$ )
       $\langle C \in ?N \rangle$  n-s  $\langle ?M \models_{as} CNot C \rangle$  bj-backjump inv unfolding M-K
      by (auto simp: cdclNOT-merged-bj-learn.simps)
    then show ?thesis by fast
  qed auto
qed

lemma full-cdclNOT-merged-bj-learn-final-state:
  fixes A :: 'v literal multiset set and S T :: 'st
  assumes
    full: full cdclNOT-merged-bj-learn S T and
    atms-S: atms-of-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 cdclNOT-merged-bj-learn-final-state[of T A]  $\langle$ finite A $\rangle$  n-s by fast
qed
end

```

14.8.1 Instantiations

locale *cdcl_{NOT}-with-backtrack-and-restarts* =
conflict-driven-clause-learning-learning-before-backjump-only-distinct-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 \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**
 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
 +
fixes f :: nat \Rightarrow nat
assumes
 unbounded: unbounded f **and** f-ge-1: $\bigwedge n. n \geq 1 \Rightarrow f\ n \geq 1$ **and**
 inv-restart: $\bigwedge S\ T. inv\ S \Rightarrow T \sim \text{reduce-trail-to}_{NOT} (\llbracket :: 'a\ \text{list} \rrbracket S \Rightarrow inv\ T$
begin

lemma *bound-inv-inv*:
assumes
 inv S **and**
 n-d: no-dup (trail S) **and**
 atms-clss-S-A: atms-of-msu (clauses S) \subseteq atms-of-ms A **and**
 atms-trail-S-A: atm-of ' lits-of (trail S) \subseteq atms-of-ms A **and**
 finite A **and**
 cdcl_{NOT}: cdcl_{NOT} S T
shows
 atms-of-msu (clauses T) \subseteq atms-of-ms A **and**
 atm-of ' lits-of (trail T) \subseteq atms-of-ms A **and**
 finite A
proof –
have cdcl_{NOT} S T
using ⟨inv S⟩ cdcl_{NOT} **by** linarith
then have atms-of-msu (clauses T) \subseteq atms-of-msu (clauses S) \cup 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) \subseteq atms-of-ms A
using atms-clss-S-A atms-trail-S-A **by** blast
next
show atm-of ' lits-of (trail T) \subseteq atms-of-ms A
by (meson ⟨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* $\lambda S\ T. T \sim \text{reduce-trail-to}_{NOT} (\llbracket :: 'a\ \text{list} \rrbracket S\ \text{cdcl}_{NOT}\ f$
 $\lambda A\ S. \text{atms-of-msu}(\text{clauses } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of}(\text{trail } S) \subseteq \text{atms-of-ms } A \wedge$
 finite A
 $\mu_{CDCL}' \lambda S. inv\ S \wedge \text{no-dup}(\text{trail } S)$

μ_{CDCL}' -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 rtrancpl-cdcl_{NOT}- μ_{CDCL}' -bound; simp*)
 apply (*rule rtrancpl- 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 (λ - - S T . backjump S T)
(λC S . distinct-mset $C \wedge \neg$ tautology $C \wedge$ learn-restrictions C S
 $\wedge (\exists F$ K 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))
(λC S . $\neg (\exists F' F$ K L . trail $S = F' @$ Marked K $() \# F \wedge F \models_{as} CNot$ ($C - \{\#L\#$)))
 \wedge forget-restrictions C S)

lemma *cdcl_{NOT}-with-restart- μ_{CDCL}' -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 $\mu_{CDCL}' A V \leq \mu_{CDCL}'$ -bound $A T$

using *cdcl_{NOT}-inv bound-inv*

proof (*induction rule: cdcl_{NOT}-with-restart-induct[OF cdcl_{NOT}]*)

case ($1 m S T n U$) **note** $U = this(3)$

show *?case*

apply (*rule rtrancpl-cdcl_{NOT}- μ_{CDCL}' -bound-reduce-trail-to_{NOT}[of S T]*)
 using $\langle (cdcl_{NOT} \rightsquigarrow m) S T \rangle$ **apply** (*fastforce dest!: relpowp-imp-rtrancpl*)
 using 1 **by** *auto*

next

case ($2 S T n$) **note** $full = this(2)$

show *?case*

apply (*rule rtrancpl-cdcl_{NOT}- μ_{CDCL}' -bound*)
 using $full$ 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:
 $atms-of-msu \text{ (clauses } T) \subseteq atms-of-ms \text{ } A$
 $atm-of \text{ ' lits-of (trail } T) \subseteq atms-of-ms \text{ } A$
 $finite \text{ } A$
shows $\mu_{CDCL}'\text{-bound } A \text{ } V \leq \mu_{CDCL}'\text{-bound } A \text{ } T$
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 $\mu_{CDCL}'\text{-bound } A \text{ } T \leq \mu_{CDCL}'\text{-bound } A \text{ } S$
apply (*rule rtrancpl- μ_{CDCL}' -bound-decreasing*)
using $\langle (cdcl_{NOT} \rightsquigarrow m) \text{ } S \text{ } T \rangle$ **apply** (*fastforce dest: relpowp-imp-rtrancpl*)
using 1 **by** *auto*
then show ?*case* **using** *U unfolding μ_{CDCL}' -bound-def* **by** *auto*
next
case (2 *S T n*) **note** $full = this(2)$
show ?*case*
apply (*rule rtrancpl- μ_{CDCL}' -bound-decreasing*)
using *full 2 unfolding full1-def* **by** *force+*
qed

sublocale *cdcl_{NOT}-increasing-restarts - - - - f*
 $\lambda S \text{ } T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}) \text{ } S$
 $\lambda A \text{ } S. atms-of-msu \text{ (clauses } S) \subseteq atms-of-ms \text{ } A$
 $\wedge atm-of \text{ ' lits-of (trail } S) \subseteq atms-of-ms \text{ } A \wedge finite \text{ } A$
 $\mu_{CDCL}' \text{ } cdcl_{NOT}$
 $\lambda S. inv \text{ } S \wedge no\text{-dup (trail } S)$
 $\mu_{CDCL}'\text{-bound}$
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*
 $inv \text{ (fst } S) \text{ and}$
 $no\text{-dup (trail (fst } S))$
 $all\text{-decomposition-implies-}m \text{ (clauses (fst } S)) \text{ (get-all-marked-decomposition (trail (fst } S)))}$
shows
 $all\text{-decomposition-implies-}m \text{ (clauses (fst } T)) \text{ (get-all-marked-decomposition (trail (fst } T)))}$
using *assms apply (induction)*
using *rtrancpl-cdcl_{NOT}-all-decomposition-implies by (auto dest!: trancpl-into-rtrancpl*
 $simp: full1\text{-def})$

lemma *rtrancpl-cdcl_{NOT}-restart-all-decomposition-implies:*
assumes *cdcl_{NOT}-restart** S T and*
 $inv: inv \text{ (fst } S) \text{ and}$
 $n\text{-d: no-dup (trail (fst } S)) \text{ and}$
 $decomp:$
 $all\text{-decomposition-implies-}m \text{ (clauses (fst } S)) \text{ (get-all-marked-decomposition (trail (fst } S)))}$
shows
 $all\text{-decomposition-implies-}m \text{ (clauses (fst } T)) \text{ (get-all-marked-decomposition (trail (fst } T)))}$
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 ⊨sextm clauses (fst S) ⟷ I ⊨sextm clauses(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 ⊨sextm clauses (fst S) ⟷ I ⊨sextm clauses(fst T)
  using st
proof (induction)
  case base
  then show ?case by simp
next
  case (step T U) note st = this(1) and r = this(2) and IH = this(3)
  have inv (fst T)
    using rtrancpl-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 cdclNOT-restart (S, n) (T, m) 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
    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*: $\bigwedge I. I \models_{\text{sextm}}$ *clauses S* \longleftrightarrow $I \models_{\text{sextm}}$ *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-cls_{NOT} remove-cls_{NOT} +
propagate-ops trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT} propagate-conds +
backjumping-ops trail clauses prepend-trail tl-trail add-cls_{NOT} remove-cls_{NOT} λ - - - -. True
for
trail :: '*st* \Rightarrow ('*v*, *unit*, *unit*) *marked-lits* **and**
clauses :: '*st* \Rightarrow '*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, 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 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 cdclNOT:
  conflict-driven-clause-learning-ops trail clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds inv backjump-conds (λC -. distinct-mset C ∧ ¬ tautology C) forget-conds
  by unfold-locales

```

```

interpretation cdclNOT:

```

conflict-driven-clause-learning trail clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{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_s $A = \{\#C \in \# A. \text{tautology } C \vee \neg \text{distinct-mset } C\}$

lemma simple-cl_s-or-not-simplified-cl_s:

assumes atms-of-msu (clauses S) \subseteq atms-of-ms A **and**

$x \in \# \text{clauses } S$ **and** finite A

shows $x \in \text{simple-clss (atms-of-ms } A) \vee x \in \# \text{not-simplified-cl}_s (\text{clauses } S)$

proof –

consider

(*simpl*) $\neg \text{tautology } x$ **and** *distinct-mset* x

| (*n-simp*) $\text{tautology } x \vee \neg \text{distinct-mset } x$

by auto

then show ?thesis

proof cases

case *simpl*

then have $x \in \text{simple-clss (atms-of-ms } A)$

by (meson assms atms-of-atms-of-ms-mono atms-of-ms-finite simple-clss-mono

distinct-mset-not-tautology-implies-in-simple-clss finite-subset

mem-set-mset-iff subsetCE)

then show ?thesis **by** blast

next

case *n-simp*

then have $x \in \# \text{not-simplified-cl}_s (\text{clauses } S)$

using $\langle x \in \# \text{clauses } S \rangle$ **unfolding** not-simplified-cl_s-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**

$n\text{-d}$: no-dup (trail S) **and**

fin-A[*simp*]: finite A

shows $\text{set-mset (clauses } T) \subseteq \text{set-mset (not-simplified-cl}_s (\text{clauses } S))$

$\cup \text{simple-clss (atms-of-ms } A)$

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!: simple-clss-or-not-simplified-cl_s)

next

case cdcl_{NOT}-merged-bj-learn-propagate_{NOT}

then show ?case **using** dpll-bj-clauses **by** (force dest!: simple-clss-or-not-simplified-cl_s)

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!: forget_{NOT} E dest: simple-clss-or-not-simplified-cl_s)

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 $\langle backjump-l S T \rangle inv\ cdcl_{NOT}$ -merged-bj-learn.simps $n-d$ **by** *blast+*
have $atm-of\ \langle lits-of\ (trail\ T) \rangle \subseteq atm-of-ms\ A$
using $cdcl_{NOT}.rtrancpl-cdcl_{NOT}$ -trail-clauses-bound[$OF\ \langle cdcl_{NOT}^{**} S T \rangle$] $inv\ atms-trail\ atms-clss\ n-d$ **by** *auto*
have $atms-of-msu\ (clauses\ T) \subseteq atm-of-ms\ A$
using $cdcl_{NOT}.rtrancpl-cdcl_{NOT}$ -trail-clauses-bound[$OF\ \langle cdcl_{NOT}^{**} S T \rangle inv\ n-d\ atms-clss\ atms-trail$]
by *fast*
moreover have $no-dup\ (trail\ T)$
using $cdcl_{NOT}.rtrancpl-cdcl_{NOT}$ -no-dup[$OF\ \langle cdcl_{NOT}^{**} S T \rangle inv\ n-d$] **by** *fast*

obtain $F' K F L l C' C$ **where**
 $tr-S: trail\ S = F' @\ Marked\ K\ ()\ \# F$ **and**
 $T: T \sim prepend-trail\ (Propagated\ L\ l)\ (reduce-trail-to_{NOT}\ F\ (add-cl_{NOT}\ (C' + \{\#L\#\})\ S))$ **and**
 $C \in \# clauses\ S$ **and**
 $trail\ S \models_{as} CNot\ C$ **and**
 $undef: undefined-lit\ F\ L$ **and**
 $atm-of\ L = atm-of\ K \vee atm-of\ L \in atm-of-msu\ (clauses\ S)$
 $\vee atm-of\ L \in atm-of\ \langle lits-of\ F' \cup lits-of\ F \rangle$ **and**
 $clauses\ S \models_{pm} C' + \{\#L\#\}$ **and**
 $F \models_{as} CNot\ C'$ **and**
 $dist: distinct-mset\ (C' + \{\#L\#\})$ **and**
 $tauto: \neg tautology\ (C' + \{\#L\#\})$ **and**
 $backjump-l-cond\ C\ C'\ L\ T$
using $\langle backjump-l S T \rangle$ **apply** (*induction rule: backjump-l.induct*) **by** *auto*

have $atms-of\ C' \subseteq atm-of\ \langle lits-of\ F \rangle$
using $\langle F \models_{as} CNot\ C' \rangle$ **by** (*simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atm-of-def image-subset-iff in-CNot-implies-uminus(2)*)
then have $atms-of\ (C' + \{\#L\#\}) \subseteq atm-of-ms\ A$
using $T\ \langle atm-of\ \langle lits-of\ (trail\ T) \rangle \subseteq atm-of-ms\ A \rangle tr-S\ undef\ n-d$ **by** *auto*
then have $simple-clss\ (atms-of\ (C' + \{\#L\#\})) \subseteq simple-clss\ (atms-of-ms\ A)$
apply – **by** (*rule simple-clss-mono*) (*simp-all*)
then have $C' + \{\#L\#\} \in simple-clss\ (atms-of-ms\ A)$
using $distinct-mset-not-tautology-implies-in-simple-clss[OF\ dist\ tauto]$
by *auto*
then show *?case*
using $T\ inv\ atms-clss\ undef\ tr-S\ n-d$
by (*force dest!: 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 $(not-simplified-cls\ (clauses\ T)) \subseteq \# (not-simplified-cls\ (clauses\ S))$
using *assms* **apply** *induction*
prefer 4
unfolding *not-simplified-cls-def* **apply** (*auto elim!: backjump-lE forget_{NOT}E*)[3]
by (*elim backjump-lE*) *auto*

lemma $rtrancpl-cdcl_{NOT}$ -merged-bj-learn-not-simplified-decreasing:
assumes $cdcl_{NOT}$ -merged-bj-learn ** $S\ T$

shows $(\text{not-simplified-cls } (\text{clauses } T)) \subseteq \# (\text{not-simplified-cls } (\text{clauses } S))$
using *assms apply induction*
apply *simp*
by $(\text{drule } \text{cdcl}_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}) \text{ auto}$

lemma *rtranclp-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 *simple-clss* (*atms-of-ms A*)

using *assms(1–5)*

proof *induction*

case *base*

then show ?*case* **by** $(\text{auto dest!}: \text{simple-clss-or-not-simplified-cls})$

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 rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}-and-inv st n-d* **by** *blast*

have *inv T*

using *inv rtranclp-cdcl_{NOT}-merged-bj-learn-inv st n-d* **by** *blast*

moreover

have *atms-of-msu* (*clauses T*) \subseteq *atms-of-ms A* **and**

atm-of ‘*lits-of* (*trail T*) \subseteq *atms-of-ms A*

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-trail-clauses-bound*[*OF st'*] *inv atms-clss-S atms-trail-S n-d*

by *blast+*

moreover moreover have *no-dup* (*trail T*)

using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-no-dup*[*OF* ‘*cdcl_{NOT}*** *S T*’ *inv n-d*] **by** *fast*

ultimately have *set-mset* (*clauses U*)

\subseteq *set-mset* (*not-simplified-cls* (*clauses T*)) \cup *simple-clss* (*atms-of-ms A*)

using *cdcl_{NOT} finite cdcl_{NOT}-merged-bj-learn-clauses-bound*

by $(\text{auto intro!}: \text{cdcl}_{NOT}\text{-merged-bj-learn-clauses-bound})$

moreover have *set-mset* (*not-simplified-cls* (*clauses T*))

\subseteq *set-mset* (*not-simplified-cls* (*clauses S*))

using *rtranclp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*[*OF st*] **by** *auto*

ultimately show ?*case* **using** *IH inv atms-clss-S*

by $(\text{auto dest!}: \text{simple-clss-or-not-simplified-cls})$

qed

abbreviation μ_{CDCL}' -*bound* **where**

μ_{CDCL}' -*bound A T* == $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * 2$

$+ \text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses } T)))$

$+ 3 \wedge \text{card } (\text{atms-of-ms } A)$

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-clauses-bound-card*:

assumes

*cdcl_{NOT}-merged-bj-learn*** *S T* **and**

inv S **and**

atms-of-msu (*clauses S*) \subseteq *atms-of-ms A* **and**

atm-of ‘(*lits-of* (*trail S*)) \subseteq *atms-of-ms A* **and**

$n\text{-d}$: $\text{no-dup } (\text{trail } S)$ **and**
 finite : $\text{finite } A$
shows $\mu_{CDCL}'\text{-merged } A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$
proof –
have $\text{set-mset } (\text{clauses } T) \subseteq \text{set-mset } (\text{not-simplified-cls}(\text{clauses } S))$
 $\cup \text{simple-clss } (\text{atms-of-ms } A)$
using $\text{rtranclp-cdcl}_{NOT}\text{-merged-bj-learn-clauses-bound}[OF \ \text{assms}]$.
moreover have $\text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses } S)))$
 $\cup \text{simple-clss } (\text{atms-of-ms } A))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses } S))) + 3 \wedge \text{card } (\text{atms-of-ms } A)$
by ($\text{meson } \text{Nat.le-trans } \text{atms-of-ms-finite } \text{simple-clss-card } \text{card-Un-le } \text{finite}$
 $\text{nat-add-left-cancel-le}$)
ultimately have $\text{card } (\text{set-mset } (\text{clauses } T))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses } S))) + 3 \wedge \text{card } (\text{atms-of-ms } A)$
by ($\text{meson } \text{Nat.le-trans } \text{atms-of-ms-finite } \text{simple-clss-finite } \text{card-mono}$
 $\text{finite-UnI } \text{finite-set-mset } \text{local.finite}$)
moreover have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A \ T) * 2$
 $\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) * 2$
by auto
ultimately show $?thesis \text{ unfolding } \mu_{CDCL}'\text{-merged-def}$ **by** auto
qed

sublocale $\text{cdcl}_{NOT}\text{-increasing-restarts-ops } \lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} ([::'a \ \text{list}] \ S)$
 $\text{cdcl}_{NOT}\text{-merged-bj-learn } f$
 $\lambda A \ S. \ \text{atms-of-msu } (\text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of } ' \text{ lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}'\text{-merged}$
 $\lambda S. \ \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
apply unfold-locales
using $\text{unbounded apply simp}$
using $f\text{-ge-1 apply force}$
apply ($\text{blast dest!} : \text{cdcl}_{NOT}\text{-merged-bj-learn-is-tranclp-cdcl}_{NOT} \ \text{tranclp-into-rtranclp}$
 $\text{cdcl}_{NOT}.\text{rtranclp-cdcl}_{NOT}\text{-trail-clauses-bound}$)
apply ($\text{simp add} : \text{cdcl}_{NOT}\text{-decreasing-measure}'$)
using $\text{rtranclp-cdcl}_{NOT}\text{-merged-bj-learn-clauses-bound-card}$ **apply** blast
apply ($\text{drule } \text{rtranclp-cdcl}_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}$)
apply ($\text{auto dest!} : \text{simp} : \text{card-mono set-mset-mono}$) []
apply simp
apply auto []
using $\text{cdcl}_{NOT}\text{-merged-bj-learn-no-dup-inv } \text{cdcl-merged-inv}$ **apply** blast
apply ($\text{auto simp} : \text{inv-restart}$) []
done

lemma $\text{cdcl}_{NOT}\text{-restart-}\mu_{CDCL}'\text{-merged-le-}\mu_{CDCL}'\text{-bound}$:
assumes
 $\text{cdcl}_{NOT}\text{-restart } T \ V$
 $\text{inv } (\text{fst } T)$ **and**
 $\text{no-dup } (\text{trail } (\text{fst } T))$ **and**
 $\text{atms-of-msu } (\text{clauses } (\text{fst } T)) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-of } ' \text{ lits-of } (\text{trail } (\text{fst } T)) \subseteq \text{atms-of-ms } A$ **and**
 $\text{finite } A$
shows $\mu_{CDCL}'\text{-merged } A \ (\text{fst } V) \leq \mu_{CDCL}'\text{-bound } A \ (\text{fst } T)$
using assms
proof induction

```

case (restart-full S T n)
show ?case
  unfolding fst-conv
  apply (rule rtrancpl-cdclNOT-merged-bj-learn-clauses-bound-card)
  using restart-full unfolding full1-def by (force dest!: trancpl-into-rtrancpl)+
next
case (restart-step m S T n U) note st = this(1) and U = this(3) and inv = this(4) and
  n-d = this(5) and atms-clss = this(6) and atms-trail = this(7) and finite = this(8)
then have st': cdclNOT-merged-bj-learn** S T
  by (blast dest: relpowp-imp-rtrancpl)
then have st'': cdclNOT** S T
  using inv n-d apply - by (rule rtrancpl-cdclNOT-merged-bj-learn-is-rtrancpl-cdclNOT) auto
have inv T
  apply (rule rtrancpl-cdclNOT-merged-bj-learn-inv)
  using inv st' n-d by auto
then have inv U
  using U by (auto simp: inv-restart)
have atms-of-msu (clauses T)  $\subseteq$  atms-of-ms A
  using cdclNOT.rtrancpl-cdclNOT-trail-clauses-bound[OF st'] inv atms-clss atms-trail n-d
  by simp
then have atms-of-msu (clauses U)  $\subseteq$  atms-of-ms A
  using U by simp
have not-simplified-cls (clauses U)  $\subseteq$  # not-simplified-cls (clauses T)
  using  $\langle U \sim \text{reduce-trail-to}_{\text{NOT}} \sqcup T \rangle$  by auto
moreover have not-simplified-cls (clauses T)  $\subseteq$  # not-simplified-cls (clauses S)
  apply (rule rtrancpl-cdclNOT-merged-bj-learn-not-simplified-decreasing)
  using  $\langle \text{cdcl}_{\text{NOT}}\text{-merged-bj-learn} \widetilde{m} S T \rangle$  by (auto dest!: relpowp-imp-rtrancpl)
ultimately have U-S: not-simplified-cls (clauses U)  $\subseteq$  # not-simplified-cls (clauses S)
  by auto

have (set-mset (clauses U))
 $\subseteq$  set-mset (not-simplified-cls (clauses U))  $\cup$  simple-clss (atms-of-ms A)
  apply (rule rtrancpl-cdclNOT-merged-bj-learn-clauses-bound)
  apply simp
  using  $\langle \text{inv } U \rangle$  apply simp
  using  $\langle \text{atms-of-msu (clauses U)} \subseteq \text{atms-of-ms A} \rangle$  apply simp
  using U apply simp
  using U apply simp
  using finite apply simp
done
then have f1: card (set-mset (clauses U))  $\leq$  card (set-mset (not-simplified-cls (clauses U))
 $\cup$  simple-clss (atms-of-ms A))
  by (simp add: simple-clss-finite card-mono local.finite)

moreover have set-mset (not-simplified-cls (clauses U))  $\cup$  simple-clss (atms-of-ms A)
 $\subseteq$  set-mset (not-simplified-cls (clauses S))  $\cup$  simple-clss (atms-of-ms A)
  using U-S by auto
then have f2:
  card (set-mset (not-simplified-cls (clauses U))  $\cup$  simple-clss (atms-of-ms A))
 $\leq$  card (set-mset (not-simplified-cls (clauses S))  $\cup$  simple-clss (atms-of-ms A))
  by (simp add: simple-clss-finite card-mono local.finite)

moreover have card (set-mset (not-simplified-cls (clauses S))
 $\cup$  simple-clss (atms-of-ms A))
 $\leq$  card (set-mset (not-simplified-cls (clauses S))) + card (simple-clss (atms-of-ms A))

```

using *card-Un-le* by *blast*
 moreover have *card* (*simple-clss* (*atms-of-ms* *A*)) $\leq 3 \wedge$ *card* (*atms-of-ms* *A*)
 using *atms-of-ms-finite 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 μ_{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 \langle *full1 cdcl_{NOT}-merged-bj-learn S T* \rangle **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 \langle *U ~ reduce-trail-to_{NOT} [] T* \rangle **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 \langle (*cdcl_{NOT}-merged-bj-learn* $\widetilde{\sim}$ *m*) *S T* \rangle **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*

$\lambda A S. \text{atms-of-msu} (\text{clauses } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}'\text{-merged } \text{cdcl}_{NOT}\text{-merged-bj-learn}$
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\lambda A T. ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * 2$
 $+ \text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses } T)))$
 $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$

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 $I \models_{\text{sextm}} \text{clauses } (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses } (\text{fst } T)$
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(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 $I \models_{\text{sextm}} \text{clauses } S \longleftrightarrow I \models_{\text{sextm}} \text{clauses } T$
using *cdcl_{NOT}.rtranclp-cdcl_{NOT}-bj-sat-ext-iff restart-step.prem(1,2)*
rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT} **by** *auto*
moreover have $I \models_{\text{sextm}} \text{clauses } T \longleftrightarrow I \models_{\text{sextm}} \text{clauses } U$
using *restart-step.hyps(3)* **by** *auto*
ultimately show *?case* **by** *auto*

qed

lemma *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 $I \models_{\text{sextm}} \text{clauses } (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses } (\text{fst } T)$
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 $I \models_{\text{sextm}} \text{clauses } (\text{fst } T) \longleftrightarrow I \models_{\text{sextm}} \text{clauses } (\text{fst } U)$
using *cdcl_{NOT}-restart-eq-sat-iff cdcl* **by** *blast*
then show *?case* **using** *IH* **by** *blast*

qed

lemma *cdcl_{NOT}-restart-all-decomposition-implies-m*:

assumes
cdcl_{NOT}-restart S T **and**
inv: inv (*fst S*) **and** *n-d: no-dup*(*trail* (*fst S*)) **and**
all-decomposition-implies-m (*clauses* (*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 rtranclp-cdclNOT-with-restart-cdclNOT-inv using full inv n-d unfolding full-def by blast+

```

moreover have
atms-cls-T: *atms-of-msu* (*clauses* (*fst T*)) \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-clss-ext
unfolding *satisfiable-def full-def* **by** *blast*
then show *?thesis* **by** *blast*
next
case *sat*
then have *lits-of* (*trail* (*fst T*)) \models_{sextm} *clauses* (*fst T*)
using *true-clss-imp-true-clss-ext* **by** (*auto simp: true-annots-true-clss*)
then have *lits-of* (*trail* (*fst T*)) \models_{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 cdcl_{NOT}-restart* (*S*, 0) *T* **and**

inv: *inv S*

shows *unsatisfiable* (*set-mset N*)

\vee *lits-of* (*trail* (*fst T*)) \models_{sextm} *N* \wedge *satisfiable* (*set-mset N*)

using *full-cdcl_{NOT}-restart-normal-form*[*of (S, 0) T*] *assms* **by** *auto*

end

end

theory *DPLL-NOT*

imports *CDCL-NOT*

begin

15 DPLL as an instance of NOT

15.1 DPLL with simple backtrack

locale *dpll-with-backtrack*

begin

inductive *backtrack* :: ('v, unit, unit) marked-lit list \times 'v clauses

\Rightarrow ('v, unit, unit) marked-lit list \times 'v clauses \Rightarrow bool **where**

backtrack-split (*fst* *S*) = (*M'*, *L* # *M*) \Longrightarrow *is-marked* *L* \Longrightarrow *D* \in # *snd* *S*

\Longrightarrow *fst* *S* \models_{as} *CNot* *D* \Longrightarrow *backtrack* *S* (*Propagated* (\neg (*lit-of* *L*)) () # *M*, *snd* *S*)

inductive-cases *backtrackE*[*elim*]: *backtrack* (*M*, *N*) (*M'*, *N'*)

lemma *backtrack-is-backjump*:

fixes *M* *M'* :: ('v, unit, unit) marked-lit list

assumes

backtrack: *backtrack* (*M*, *N*) (*M'*, *N'*) **and**

no-dup: (*no-dup* \circ *fst*) (*M*, *N*) **and**

decomp: *all-decomposition-implies-m* *N* (*get-all-marked-decomposition* *M*)

shows

$\exists C F' K F L l C'$.

$M = F' @ \text{Marked } K () \# F \wedge$

$M' = \text{Propagated } L l \# F \wedge N = N' \wedge C \in \# N \wedge F' @ \text{Marked } K d \# F \models_{as} \text{CNot } C \wedge$

undefined-lit *F* *L* \wedge *atm-of* *L* \in *atms-of-msu* *N* \cup *atm-of* ' *lits-of* (*F'* @ *Marked* *K* *d* # *F*) \wedge

$N \models_{pm} C' + \{\#L\# \} \wedge F \models_{as} \text{CNot } C'$

proof –

let ?*S* = (*M*, *N*)

let ?*T* = (*M'*, *N'*)

obtain *F* *F'* *P* *L* *D* **where**

b-sp: *backtrack-split* *M* = (*F'*, *L* # *F*) **and**

is-marked *L* **and**

D \in # *snd* ?*S* **and**

M \models_{as} *CNot* *D* **and**

bt: *backtrack* ?*S* (*Propagated* (\neg (*lit-of* *L*)) *P* # *F*, *N*) **and**

M': *M'* = *Propagated* (\neg (*lit-of* *L*)) *P* # *F* **and**

[*simp*]: *N'* = *N*

using *backtrackE*[*OF* *backtrack*] **by** (*metis* *backtrack* *fstI* *sndI*)

let ?*K* = *lit-of* *L*

let ?*C* = *image-mset* *lit-of* { $\#K \in \#mset$ *M*. *is-marked* *K* \wedge *K* \neq *L* #} :: 'v literal multiset

let ?*C'* = *set-mset* (*image-mset* *single* (?*C* + { $\#?K\#$ }))

obtain *K* **where** *L*: *L* = *Marked* *K* () **using** \langle *is-marked* *L* \rangle **by** (*cases* *L*) *auto*

have *M*: *M* = *F'* @ *Marked* *K* () # *F*

using *b-sp* **by** (*metis* *L* *backtrack-split-list-eq* *fst-conv* *snd-conv*)

moreover **have** *F'* @ *Marked* *K* () # *F* \models_{as} *CNot* *D*

using \langle *M* \models_{as} *CNot* *D* \rangle **unfolding** *M* .

moreover **have** *undefined-lit* *F* (\neg ?*K*)

using *no-dup* **unfolding** *M* *L* **by** (*simp* *add*: *defined-lit-map*)

```

moreover have  $\text{atm-of } (-K) \in \text{atms-of-msu } N \cup \text{atm-of ' lit-of } (F' @ \text{Marked } K \text{ d } \# F)$ 
  by auto
moreover
have  $\text{set-mset } N \cup ?C' \models_{ps} \{\{\#\}\}$ 
  proof –
    have  $A: \text{set-mset } N \cup ?C' \cup \text{unmark } M =$ 
       $\text{set-mset } N \cup \text{unmark } M$ 
    unfolding  $M \text{ } L$  by auto
    have  $\text{set-mset } N \cup \{\{\#\text{lit-of } L\#\} \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$ 
       $\models_{ps} \text{unmark } M$ 
    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  $M \text{ } L$  apply standard
    apply force
    using IntI by auto
    ultimately have  $N\text{-}C\text{-}M: \text{set-mset } N \cup ?C' \models_{ps} \text{unmark } M$ 
    by auto
    have  $\text{set-mset } N \cup (\lambda L. \{\#\text{lit-of } L\#\}) \text{ ' } (\text{set } M) \models_{ps} \{\{\#\}\}$ 
    unfolding true-clss-clss-def
    proof (intro allI impI, goal-cases)
      case ( $1 \text{ } I$ ) note  $\text{tot} = \text{this}(1)$  and  $\text{cons} = \text{this}(2)$  and  $I\text{-}N\text{-}M = \text{this}(3)$ 
      have  $I \models D$ 
        using  $I\text{-}N\text{-}M \langle D \in \# \text{ snd } ?S \rangle$  unfolding true-clss-def by auto
      moreover have  $I \models_s C\text{Not } D$ 
        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  $N \models_{pm} \text{image-mset } \text{uminus } ?C + \{\#\text{--} ?K\#\}$ 
unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
proof (intro allI impI)
  fix  $I$ 
  assume
     $\text{tot}: \text{total-over-set } I (\text{atms-of-ms } (\text{set-mset } N \cup \{\text{image-mset } \text{uminus } ?C + \{\#\text{--} ?K\#\}\}))$  and
     $\text{cons}: \text{consistent-interp } I$  and
     $I \models_{sm} N$ 
  have  $(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$ 
    using cons tot unfolding consistent-interp-def L by (cases K) auto
  have  $tI: \text{total-over-set } I (\text{atm-of ' lit-of ' } (\text{set } M \cap \{L. \text{is-marked } L \wedge L \neq \text{Marked } K \text{ d}\}))$ 
    using tot by (auto simp add: L atms-of-uminus-lit-atm-of-lit-of)

then have  $H: \bigwedge x.$ 
   $\text{lit-of } x \notin I \implies x \in \text{set } M \implies \text{is-marked } x$ 
   $\implies x \neq \text{Marked } K \text{ d} \implies -\text{lit-of } x \in I$ 
  proof –
    fix  $x :: ('v, \text{unit}, \text{unit}) \text{ marked-lit}$ 
    assume  $a1: x \neq \text{Marked } K \text{ d}$ 
    assume  $a2: \text{is-marked } x$ 
    assume  $a3: x \in \text{set } M$ 
    assume  $a4: \text{lit-of } x \notin I$ 
    have  $\text{atm-of } (\text{lit-of } x) \in \text{atm-of ' lit-of ' }$ 

```

```

    (set  $M \cap \{m. \text{ is-marked } m \wedge m \neq \text{Marked } K \, d\}$ )
    using a3 a2 a1 by blast
  then have Pos (atm-of (lit-of  $x$ ))  $\in I \vee \text{Neg (atm-of (lit-of } x)) \in I$ 
    using tI unfolding total-over-set-def by blast
  then show  $\neg \text{lit-of } x \in I$ 
    using a4 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      literal.sel(1,2))
qed
have  $\neg I \models_s ?C'$ 
  using (set-mset  $N \cup ?C' \models_{ps} \{\{\#\}\}$  tot cons  $\langle I \models_{sm} N \rangle$ )
  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  $I \models \text{image-mset uminus } ?C + \{\# - \text{lit-of } L\# \}$ 
  unfolding true-clss-def true-cls-def Bex-mset-def
  using  $\langle (K \in I \wedge \neg K \notin I) \vee (\neg K \in I \wedge K \notin I) \rangle$ 
  unfolding L by (auto dest!: H)
qed
moreover
  have 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  $F \models_{as} \text{CNot (image-mset uminus } ?C)$ 
    unfolding M CNot-def true-annots-def by (auto simp add: L lits-of-def)
ultimately show ?thesis
  using  $M' \langle D \in \# \text{ snd } ?S \rangle L$  by force
qed

lemma backtrack-is-backjump':
  fixes  $M \, M' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list}$ 
  assumes
    backtrack: 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
     $\exists C \, F' \, K \, F \, L \, l \, C'.$ 
     $\text{fst } S = F' @ \text{Marked } K \, () \, \# \, F \wedge$ 
     $T = (\text{Propagated } L \, l \, \# \, F, \text{snd } S) \wedge C \in \# \, \text{snd } S \wedge \text{fst } S \models_{as} \text{CNot } C$ 
     $\wedge \text{undefined-lit } F \, L \wedge \text{atm-of } L \in \text{atms-of-msu (snd } S) \cup \text{atm-of ' lits-of (fst } S) \wedge$ 
     $\text{snd } S \models_{pm} C' + \{\#L\# \} \wedge F \models_{as} \text{CNot } C'$ 
  apply (cases S, cases T)
  using backtrack-is-backjump[of fst S snd S fst T snd T] assms by fastforce

sublocale dpll-state fst snd  $\lambda L \, (M, N). (L \# M, N) \lambda(M, N). (\text{tl } M, N)$ 
 $\lambda C \, (M, N). (M, \{\#C\# \} + N) \lambda C \, (M, N). (M, \text{remove-mset } C \, N)$ 
by unfold-locales auto

sublocale backjumping-ops fst snd  $\lambda L \, (M, N). (L \# M, N) \lambda(M, N). (\text{tl } M, N)$ 
 $\lambda C \, (M, N). (M, \{\#C\# \} + N) \lambda C \, (M, N). (M, \text{remove-mset } C \, N) \lambda - - S \, T. \text{backtrack } S \, T$ 
by unfold-locales

lemma backtrack-is-backjump'':
  fixes  $M \, M' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list}$ 
  assumes
    backtrack: 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 *backjump* $S\ T$
proof –
obtain $C\ F'\ K\ F\ L\ l\ C'$ **where**
1: $\text{fst } S = F' @ \text{Marked } K\ () \# F$ **and**
2: $T = (\text{Propagated } L\ l \# F, \text{snd } S)$ **and**
3: $C \in \# \text{snd } S$ **and**
4: $\text{fst } S \models_{\text{as}} \text{CNot } C$ **and**
5: *undefined-lit* $F\ L$ **and**
6: $\text{atm-of } L \in \text{atms-of-msu } (\text{snd } S) \cup \text{atm-of 'lits-of } (\text{fst } S)$ **and**
7: $\text{snd } S \models_{\text{pm}} C' + \{\#L\# \}$ **and**
8: $F \models_{\text{as}} \text{CNot } C'$
using *backtrack-is-backjump'*[*OF* *assms*] **by** *blast*
show *?thesis*
using *backjump.intros*[*OF* 1 - 3 4 5 6 7 8] 2 *backtrack* 1 5
by (*auto simp: state-eq_{NOT}-def simp del: state-simp_{NOT}*)
qed

lemma *can-do-bt-step*:
assumes
 $M: \text{fst } S = F' @ \text{Marked } K\ d \# F$ **and**
 $C \in \# \text{snd } S$ **and**
 $C: \text{fst } S \models_{\text{as}} \text{CNot } C$
shows $\neg \text{no-step backtrack } S$
proof –
obtain $L\ G'\ G$ **where**
backtrack-split $(\text{fst } S) = (G', L \# G)$
unfolding M **by** (*induction* F' *rule: marked-lit-list-induct*) *auto*
moreover then have *is-marked* L
by (*metis backtrack-split-snd-hd-marked list.distinct*(1) *list.sel*(1) *snd-conv*)
ultimately show *?thesis*
using *backtrack.intros*[*of* $S\ G'\ L\ G\ C$] $\langle C \in \# \text{snd } S \rangle C$ **unfolding** M **by** *auto*
qed

end

sublocale *dpll-with-backtrack* \subseteq *dpll-with-backjumping-ops* *fst* *snd* $\lambda L\ (M, N). (L \# M, N)$
 $\lambda(M, N). (\text{tl } M, N) \lambda C\ (M, N). (M, \{\#C\# \} + N) \lambda C\ (M, N). (M, \text{remove-mset } C\ N) \lambda - -. \text{True}$
 $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N\ (\text{get-all-marked-decomposition } M)$
 $\lambda - - S\ T. \text{backtrack } S\ T$
by *unfold-locales* (*metis* (*mono-tags*, *lifting*) *dpll-with-backtrack.backtrack-is-backjump''*
dpll-with-backtrack.can-do-bt-step prod.case-eq-if comp-apply)

sublocale *dpll-with-backtrack* \subseteq *dpll-with-backjumping* *fst* *snd* $\lambda L\ (M, N). (L \# M, N)$
 $\lambda(M, N). (\text{tl } M, N) \lambda C\ (M, N). (M, \{\#C\# \} + N) \lambda C\ (M, N). (M, \text{remove-mset } C\ N) \lambda - -. \text{True}$
 $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N\ (\text{get-all-marked-decomposition } M)$
 $\lambda - - S\ T. \text{backtrack } S\ T$
apply *unfold-locales*
using *dpll-bj-no-dup dpll-bj-all-decomposition-implies-inv* **apply** *fastforce*
done

sublocale *dpll-with-backtrack* \subseteq *conflict-driven-clause-learning-ops*
fst *snd* $\lambda L\ (M, N). (L \# M, N)$
 $\lambda(M, N). (\text{tl } M, N) \lambda C\ (M, N). (M, \{\#C\# \} + N) \lambda C\ (M, N). (M, \text{remove-mset } C\ N) \lambda - -. \text{True}$
 $\lambda(M, N). \text{no-dup } M \wedge \text{all-decomposition-implies-m } N\ (\text{get-all-marked-decomposition } M)$
 $\lambda - - S\ T. \text{backtrack } S\ T \lambda - -. \text{False } \lambda - -. \text{False}$

by *unfold-locales*

sublocale *dpll-with-backtrack* \subseteq *conflict-driven-clause-learning*

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 forget_{NOT}E learn_{NOT}E* **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-cdcl_{NOT}-no-learn-and-forget-infinite-chain*)

(*auto simp: learn.simps forget_{NOT}.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 (rename-tac A T U, case-tac T, simp)
    apply (rename-tac A T U, case-tac U, simp)
    using dpll-bj-clauses dpll-bj-all-decomposition-implies-inv dpll-bj-no-dup by fastforce +
end

end
theory DPLL-W
imports Main Partial-Clausal-Logic Partial-Annotated-Clausal-Logic List-More Wellfounded-More
  DPLL-NOT
begin

```

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:  $backtrack-split\ (trail\ S) = (M', L \# M) \implies is-marked\ L \implies D \in \# clauses\ S$ 
     $\implies trail\ S \models_{as} CNot\ D \implies dpll_W\ S\ (Propagated\ (-\ (lit-of\ L))\ () \# M, clauses\ S)$ 

```

16.2 Invariants

```

lemma dpllW-distinct-inv:
  assumes dpllW S S'
  and no-dup (trail S)
  shows no-dup (trail S')

```



```

using assms
proof (induct rule: dpllW.induct)
  case (decided L S)
  then show ?case using defined-lit-map by force
next
  case (propagate C L S)
  then show ?case using defined-lit-map by force
next
  case (backtrack S M' L M D) note extracted = this(1) and no-dup = this(5)
  show ?case
    using no-dup backtrack-split-list-eq[of trail S, symmetric] unfolding extracted by auto
qed

lemma dpllW-consistent-interp-inv:
  assumes dpllW 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: dpllW.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)  $\subseteq$  lits-of (trail S)
    using backtrack-split-list-eq[of trail S, symmetric] unfolding extracted by auto
  then have cons: consistent-interp (insert (lit-of L) (lits-of M))
    using consistent-interp-subset cons by blast
  moreover
    have lit-of L  $\notin$  lits-of M
      using no-dup backtrack-split-list-eq[of trail S, symmetric] extracted
      unfolding lits-of-def by force
  moreover
    have atm-of ( $\neg$ lit-of L)  $\notin$  ( $\lambda m.$  atm-of (lit-of m)) ' set M
      using no-dup backtrack-split-list-eq[of trail S, symmetric] unfolding extracted by force
    then have  $\neg$ lit-of L  $\notin$  lits-of M
      unfolding lits-of-def by force
  ultimately show ?case by simp
qed (auto intro: consistent-add-undefined-lit-consistent)

lemma dpllW-vars-in-snd-inv:
  assumes dpllW S S'
  and atm-of ' (lits-of (trail S))  $\subseteq$  atms-of-msu (clauses S)
  shows atm-of ' (lits-of (trail S'))  $\subseteq$  atms-of-msu (clauses S')
  using assms
proof (induct rule: dpllW.induct)
  case (backtrack S M' L M D)
  then have atm-of (lit-of L)  $\in$  atms-of-msu (clauses S)
    using backtrack-split-list-eq[of trail S, symmetric] by auto
  moreover
    have atm-of ' lits-of (trail S)  $\subseteq$  atms-of-msu (clauses S)
      using backtrack(5) by simp
    then have  $\bigwedge xb. xb \in \text{set } M \implies \text{atm-of (lit-of } xb) \in \text{atms-of-msu (clauses } S)$ 
      using backtrack-split-list-eq[symmetric, of trail S] backtrack.hyps(1)

```

unfolding *lits-of-def* **by** *auto*
ultimately show *?case* **by** (*auto simp : lits-of-def*)
qed (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

lemma *atms-of-ms-lit-of-atms-of*: *atms-of-ms* (($\lambda a. \{\# \text{lit-of } a \# \}$) ‘ *c*) = *atm-of* ‘ *lit-of* ‘ *c*
unfolding *atms-of-ms-def* **using** *image-iff* **by** *force*

Lemma theorem 2.8.2 page 71 of CW

lemma *dpll_W-propagate-is-conclusion*:

assumes *dpll_W S S'*
and *all-decomposition-implies-m* (*clauses S*) (*get-all-marked-decomposition* (*trail S*))
and *atm-of* ‘ *lits-of* (*trail S*) \subseteq *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*)) \cup *set-mset* (*clauses S*)
have *?I* $\models_p C + \{\# L \# \}$ **by** (*auto simp add: inS*)
moreover have *?I* $\models_{ps} CNot\ C$ **using** *true-annots-true-clss-cl* *cnot* **by** *fastforce*
ultimately have *?I* $\models_p \{\# L \# \}$ **using** *true-clss-cl* *plus-CNot* [*of ?I C L*] *inS* **by** *blast*
{
assume *get-all-marked-decomposition* (*trail S*) = []
then have *?case* **by** *blast*
}
moreover {
assume *n*: *get-all-marked-decomposition* (*trail S*) \neq []
have 1: $\bigwedge a\ b. (a, b) \in \text{set } (tl\ (get-all-marked-decomposition\ (trail\ S)))$
\implies (*unmark a* \cup *set-mset* (*clauses S*)) \models_{ps} *unmark b*
using *IH* **unfolding** *all-decomposition-implies-def* **by** (*fastforce simp add: list.set-sel*(2) *n*)
moreover have 2: $\bigwedge a\ c. hd\ (get-all-marked-decomposition\ (trail\ S)) = (a, c)$
\implies (*unmark a* \cup *set-mset* (*clauses S*)) \models_{ps} (*unmark c*)
by (*metis IH all-decomposition-implies-cons-pair all-decomposition-implies-single list.collapse n*)
moreover have 3: $\bigwedge a\ c. hd\ (get-all-marked-decomposition\ (trail\ S)) = (a, c)$
\implies (*unmark a* \cup *set-mset* (*clauses S*)) $\models_p \{\# L \# \}$
proof –
fix *a c*
assume *h*: *hd* (*get-all-marked-decomposition* (*trail S*)) = (*a*, *c*)
have *h'*: *trail S* = *c* @ *a* **using** *get-all-marked-decomposition-decomp* *h* **by** *blast*
have *I*: *set* (*map* ($\lambda a. \{\# \text{lit-of } a \# \}$) *a*) \cup *set-mset* (*clauses S*)
\cup *unmark c* $\models_{ps} CNot\ C$
using (*?I* $\models_{ps} CNot\ C$) **unfolding** *h'* **by** (*simp add: Un-commute Un-left-commute*)
have
***atms-of-ms* (*CNot C*) \subseteq *atms-of-ms* (*set* (*map* ($\lambda a. \{\# \text{lit-of } a \# \}$) *a*) \cup *set-mset* (*clauses S*))**
and
***atms-of-ms* (*unmark c*) \subseteq *atms-of-ms* (*set* (*map* ($\lambda a. \{\# \text{lit-of } a \# \}$) *a*)**
\cup *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 unmark  $a \cup \text{set-mset}(\text{clauses } S) \models_{ps} \text{CNot } C$ 
      using true-clss-clss-left-right[OF - I] h 2 by auto
    then show unmark  $a \cup \text{set-mset}(\text{clauses } S) \models_p \{\#L\# \}$ 
      by (metis (no-types) Un-insert-right inS insertI1 mk-disjoint-insert inS mem-set-mset-iff
        true-clss-clss-in true-clss-clss-plus-CNot)
    qed
  ultimately have ?case
    by (cases hd (get-all-marked-decomposition (trail S)))
      (auto simp: all-decomposition-implies-def)
}
ultimately show ?case by auto
next
case (backtrack  $S M' L M D$ ) note extracted = this(1) and marked = this(2) and  $D = \text{this}(3)$  and
  cnot = this(4) and cons = this(4) and  $IH = \text{this}(5)$  and  $\text{atms-incl} = \text{this}(6)$ 
have  $S: \text{trail } S = M' @ L \# M$ 
  using backtrack-split-list-eq[of trail S] unfolding extracted by auto
have  $M': \forall l \in \text{set } M'. \neg \text{is-marked } l$ 
  using extracted backtrack-split-fst-not-marked[of - trail S] by simp
have  $n: \text{get-all-marked-decomposition}(\text{trail } S) \neq []$  by auto
then have all-decomposition-implies-m (clauses S) (( $L \# M, M'$ )
  # tl (get-all-marked-decomposition (trail S)))
  by (metis (no-types) IH extracted get-all-marked-decomposition-backtrack-split list.exhaust-sel)
then have 1: unmark ( $L \# M$ )  $\cup \text{set-mset}(\text{clauses } S) \models_{ps} (\lambda a. \{\#\text{lit-of } a\# \})$  'set  $M'$ '
  by simp
moreover
have unmark ( $L \# M$ )  $\cup \text{unmark } M' \models_{ps} \text{CNot } D$ 
  by (metis (mono-tags, lifting) S Un-commute cons image-Un set-append
    true-annots-true-clss-clss)
then have 2: unmark ( $L \# M$ )  $\cup \text{set-mset}(\text{clauses } S) \cup \text{unmark } M' \models_{ps} \text{CNot } D$ 
  by (metis (no-types, lifting) Un-assoc Un-left-commute true-clss-clss-union-l-r)
ultimately
have set (map ( $\lambda a. \{\#\text{lit-of } a\# \}$ ) ( $L \# M$ ))  $\cup \text{set-mset}(\text{clauses } S) \models_{ps} \text{CNot } D$ 
  using true-clss-clss-left-right by fastforce
then have set (map ( $\lambda a. \{\#\text{lit-of } a\# \}$ ) ( $L \# M$ ))  $\cup \text{set-mset}(\text{clauses } S) \models_p \{\#\}$ 
  by (metis (mono-tags, lifting) D Un-def mem-Collect-eq set-mset-def
    true-clss-clss-contradiction-true-clss-clss-false)
then have IL: unmark  $M \cup \text{set-mset}(\text{clauses } S) \models_p \{\# - \text{lit-of } L\# \}$ 
  using true-clss-clss-false-left-right by auto
show ?case unfolding S all-decomposition-implies-def
proof
  fix  $x$  P level
  assume  $x: x \in \text{set}(\text{get-all-marked-decomposition}(\text{fst}(\text{Propagated}(\neg \text{lit-of } L) P \# M, \text{clauses } S)))$ 
  let  $?M' = \text{Propagated}(\neg \text{lit-of } L) P \# M$ 
  let  $?hd = \text{hd}(\text{get-all-marked-decomposition } ?M')$ 
  let  $?tl = \text{tl}(\text{get-all-marked-decomposition } ?M')$ 
  have  $x = ?hd \vee x \in \text{set } ?tl$ 
  using x
  by (cases get-all-marked-decomposition ?M')
    auto
  moreover {
    assume  $x': x \in \text{set } ?tl$ 
    have  $L': \text{Marked}(\text{lit-of } L) () = L$  using marked by (cases L, auto)
    have  $x \in \text{set}(\text{get-all-marked-decomposition}(M' @ L \# M))$ 

```

```

    using x' get-all-marked-decomposition-except-last-choice-equal[of M' lit-of L P M]
    L' by (metis (no-types) M' list.set-sel(2) tl-Nil)
  then have case x of (Ls, seen)  $\Rightarrow$  unmark Ls  $\cup$  set-mset (clauses S)
     $\models_{ps}$  unmark seen
    using marked IH by (cases L) (auto simp add: S all-decomposition-implies-def)
}
moreover {
  assume x': x = ?hd
  have tl: tl (get-all-marked-decomposition (M' @ L # M))  $\neq$  []
  proof -
    have f1:  $\bigwedge ms.$  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': unmark M0  $\cup$  set-mset (clauses S)
     $\cup$  unmark M0'  $\models_{ps}$   $\{\{\# - \text{lit-of } L\#\}\}$ 
    using IL by (simp add: Un-commute Un-left-commute image-Un)
  moreover have H: unmark M0  $\cup$  set-mset (clauses S)
     $\models_{ps}$  unmark M0'
    using IH x'' unfolding all-decomposition-implies-def by (metis (no-types, lifting) L0 S
      list.set-sel(1) list.set-sel(2) old.prod.case tl tl-Nil)
  ultimately have case x of (Ls, seen)  $\Rightarrow$  unmark Ls  $\cup$  set-mset (clauses S)
     $\models_{ps}$  unmark seen
    using true-clss-clss-left-right unfolding x'' by auto
}
ultimately show case x of (Ls, seen)  $\Rightarrow$ 
  unmark Ls  $\cup$  set-mset (snd (?M', clauses S))
   $\models_{ps}$  unmark 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 $\{\{\#\text{lit-of } L\#\} \mid L. \text{ is-marked } L \wedge L \in \text{set (trail S')}\}$*

$\models_{ps} (\lambda a. \{\#lit\text{-of } a\#\}) \text{ ' } \bigcup (\text{set ' } snd \text{ ' set (get-all-marked-decomposition (trail } S')))$
using *all-decomposition-implies-trail-is-implied*[*OF dpll_W-propagate-is-conclusion*[*OF assms*]] .

Lemma theorem 2.8.4 page 72 of CW

lemma *only-propagated-vars-unsat*:

assumes *marked*: $\forall x \in \text{set } M. \neg \text{is-marked } x$

and *DN*: $D \in N$ **and** *D*: $M \models_{as} CNot D$

and *inv*: *all-decomposition-implies* *N* (*get-all-marked-decomposition* *M*)

and *atm-incl*: *atm-of* ' *lits-of* *M* \subseteq *atms-of-ms* *N*

shows *unsatisfiable* *N*

proof (*rule ccontr*)

assume $\neg \text{unsatisfiable } N$

then obtain *I* **where**

I: $I \models_s N$ **and**

cons: *consistent-interp* *I* **and**

tot: *total-over-m* *I* *N*

unfolding *satisfiable-def* **by** *auto*

then have *I-D*: $I \models D$

using *DN* **unfolding** *true-clss-def* **by** *auto*

have *l0*: $\{\{\#lit\text{-of } L\#\} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} = \{\}$ **using** *marked* **by** *auto*

have *atms-of-ms* ($N \cup \text{unmark } M$) = *atms-of-ms* *N*

using *atm-incl* **unfolding** *atms-of-ms-def lits-of-def* **by** *auto*

then have *total-over-m* *I* ($N \cup (\lambda a. \{\#lit\text{-of } a\#\}) \text{ ' (set } M)$)

using *tot* **unfolding** *total-over-m-def* **by** *auto*

then have $I \models_s (\lambda a. \{\#lit\text{-of } a\#\}) \text{ ' (set } M)$

using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons* *I*

unfolding *true-clss-clss-def* **by** *auto*

then have *IM*: $I \models_s \text{unmark } M$ **by** *auto*

{

fix *K*

assume $K \in \# D$

then have $-K \in \text{lits-of } M$

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 *rtrancp-dpll_W-inv*:

assumes *rtrancp 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* $S = \text{clauses } S'$
and *consistent-interp* (*lits-of* (*trail* S'))
and *no-dup* (*trail* S')
using *assms*
proof (*induct rule: rtranclp-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 $S = \text{clauses } S$ **and**
consistent-interp (*lits-of* (*trail* S)) **and**
no-dup (*trail* S) **using** *assms* **by** *auto*
next
case (*step* $S' S''$) **note** $\text{dpll}_W \text{Star} = \text{this}(1)$ **and** $IH = \text{this}(3,4,5,6,7)$ **and**
 $\text{dpll}_W = \text{this}(2)$
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* $S = \text{clauses } S'$ **and**
cons': *consistent-interp* (*lits-of* (*trail* S')) **and**
no-dup': *no-dup* (*trail* S') **by** *blast+*
show *clauses* $S = \text{clauses } S''$ **using** $\text{dpll}_W\text{-same-clauses}[OF \text{dpll}_W]$ *snd* **by** *metis*

show *all-decomposition-implies-m* (*clauses* S'') (*get-all-marked-decomposition* (*trail* S''))
using $\text{dpll}_W\text{-propagate-is-conclusion}[OF \text{dpll}_W]$ *decomp* *atm-incl'* **by** *auto*
show *atm-of* ' *lits-of* (*trail* S'') \subseteq *atms-of-msu* (*clauses* S'')
using $\text{dpll}_W\text{-vars-in-snd-inv}[OF \text{dpll}_W]$ *atm-incl* *atm-incl'* **by** *auto*
show *no-dup* (*trail* S'') **using** $\text{dpll}_W\text{-distinct-inv}[OF \text{dpll}_W]$ *no-dup'* dpll_W **by** *auto*
show *consistent-interp* (*lits-of* (*trail* S''))
using *cons'* *no-dup'* $\text{dpll}_W\text{-consistent-interp-inv}[OF \text{dpll}_W]$ **by** *auto*
qed

definition $\text{dpll}_W\text{-all-inv } S \equiv$
(*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 $\text{dpll}_W\text{-all-inv-dest}[dest]$:
assumes $\text{dpll}_W\text{-all-inv } S$
shows *all-decomposition-implies-m* (*clauses* S) (*get-all-marked-decomposition* (*trail* S))
and *atm-of* ' *lits-of* (*trail* S) \subseteq *atms-of-msu* (*clauses* S)
and *consistent-interp* (*lits-of* (*trail* S)) \wedge *no-dup* (*trail* S)
using *assms* **unfolding** $\text{dpll}_W\text{-all-inv-def}$ *lits-of-def* **by** *auto*

lemma $\text{rtranclp-dpll}_W\text{-all-inv}$:
assumes $\text{rtranclp } \text{dpll}_W S S'$
and $\text{dpll}_W\text{-all-inv } S$
shows $\text{dpll}_W\text{-all-inv } S'$

using *assms rtrancpl-dpll_W-inv*[*OF assms(1)*] **unfolding** *dpll_W-all-inv-def lits-of-def* **by** *blast*

lemma *dpll_W-all-inv*:
assumes *dpll_W S S'*
and *dpll_W-all-inv S*
shows *dpll_W-all-inv S'*
using *assms rtrancpl-dpll_W-all-inv* **by** *blast*

lemma *rtrancpl-dpll_W-inv-starting-from-0*:
assumes *rtrancpl dpll_W S S'*
and *inv: trail S = []*
shows *dpll_W-all-inv S'*

proof –
have *dpll_W-all-inv S*
using *assms* **unfolding** *all-decomposition-implies-def dpll_W-all-inv-def* **by** *auto*
then show *?thesis* **using** *rtrancpl-dpll_W-all-inv*[*OF assms(1)*] **by** *blast*
qed

lemma *dpll_W-can-do-step*:
assumes *consistent-interp (set M)*
and *distinct M*
and *atm-of ' (set M) ⊆ atms-of-msu N*
shows *rtrancpl dpll_W ([], N) (map (λM. Marked M ()) M, N)*
using *assms*

proof (*induct M*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons L M*)
then have *undefined-lit (map (λM. Marked M ()) M) L*
unfolding *defined-lit-def consistent-interp-def* **by** *auto*
moreover have *atm-of L ∈ atms-of-msu N* **using** *Cons.prem(3)* **by** *auto*
ultimately have *dpll_W (map (λM. Marked M ()) M, N) (map (λM. Marked M ()) (L # M), N)*
using *dpll_W.decided* **by** *auto*
moreover have *consistent-interp (set M)* **and** *distinct M* **and** *atm-of ' set M ⊆ atms-of-msu N*
using *Cons.prem* **unfolding** *consistent-interp-def* **by** *auto*
ultimately show *?case* **using** *Cons.hyps* **by** *auto*
qed

definition *conclusive-dpll_W-state* (*S:: 'v dpll_W-state*) \longleftrightarrow
(*trail S* \models_{asm} *clauses S* \vee ($\forall L \in \text{set } (\text{trail } S). \neg \text{is-marked } L$)
 \wedge ($\exists C \in \# \text{ clauses } S. \text{trail } S \models_{as} C \text{Not } C$))

lemma *dpll_W-strong-completeness*:
assumes *set M* \models_{sm} *N*
and *consistent-interp (set M)*
and *distinct M*
and *atm-of ' (set M) ⊆ atms-of-msu N*
shows *dpll_W** ([], N) (map (λM. Marked M ()) M, N)*
and *conclusive-dpll_W-state (map (λM. Marked M ()) M, N)*
proof –
show *rtrancpl dpll_W ([], N) (map (λM. Marked M ()) M, N)* **using** *dpll_W-can-do-step assms* **by** *auto*
have *map (λM. Marked M ()) M* \models_{asm} *N* **using** *assms(1) true-annots-marked-true-cl* **by** *auto*
then show *conclusive-dpll_W-state (map (λM. Marked M ()) M, N)*

unfolding *conclusive-dpll_W-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 $M \models_{asm} N \longleftrightarrow \text{satisfiable } (set\text{-}mset\ N)$ **(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-dpll_W-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*

by (*metis Bex-mset-def D atms-of-atms-of-ms-mono mem-set-mset-iff rev-subsetD*)

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 $\forall l \in \text{set } M. \neg \text{is-marked } l$

moreover have *dpll_W-all-inv* ([], N)

using *assms* **unfolding** *all-decomposition-implies-def dpll_W-all-inv-def* **by** *auto*

ultimately have *unsatisfiable* (*set-mset* N)

using *only-propagated-vars-unsat*[*of* $M\ D\ \text{set-mset } N$] $DN\ MD$

rtrancpl-dpll_W-all-inv[*OF assms(1)*] **by** *force*

then have *False* **using** $\langle ?B \rangle$ **by** *blast*

}

moreover {

assume $l: \exists l \in \text{set } M. \text{is-marked } l$

then have *False*

using *backtrack*[*of* $(M, N) \text{ - - - } D$] $DN\ MD\ \text{assms}(2)$

backtrack-split-some-is-marked-then-snd-has-hd[*OF* l]

by (*metis backtrack-split-snd-hd-marked fst-conv list.distinct(1) list.sel(1) snd-conv*)

}

ultimately have *False* **by** *blast*


```

    }
    ultimately show False by blast
qed
qed

```

16.3 Termination

definition $dpll_W\text{-mes } M n =$
 $map (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } (1::nat)) (\text{rev } M) @ replicate (n - \text{length } M) 3$

lemma $\text{length-dpll}_W\text{-mes}$:
assumes $\text{length } M \leq n$
shows $\text{length } (dpll_W\text{-mes } M n) = n$
using *assms* **unfolding** $dpll_W\text{-mes-def}$ **by** *auto*

lemma $\text{distinctcard-atm-of-lits-of-eq-length}$:
assumes $\text{no-dup } S$
shows $\text{card } (\text{atm-of } \text{'lits-of } S) = \text{length } S$
using *assms* **by** (*induct* S) (*auto simp add: image-image lits-of-def*)

lemma $dpll_W\text{-card-decrease}$:
assumes $dpll: dpll_W S S'$ **and** $\text{length } (\text{trail } S') \leq \text{card vars}$
and $\text{length } (\text{trail } S) \leq \text{card vars}$
shows $(dpll_W\text{-mes } (\text{trail } S') (\text{card vars}), dpll_W\text{-mes } (\text{trail } S) (\text{card vars}))$
 $\in \text{lexn } \{(a, b). a < b\} (\text{card vars})$
using *assms*

proof (*induct rule: dpll_W.induct*)
case (*propagate C L S*)
have $m: map (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S))$
 $@ replicate (\text{card vars} - \text{length } (\text{trail } S)) 3$
 $= map (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S)) @ 3$
 $\# replicate (\text{card vars} - \text{Suc } (\text{length } (\text{trail } S))) 3$
using *propagate.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*
then show *?case*
using *propagate.prem[s(1)]* **unfolding** $dpll_W\text{-mes-def}$ **by** (*fastforce simp add: lexn-conv assms(2)*)

next

case (*decided S L*)
have $m: map (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S))$
 $@ replicate (\text{card vars} - \text{length } (\text{trail } S)) 3$
 $= map (\lambda l. \text{if is-marked } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S)) @ 3$
 $\# replicate (\text{card vars} - \text{Suc } (\text{length } (\text{trail } S))) 3$
using *decided.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*
then show *?case*
using *decided.prem* **unfolding** $dpll_W\text{-mes-def}$ **by** (*force simp add: lexn-conv assms(2)*)

next

case (*backtrack S M' L M D*)
have $L: \text{is-marked } L$ **using** *backtrack.hyps(2)* **by** *auto*
have $S: \text{trail } S = M' @ L \# M$
using *backtrack.hyps(1)* *backtrack-split-list-eq[of trail S]* **by** *auto*
show *?case*
using *backtrack.prem L* **unfolding** $dpll_W\text{-mes-def}$ S **by** (*fastforce simp add: lexn-conv assms(2)*)

qed

Proposition theorem 2.8.7 page 73 of CW

lemma $dpll_W\text{-card-decrease'}$:
assumes $dpll: dpll_W S S'$

and *atm-incl*: *atm-of* ' *lits-of* (*trail S*) \subseteq *atms-of-msu* (*clauses S*)
and *no-dup*: *no-dup* (*trail S*)
shows (*dpll_W-mes* (*trail S'*) (*card* (*atms-of-msu* (*clauses S'*))),
dpll_W-mes (*trail S*) (*card* (*atms-of-msu* (*clauses S*)))) \in *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*) \leq *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 dpll_W-distinct-inv no-dup* **by** *blast*
have *SS'*: *clauses S' = clauses S* **using** *dpll* **by** (*auto dest!*: *dpll_W-same-clauses*)
have *atm-incl'*: *atm-of* ' *lits-of* (*trail S'*) \subseteq *atms-of-msu* (*clauses S'*)
using *atm-incl dpll dpll_W-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'*) \leq *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 (*dpll_W-mes* (*trail S'*) (*card* (*atms-of-msu* (*clauses S*))),
dpll_W-mes (*trail S*) (*card* (*atms-of-msu* (*clauses S*))))
 \in *lexn* {(*a*, *b*). *a* < *b*} (*card* (*atms-of-msu* (*clauses S*)))
using *dpll_W-card-decrease*[*OF assms(1)*, *of atms-of-msu* (*clauses S*)] **by** *blast*
then have (*dpll_W-mes* (*trail S'*) (*card* (*atms-of-msu* (*clauses S*))),
dpll_W-mes (*trail S*) (*card* (*atms-of-msu* (*clauses S*)))) \in *lex* {(*a*, *b*). *a* < *b*}
unfolding *lex-def* **by** *auto*
then show (*dpll_W-mes* (*trail S'*) (*card* (*atms-of-msu* (*clauses S'*))),
dpll_W-mes (*trail S*) (*card* (*atms-of-msu* (*clauses S*)))) \in *lex* {(*a*, *b*). *a* < *b*}
using *dpll_W-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 *dpll_W-wf*:
wf {(*S'*, *S*). *dpll_W-all-inv S* \wedge *dpll_W S S'*}
apply (*rule wf-wf-if-measure'*[*OF wf-lex-less*, *of - -*
 $\lambda S. \text{dpll}_W\text{-mes } (\text{trail } S) (\text{card } (\text{atms-of-msu } (\text{clauses } S)))$])
using *dpll_W-card-decrease'* **by** *fast*

lemma *dpll_W-tranclp-star-commute*:
 $\{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+ = \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{tranclp } \text{dpll}_W S S'\}$
(is ?A = ?B)
proof
{ **fix** *S S'*
assume (*S*, *S'*) \in ?A
then have (*S*, *S'*) \in ?B
by (*induct rule: trancl.induct, auto*)
}
then show ?A \subseteq ?B **by** *blast*
{ **fix** *S S'*

```

assume  $(S, S') \in ?B$ 
then have  $dpll_W^{++} S' S$  and  $dpll_W\text{-all-inv } S'$  by auto
then have  $(S, S') \in ?A$ 
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  $(S', S) \in \{a. \text{case } a \text{ of } (S', S) \Rightarrow dpll_W\text{-all-inv } S \wedge dpll_W S S'\}^+$  by blast
  moreover have  $dpll_W\text{-all-inv } S'$ 
    using rtrancpl-dpll_W-all-inv[OF trancpl-into-rtrancpl[OF trancpl-into-trancpl.hyps(1)]]
    trancpl-into-trancpl.prems by auto
  ultimately have  $(S'', S') \in \{(pa, p). dpll_W\text{-all-inv } p \wedge dpll_W p pa\}^+$ 
    using  $\langle dpll_W\text{-all-inv } S' \rangle$  trancpl-into-trancpl.hyps(3) by blast
  then show  $?case$ 
    using  $\langle (S', S) \in \{a. \text{case } a \text{ of } (S', S) \Rightarrow dpll_W\text{-all-inv } S \wedge dpll_W S S'\}^+ \rangle$  by auto
  qed
}
then show  $?B \subseteq ?A$  by blast
qed

```

lemma *dpll_W-wf-trancpl*: $wf \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W^{++} S S'\}$
unfolding *dpll_W-trancpl-star-commute[symmetric]* **by** (*simp add: dpll_W-wf wf-trancpl*)

lemma *dpll_W-wf-plus*:
shows $wf \{(S', ([], N)) \mid S'. dpll_W^{++} ([], N) S'\}$ (**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 $\forall S'. \neg dpll_W S S'$
shows *conclusive-dpll_W-state* S
proof –
have *vars*: $\forall s \in \text{atms-of-msu}(\text{clauses } S). s \in \text{atm-of ' lits-of } (\text{trail } S)$
proof (*rule ccontr*)
assume $\neg (\forall s \in \text{atms-of-msu}(\text{clauses } S). s \in \text{atm-of ' lits-of } (\text{trail } S))$
then obtain L **where**
 $L \in \text{atms-of-msu}(\text{clauses } S)$ **and**
 $L \notin \text{atm-of ' lits-of } (\text{trail } S)$ **by** *metis*
obtain L' **where** $L': \text{atm-of } L' = L$ **by** (*meson literal.sel(2)*)
then have *undefined-lit* $(\text{trail } S) L'$
 unfolding *Marked-Propagated-in-iff-in-lits-of* **by** (*metis L-notin-trail atm-of-uminus imageI*)
then show *False* **using** *dpll_W.decided assms(1) L-in-atms L'* **by** *blast*
qed
show $?thesis$
proof (*rule ccontr*)
assume *not-final*: $\neg ?thesis$
then have
 $\neg \text{trail } S \models_{asm} \text{clauses } S$ **and**
 $(\exists L \in \text{set } (\text{trail } S). \text{is-marked } L) \vee (\forall C \in \# \text{clauses } S. \neg \text{trail } S \models_{as} C \text{Not } C)$
 unfolding *conclusive-dpll_W-state-def* **by** *auto*
moreover {
 assume $\exists L \in \text{set } (\text{trail } S). \text{is-marked } L$
 then obtain $L M' M$ **where** $L: \text{backtrack-split } (\text{trail } S) = (M', L \# M)$

```

    using backtrack-split-some-is-marked-then-snd-has-hd by blast
  obtain D where D ∈ # clauses S and ¬ trail S ⊨a D
    using ⟨¬ trail S ⊨asm clauses S⟩ unfolding true-annots-def by auto
  then have ∀ s ∈ atms-of-ms {D}. s ∈ atm-of ' lits-of (trail S)
    using vars unfolding atms-of-ms-def by auto
  then have trail S ⊨as CNot D
    using all-variables-defined-not-imply-cnot[of D] ⟨¬ trail S ⊨a D⟩ 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 ⟨D ∈ # clauses S⟩ ⟨trail S ⊨as CNot D⟩ by blast
}
moreover {
  assume tr: ∀ C ∈ # clauses S. ¬ trail S ⊨as CNot C
  obtain C where C-in-cl: C ∈ # clauses S and trC: ¬ trail S ⊨a C
    using ⟨¬ trail S ⊨asm clauses S⟩ unfolding true-annots-def by auto
  have ∀ s ∈ atms-of-ms {C}. s ∈ atm-of ' lits-of (trail S)
    using vars ⟨C ∈ # clauses S⟩ unfolding atms-of-ms-def by auto
  then have trail S ⊨as CNot C
    by (meson C-in-cl tr trC all-variables-defined-not-imply-cnot)
  then have False using tr C-in-cl by auto
}
ultimately show False by blast
qed
qed

lemma dpllW-conclusive-state-correct:
  assumes dpllW** ([], N) (M, N) and conclusive-dpllW-state (M, N)
  shows M ⊨asm N ⟷ satisfiable (set-mset N) (is ?A ⟷ ?B)
proof
  let ?M' = lits-of M
  assume ?A
  then have ?M' ⊨sm N by (simp add: true-annots-true-cl)
  moreover have consistent-interp ?M'
    using rtrancpl-dpllW-inv-starting-from-0[OF assms(1)] by auto
  ultimately show ?B by auto
next
  assume ?B
  show ?A
  proof (rule ccontr)
    assume n: ¬ ?A
    have no-mark: ∀ L ∈ set M. ¬ is-marked L ∃ C ∈ # N. M ⊨as CNot C
      using n assms(2) unfolding conclusive-dpllW-state-def by auto
    moreover obtain D where DN: D ∈ # N and MD: M ⊨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 $state\text{-}eq_{NOT}\text{-}iff\text{-}eq[iff, simp]$: $dpll_{W-NOT}.state\text{-}eq_{NOT} S T \longleftrightarrow S = T$

```

unfolding  $dpll_W\text{-}NOT.state\text{-}eq_{NOT}\text{-}def$  by (cases S, cases T) auto

declare  $dpll_W\text{-}NOT.state\text{-}simp_{NOT}[simp\ del]$ 

lemma  $dpll_W\text{-}dpll_W\text{-}bj$ :
  assumes  $inv: dpll_W\text{-}all\text{-}inv\ S$  and  $dpll: dpll_W\ S\ T$ 
  shows  $dpll_W\text{-}NOT.dpll\text{-}bj\ S\ T$ 
  using  $dpll\ inv$ 
  apply (induction rule: dpll_W.induct)
    using  $dpll_W\text{-}NOT.dpll\text{-}bj.simps$  apply fastforce
    using  $dpll_W\text{-}NOT.bj\text{-}decide_{NOT}$  apply fastforce
  apply (frule dpll_W\text{-}NOT.backtrack.intros[of - - - -], simp-all)
  apply (rule dpll_W\text{-}NOT.dpll\text{-}bj.bj-backjump)
  apply (rule dpll_W\text{-}NOT.backtrack-is-backjump'',
    simp-all add: dpll_W\text{-}all\text{-}inv\text{-}def)
  done

lemma  $dpll_W\text{-}bj\text{-}dpll$ :
  assumes  $inv: dpll_W\text{-}all\text{-}inv\ S$  and  $dpll: dpll_W\text{-}NOT.dpll\text{-}bj\ S\ T$ 
  shows  $dpll_W\ S\ T$ 
  using  $dpll$ 
  apply (induction rule: dpll_W\text{-}NOT.dpll\text{-}bj.induct)
    apply (elim dpll_W\text{-}NOT.decide_{NOT}E, cases S)
    using decided apply fastforce
  apply (elim dpll_W\text{-}NOT.propagate_{NOT}E, cases S)
  using  $dpll_W.simps$  apply fastforce
  apply (elim dpll_W\text{-}NOT.backjumpE, cases S)
  by (simp add: dpll_W.simps dpll-with-backtrack.backtrack.simps)

lemma  $rtrancp\text{-}dpll_W\text{-}rtrancp\text{-}dpll_W\text{-}NOT$ :
  assumes  $dpll_W^{**}\ S\ T$  and  $dpll_W\text{-}all\text{-}inv\ S$ 
  shows  $dpll_W\text{-}NOT.dpll\text{-}bj^{**}\ S\ T$ 
  using assms apply (induction)
  apply simp
  by (auto intro: rtrancp\text{-}dpll_W\text{-}all\text{-}inv dpll_W\text{-}dpll_W\text{-}bj rtrancp.rtrancp\text{-}into\text{-}rtrancp)

lemma  $rtrancp\text{-}dpll\text{-}rtrancp\text{-}dpll_W$ :
  assumes  $dpll_W\text{-}NOT.dpll\text{-}bj^{**}\ S\ T$  and  $dpll_W\text{-}all\text{-}inv\ S$ 
  shows  $dpll_W^{**}\ S\ T$ 
  using assms apply (induction)
  apply simp
  by (auto intro: dpll_W\text{-}bj\text{-}dpll rtrancp.rtrancp\text{-}into\text{-}rtrancp rtrancp\text{-}dpll\text{-}rtrancp\text{-}dpll_W)

lemma  $dpll\text{-}conclusive\text{-}state\text{-}correctness$ :
  assumes  $dpll_W\text{-}NOT.dpll\text{-}bj^{**}\ (\[], N)\ (M, N)$  and  $conclusive\text{-}dpll_W\text{-}state\ (M, N)$ 
  shows  $M \models_{asm} N \longleftrightarrow \text{satisfiable}\ (\text{set-mset}\ N)$ 
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) rtrancp\text{-}dpll\text{-}rtrancp\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 function after reversing.

```

fun get-rev-level :: ('v, nat, 'a) marked-lits  $\Rightarrow$  nat  $\Rightarrow$  'v literal  $\Rightarrow$  nat where
  get-rev-level [] - = 0 |
  get-rev-level (Marked l level # Ls) n L =
    (if atm-of l = atm-of L then level else get-rev-level Ls level L) |
  get-rev-level (Propagated l - # Ls) n L =
    (if atm-of l = atm-of L then n else get-rev-level Ls n L)

```

abbreviation get-level $M\ L \equiv$ get-rev-level (rev M) 0 L

lemma get-rev-level-uminus[simp]: get-rev-level $M\ n(-L) =$ get-rev-level $M\ n\ L$
by (induct arbitrary: n rule: get-rev-level.induct) auto

lemma atm-of-notin-get-rev-level-eq-0[simp]:
assumes atm-of $L \notin$ atm-of ' lits-of M
shows get-rev-level $M\ n\ L = 0$
using assms **by** (induct M arbitrary: n rule: marked-lit-list-induct) auto

lemma get-rev-level-ge-0-atm-of-in:
assumes get-rev-level $M\ n\ L > n$
shows atm-of $L \in$ atm-of ' lits-of M
using assms **by** (induct M arbitrary: n rule: marked-lit-list-induct) fastforce+

In *get-rev-level* (resp. *get-level*), the beginning (resp. the end) can be skipped if the literal is not in the beginning (resp. the end).

lemma get-rev-level-skip[simp]:
assumes atm-of $L \notin$ atm-of ' lits-of M
shows get-rev-level ($M @$ Marked $K\ i \# M'$) $n\ L =$ get-rev-level (Marked $K\ i \# M'$) $i\ L$
using assms **by** (induct M arbitrary: $n\ i$ rule: marked-lit-list-induct) auto

lemma get-rev-level-notin-end[simp]:
assumes atm-of $L \notin$ atm-of ' lits-of M'
shows get-rev-level ($M @ M'$) $n\ L =$ get-rev-level $M\ n\ L$
using assms **by** (induct M arbitrary: n rule: marked-lit-list-induct) auto

If the literal is at the beginning, then the end can be skipped

lemma get-rev-level-skip-end[simp]:
assumes atm-of $L \in$ atm-of ' lits-of M
shows get-rev-level ($M @ M'$) $n\ L =$ get-rev-level $M\ n\ L$
using assms **by** (induct arbitrary: n rule: marked-lit-list-induct) auto

lemma get-level-skip-beginning:
assumes atm-of $L' \neq$ atm-of (lit-of K)
shows get-level ($K \# M$) $L' =$ get-level $M\ L'$
using assms **by** auto

lemma *get-level-skip-beginning-not-marked-rev*:
assumes *atm-of* $L \notin \text{atm-of 'lit-of '(set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows *get-level* ($M @ \text{rev } S$) $L = \text{get-level } M L$
using *assms* **by** (*induction* S *rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-beginning-not-marked[simp]*:
assumes *atm-of* $L \notin \text{atm-of 'lit-of '(set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows *get-level* ($M @ S$) $L = \text{get-level } M L$
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* $L \notin \text{atm-of 'lit-of '(set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows *get-rev-level* ($\text{rev } S @ \text{rev } M$) $0 L = \text{get-level } M L$
using *get-level-skip-beginning-not-marked-rev*[*of* L *rev* S M] *assms* **by** *auto*

lemma *get-level-skip-in-all-not-marked*:
fixes $M :: ('a, \text{nat}, 'b) \text{ marked-lit list}$ **and** $L :: 'a \text{ literal}$
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
and *atm-of* $L \in \text{atm-of 'lit-of '(set } M)$
shows *get-rev-level* $M \ n \ L = n$
using *assms* **by** (*induction* M *rule*: *marked-lit-list-induct*) *auto*

lemma *get-level-skip-all-not-marked[simp]*:
fixes M
defines $M' \equiv \text{rev } M$
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows *get-level* $M \ L = 0$
proof –
have $M: M = \text{rev } M'$
unfolding $M'\text{-def}$ **by** *auto*
show *?thesis*
using *assms* **unfolding** M **by** (*induction* M' *rule*: *marked-lit-list-induct*) *auto*
qed

abbreviation $M\text{Max } M \equiv \text{Max } (\text{set-mset } M)$

the $\{\#0::'a\# \}$ is there to ensures that the set is not empty.

definition *get-maximum-level* $:: ('a, \text{nat}, 'b) \text{ marked-lit list} \Rightarrow 'a \text{ literal multiset} \Rightarrow \text{nat}$
where
get-maximum-level $M \ D = M\text{Max } (\{\#0\# \} + \text{image-mset } (\text{get-level } M) \ D)$

lemma *get-maximum-level-ge-get-level*:
 $L \in \# \ D \Longrightarrow \text{get-maximum-level } M \ D \geq \text{get-level } M \ L$
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-empty[simp]*:
get-maximum-level $M \ \{\# \} = 0$
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-exists-lit-of-max-level*:
 $D \neq \{\# \} \Longrightarrow \exists L \in \# \ D. \text{get-level } M \ L = \text{get-maximum-level } M \ D$
unfolding *get-maximum-level-def*

```

apply (induct D)
apply simp
by (rename-tac D x, case-tac D = {#}) (auto simp add: max-def)

lemma get-maximum-level-empty-list[simp]:
  get-maximum-level [] D = 0
unfolding get-maximum-level-def by (simp add: image-constant-conv)

lemma get-maximum-level-single[simp]:
  get-maximum-level M {#L#} = get-level M L
unfolding get-maximum-level-def by simp

lemma get-maximum-level-plus:
  get-maximum-level M (D + D') = max (get-maximum-level M D) (get-maximum-level M D')
by (induct D) (auto simp add: get-maximum-level-def)

lemma get-maximum-level-exists-lit:
  assumes n: n > 0
  and max: get-maximum-level M D = n
  shows  $\exists L \in \#D. \text{get-level } M L = n$ 
proof –
  have f: finite (insert 0 (( $\lambda L. \text{get-level } M L$ ) ‘ set-mset D)) by auto
  then have n  $\in$  (( $\lambda L. \text{get-level } M L$ ) ‘ set-mset D)
    using n max Max-in[OF f] unfolding get-maximum-level-def by simp
  then show  $\exists L \in \# D. \text{get-level } M L = n$  by auto
qed

lemma get-maximum-level-skip-first[simp]:
  assumes atm-of L  $\notin$  atms-of D
  shows get-maximum-level (Propagated L C # M) D = get-maximum-level M D
  using asms 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: atms-of D  $\subseteq$  atm-of ‘lits-of H
  shows get-maximum-level (c @ Marked Kh i # H) D = get-maximum-level H D
proof –
  have (get-rev-level (rev H @ Marked Kh i # rev c) 0) ‘ set-mset D
    = (get-rev-level (rev H) 0) ‘ 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)+
  then show ?thesis using DH unfolding get-maximum-level-def by auto
qed

lemma get-maximum-level-D-single-propagated:
  get-maximum-level [Propagated x21 x22] D = 0
proof –
  have A: insert 0 (( $\lambda L. 0$ ) ‘ (set-mset D  $\cap$  {L. atm-of x21 = atm-of L}))
     $\cup$  (( $\lambda L. 0$ ) ‘ (set-mset D  $\cap$  {L. atm-of x21  $\neq$  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 $\text{get-maximum-level } M \ D = \text{get-maximum-level } (\text{Propagated } x21 \ x22 \ \# \ M) \ D$
proof –
have $A: (\text{get-rev-level } (\text{rev } M \ @ \ [\text{Propagated } x21 \ x22]) \ 0) \ ' \ \text{set-mset } D$
 $= (\text{get-rev-level } (\text{rev } M) \ 0) \ ' \ \text{set-mset } D$
using D **by** (*auto intro!*: *image-cong simp add: lits-of-def*)
show ?thesis **unfolding** *get-maximum-level-def* **by** (*auto simp: A*)
qed

lemma *get-maximum-level-skip-un-marked-not-present*:
assumes $\forall L \in \#D. \text{atm-of } L \in \text{atm-of 'lits-of } aa$ **and**
 $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\text{get-maximum-level } aa \ D = \text{get-maximum-level } (M \ @ \ aa) \ D$
using *assms* **by** (*induction M rule: marked-lit-list-induct*)
(*auto intro!*: *get-maximum-level-skip-notin[of D - @ aa] simp add: image-Un*)

fun *get-maximum-possible-level*:: ('b, nat, 'c) *marked-lit list* \Rightarrow nat **where**
get-maximum-possible-level [] = 0 |
get-maximum-possible-level (Marked $K \ i \ \# \ l$) = max i (*get-maximum-possible-level* l) |
get-maximum-possible-level (Propagated - - $\# \ l$) = *get-maximum-possible-level* l

lemma *get-maximum-possible-level-append[simp]*:
get-maximum-possible-level (M @ M')
= max (*get-maximum-possible-level* M) (*get-maximum-possible-level* M')
by (*induct M rule: marked-lit-list-induct*) *auto*

lemma *get-maximum-possible-level-rev[simp]*:
get-maximum-possible-level (rev M) = *get-maximum-possible-level* M
by (*induct M rule: marked-lit-list-induct*) *auto*

lemma *get-maximum-possible-level-ge-get-rev-level*:
max (*get-maximum-possible-level* M) $i \geq \text{get-rev-level } M \ i \ L$
by (*induct M arbitrary: i rule: marked-lit-list-induct*) (*auto simp add: le-max-iff-disj*)

lemma *get-maximum-possible-level-ge-get-level[simp]*:
get-maximum-possible-level M $\geq \text{get-level } M \ L$
using *get-maximum-possible-level-ge-get-rev-level* [*of rev - 0*] **by** *auto*

lemma *get-maximum-possible-level-ge-get-maximum-level[simp]*:
get-maximum-possible-level M $\geq \text{get-maximum-level } M \ D$
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
by (*induct A rule: marked-lit-list-induct*) *auto*

16.5.2 Properties about the levels

fun *get-all-levels-of-marked* :: ('b, 'a, 'c) *marked-lit list* \Rightarrow 'a *list* **where**
get-all-levels-of-marked [] = [] |
get-all-levels-of-marked (Marked l level # Ls) = level # *get-all-levels-of-marked* Ls |
get-all-levels-of-marked (Propagated - - # Ls) = *get-all-levels-of-marked* Ls

lemma *get-all-levels-of-marked-nil-iff-not-is-marked*:
get-all-levels-of-marked xs = [] \longleftrightarrow ($\forall x \in \text{set } xs. \neg \text{is-marked } x$)
using *assms* **by** (*induction* xs *rule*: *marked-lit-list-induct*) *auto*

lemma *get-all-levels-of-marked-cons*:
get-all-levels-of-marked (a # b) =
 (if *is-marked* a then [level-of a] else []) @ *get-all-levels-of-marked* b
by (*cases* a) *simp-all*

lemma *get-all-levels-of-marked-append[simp]*:
get-all-levels-of-marked (a @ b) = *get-all-levels-of-marked* a @ *get-all-levels-of-marked* b
by (*induct* a) (*simp-all* *add*: *get-all-levels-of-marked-cons*)

lemma *in-get-all-levels-of-marked-iff-decomp*:
 $i \in \text{set } (\text{get-all-levels-of-marked } M) \longleftrightarrow (\exists c \ K \ c'. M = c @ \text{Marked } K \ i \ \# \ c')$ (**is** ?A \longleftrightarrow ?B)
proof
assume ?B
then show ?A **by** *auto*
next
assume ?A
then show ?B
apply (*induction* M *rule*: *marked-lit-list-induct*)
apply *auto*[]
apply (*metis* *append-Cons* *append-Nil* *get-all-levels-of-marked.simps*(2) *set-ConsD*)
by (*metis* *append-Cons* *get-all-levels-of-marked.simps*(3))
qed

lemma *get-rev-level-less-max-get-all-levels-of-marked*:
get-rev-level M n L \leq Max (set (n # *get-all-levels-of-marked* M))
by (*induct* M *arbitrary*: n *rule*: *get-all-levels-of-marked.induct*)
 (*simp-all* *add*: *max.coboundedI2*)

lemma *get-rev-level-ge-min-get-all-levels-of-marked*:
assumes *atm-of* L \in *atm-of* ' lits-of M
shows *get-rev-level* M n L \geq Min (set (n # *get-all-levels-of-marked* M))
using *assms* **by** (*induct* M *arbitrary*: n *rule*: *get-all-levels-of-marked.induct*)
 (*auto* *simp* *add*: *min-le-iff-disj*)

lemma *get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked[simp]*:
get-all-levels-of-marked (rev M) = rev (*get-all-levels-of-marked* M)
by (*induct* M *rule*: *get-all-levels-of-marked.induct*)
 (*simp-all* *add*: *max.coboundedI2*)

lemma *get-maximum-possible-level-max-get-all-levels-of-marked*:
get-maximum-possible-level M = Max (insert 0 (set (*get-all-levels-of-marked* M)))
by (*induct* M *rule*: *marked-lit-list-induct*) (*auto* *simp*: *insert-commute*)

lemma *get-rev-level-in-levels-of-marked*:
get-rev-level M n L \in {0, n} \cup set (*get-all-levels-of-marked* M)

by (induction M arbitrary: n rule: marked-lit-list-induct) (force simp add: atm-of-eq-atm-of)+

lemma get-rev-level-in-atms-in-levels-of-marked:

atm-of L ∈ atm-of ‘ (lits-of M) \implies get-rev-level M n L ∈ {n} ∪ set (get-all-levels-of-marked M)

by (induction M arbitrary: n rule: marked-lit-list-induct) (auto simp add: atm-of-eq-atm-of)

lemma get-all-levels-of-marked-no-marked:

($\forall l \in \text{set } Ls. \neg \text{is-marked } l$) \longleftrightarrow get-all-levels-of-marked Ls = []

by (induction Ls) (auto simp add: get-all-levels-of-marked-cons)

lemma get-level-in-levels-of-marked:

get-level M L ∈ {0} ∪ set (get-all-levels-of-marked M)

using get-rev-level-in-levels-of-marked[of rev M 0 L] by auto

The zero is here to avoid empty-list issues with last:

lemma get-level-get-rev-level-get-all-levels-of-marked:

assumes atm-of L \notin atm-of ‘ (lits-of M)

shows get-level (K @ M) L = get-rev-level (rev K) (last (0 # get-all-levels-of-marked (rev M))) L

using assms

proof (induct M arbitrary: K)

case Nil

then show ?case by auto

next

case (Cons a M)

then have H: $\bigwedge K. \text{get-level } (K @ M) L$

= get-rev-level (rev K) (last (0 # get-all-levels-of-marked (rev M))) L

by auto

have get-level ((K @ [a]) @ M) L

= get-rev-level (a # rev K) (last (0 # get-all-levels-of-marked (rev M))) L

using H[of K @ [a]] by simp

then show ?case using Cons(2) by (cases a) auto

qed

lemma get-rev-level-can-skip-correctly-ordered:

assumes

no-dup M and

atm-of L \notin atm-of ‘ (lits-of M) and

get-all-levels-of-marked M = rev [Suc 0.. $\text{Suc } (\text{length } (\text{get-all-levels-of-marked } M))$]

shows get-rev-level (rev M @ K) 0 L = get-rev-level K (length (get-all-levels-of-marked M)) L

using assms

proof (induct M arbitrary: K rule: marked-lit-list-induct)

case nil

then show ?case by simp

next

case (marked L' i M K)

then have

i: i = Suc (length (get-all-levels-of-marked M)) and

get-all-levels-of-marked M = rev [Suc 0.. $\text{Suc } (\text{length } (\text{get-all-levels-of-marked } M))$]

by auto

then have get-rev-level (rev M @ (Marked L' i # K)) 0 L

= get-rev-level (Marked L' i # K) (length (get-all-levels-of-marked M)) L

using marked by auto

then show ?case using marked unfolding i by auto

```

next
  case (proped L' D M K)
  then have get-all-levels-of-marked M = rev [Suc 0..Suc (length (get-all-levels-of-marked M))]
    by auto
  then have get-rev-level (rev M @ (Propagated L' D # K)) 0 L
    = get-rev-level (Propagated L' D # K) (length (get-all-levels-of-marked M)) L
    using proped by auto
  then show ?case using proped by auto
qed

lemma get-level-skip-beginning-hd-get-all-levels-of-marked:
  assumes atm-of L  $\notin$  atm-of ' lits-of S
  and get-all-levels-of-marked S  $\neq$  []
  shows get-level (M@ S) L = get-rev-level (rev M) (hd (get-all-levels-of-marked S)) L
  using assms
proof (induction S arbitrary: M rule: marked-lit-list-induct)
  case nil
  then show ?case by (auto simp add: lits-of-def)
next
  case (marked K m) note notin = this(2)
  then show ?case by (auto simp add: lits-of-def)
next
  case (proped L l) note IH = this(1) and L = this(2) and neq = this(3)
  show ?case using IH[of M@[Propagated L l]] L neq by (auto simp add: atm-of-eq-atm-of)
qed

end
theory CDCL-W
imports 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]

```

17.1 The State

```

locale stateW =
  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 option 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

```

$remove_cls :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$
 $update_backtrack_lvl :: nat \Rightarrow 'st \Rightarrow 'st \text{ and}$
 $update_conflicting :: 'v \text{ clause option} \Rightarrow 'st \Rightarrow 'st \text{ and}$

$init_state :: 'v \text{ clauses} \Rightarrow 'st \text{ and}$
 $restart_state :: 'st \Rightarrow 'st$

assumes

$trail_cons_trail[simp]:$

$\bigwedge L \text{ st. } undefined_lit (trail \text{ st}) (lit_of \ L) \Longrightarrow trail (cons_trail \ L \ st) = L \# trail \ st \text{ and}$

$trail_tl_trail[simp]: \bigwedge st. trail (tl_trail \ st) = tl (trail \ st) \text{ and}$

$trail_add_init_cls[simp]:$

$\bigwedge st \ C. no_dup (trail \ st) \Longrightarrow trail (add_init_cls \ C \ st) = trail \ st \text{ and}$

$trail_add_learned_cls[simp]:$

$\bigwedge C \ st. no_dup (trail \ st) \Longrightarrow trail (add_learned_cls \ C \ st) = trail \ st \text{ and}$

$trail_remove_cls[simp]:$

$\bigwedge C \ st. trail (remove_cls \ C \ st) = trail \ st \text{ and}$

$trail_update_backtrack_lvl[simp]: \bigwedge st \ C. trail (update_backtrack_lvl \ C \ st) = trail \ st \text{ and}$

$trail_update_conflicting[simp]: \bigwedge C \ st. trail (update_conflicting \ C \ st) = trail \ st \text{ and}$

$init_clss_cons_trail[simp]:$

$\bigwedge M \ st. undefined_lit (trail \ st) (lit_of \ M) \Longrightarrow init_clss (cons_trail \ M \ st) = init_clss \ st$
and

$init_clss_tl_trail[simp]:$

$\bigwedge st. init_clss (tl_trail \ st) = init_clss \ st \text{ and}$

$init_clss_add_init_cls[simp]:$

$\bigwedge st \ C. no_dup (trail \ st) \Longrightarrow init_clss (add_init_cls \ C \ st) = \{\#C\# \} + init_clss \ st \text{ and}$

$init_clss_add_learned_cls[simp]:$

$\bigwedge C \ st. no_dup (trail \ st) \Longrightarrow init_clss (add_learned_cls \ C \ st) = init_clss \ st \text{ and}$

$init_clss_remove_cls[simp]:$

$\bigwedge C \ st. init_clss (remove_cls \ C \ st) = remove_mset \ C \ (init_clss \ st) \text{ and}$

$init_clss_update_backtrack_lvl[simp]:$

$\bigwedge st \ C. init_clss (update_backtrack_lvl \ C \ st) = init_clss \ st \text{ and}$

$init_clss_update_conflicting[simp]:$

$\bigwedge C \ st. init_clss (update_conflicting \ C \ st) = init_clss \ st \text{ and}$

$learned_clss_cons_trail[simp]:$

$\bigwedge M \ st. undefined_lit (trail \ st) (lit_of \ M) \Longrightarrow$
 $learned_clss (cons_trail \ M \ st) = learned_clss \ st \text{ and}$

$learned_clss_tl_trail[simp]:$

$\bigwedge st. learned_clss (tl_trail \ st) = learned_clss \ st \text{ and}$

$learned_clss_add_init_cls[simp]:$

$\bigwedge st \ C. no_dup (trail \ st) \Longrightarrow learned_clss (add_init_cls \ C \ st) = learned_clss \ st \text{ and}$

$learned_clss_add_learned_cls[simp]:$

$\bigwedge C \ st. no_dup (trail \ st) \Longrightarrow learned_clss (add_learned_cls \ C \ st) = \{\#C\# \} + learned_clss \ st$
and

$learned_clss_remove_cls[simp]:$

$\bigwedge C \ st. learned_clss (remove_cls \ C \ st) = remove_mset \ C \ (learned_clss \ st) \text{ and}$

$learned_clss_update_backtrack_lvl[simp]:$

$\bigwedge st \ C. learned_clss (update_backtrack_lvl \ C \ st) = learned_clss \ st \text{ and}$

$learned_clss_update_conflicting[simp]:$

$\bigwedge C \ st. learned_clss (update_conflicting \ C \ st) = learned_clss \ st \text{ and}$

$backtrack_lvl_cons_trail[simp]:$

$\bigwedge M \ st. undefined_lit (trail \ st) (lit_of \ M) \Longrightarrow$
 $backtrack_lvl (cons_trail \ M \ st) = backtrack_lvl \ st \text{ and}$

backtrack-lvl-tl-trail[simp]:

$\bigwedge st. \text{backtrack-lvl } (tl\text{-trail } st) = \text{backtrack-lvl } st \text{ and}$

backtrack-lvl-add-init-cls[simp]:

$\bigwedge st \ C. \text{no-dup } (trail \ st) \implies \text{backtrack-lvl } (add\text{-init-cls } C \ st) = \text{backtrack-lvl } st \text{ and}$

backtrack-lvl-add-learned-cls[simp]:

$\bigwedge C \ st. \text{no-dup } (trail \ st) \implies \text{backtrack-lvl } (add\text{-learned-cls } C \ st) = \text{backtrack-lvl } st \text{ and}$

backtrack-lvl-remove-cls[simp]:

$\bigwedge C \ st. \text{backtrack-lvl } (remove\text{-cls } C \ st) = \text{backtrack-lvl } st \text{ and}$

backtrack-lvl-update-backtrack-lvl[simp]:

$\bigwedge st \ k. \text{backtrack-lvl } (update\text{-backtrack-lvl } k \ st) = k \text{ and}$

backtrack-lvl-update-conflicting[simp]:

$\bigwedge C \ st. \text{backtrack-lvl } (update\text{-conflicting } C \ st) = \text{backtrack-lvl } st \text{ and}$

conflicting-cons-trail[simp]:

$\bigwedge M \ st. \text{undefined-lit } (trail \ st) \ (lit\text{-of } M) \implies$
 $\text{conflicting } (cons\text{-trail } M \ st) = \text{conflicting } st \text{ and}$

conflicting-tl-trail[simp]:

$\bigwedge st. \text{conflicting } (tl\text{-trail } st) = \text{conflicting } st \text{ and}$

conflicting-add-init-cls[simp]:

$\bigwedge st \ C. \text{no-dup } (trail \ st) \implies \text{conflicting } (add\text{-init-cls } C \ st) = \text{conflicting } st \text{ and}$

conflicting-add-learned-cls[simp]:

$\bigwedge C \ st. \text{no-dup } (trail \ st) \implies \text{conflicting } (add\text{-learned-cls } C \ st) = \text{conflicting } st \text{ and}$

conflicting-remove-cls[simp]:

$\bigwedge C \ st. \text{conflicting } (remove\text{-cls } C \ st) = \text{conflicting } st \text{ and}$

conflicting-update-backtrack-lvl[simp]:

$\bigwedge st \ C. \text{conflicting } (update\text{-backtrack-lvl } C \ st) = \text{conflicting } st \text{ and}$

conflicting-update-conflicting[simp]:

$\bigwedge C \ st. \text{conflicting } (update\text{-conflicting } C \ st) = C \text{ and}$

init-state-trail[simp]: $\bigwedge N. \text{trail } (init\text{-state } N) = [] \text{ and}$

init-state-clss[simp]: $\bigwedge N. \text{init-clss } (init\text{-state } N) = N \text{ and}$

init-state-learned-clss[simp]: $\bigwedge N. \text{learned-clss } (init\text{-state } N) = \{\#\} \text{ and}$

init-state-backtrack-lvl[simp]: $\bigwedge N. \text{backtrack-lvl } (init\text{-state } N) = 0 \text{ and}$

init-state-conflicting[simp]: $\bigwedge N. \text{conflicting } (init\text{-state } N) = \text{None} \text{ and}$

trail-restart-state[simp]: $\text{trail } (restart\text{-state } S) = [] \text{ and}$

init-clss-restart-state[simp]: $\text{init-clss } (restart\text{-state } S) = \text{init-clss } S \text{ and}$

learned-clss-restart-state[intro]: $\text{learned-clss } (restart\text{-state } S) \subseteq \# \text{learned-clss } S \text{ and}$

backtrack-lvl-restart-state[simp]: $\text{backtrack-lvl } (restart\text{-state } S) = 0 \text{ and}$

conflicting-restart-state[simp]: $\text{conflicting } (restart\text{-state } S) = \text{None}$

begin

definition *clauses* :: 'st \Rightarrow 'v clauses **where**

clauses *S* = *init-clss* *S* + *learned-clss* *S*

lemma

shows

clauses-cons-trail[simp]:

$\text{undefined-lit } (trail \ S) \ (lit\text{-of } M) \implies \text{clauses } (cons\text{-trail } M \ S) = \text{clauses } S \text{ and}$

clss-tl-trail[simp]: $\text{clauses } (tl\text{-trail } S) = \text{clauses } S \text{ and}$

clauses-add-learned-cls-unfolded:

$\text{no-dup } (trail \ S) \implies \text{clauses } (add\text{-learned-cls } U \ S) = \{\#U\#\} + \text{learned-clss } S + \text{init-clss } S$
and

clauses-add-init-cls[simp]:

$no_dup\ (trail\ S) \implies clauses\ (add_init_cls\ N\ S) = \{\#N\# \} + init_clss\ S + learned_clss\ S$ **and**
 $clauses_update_backtrack_lvl[simp]: clauses\ (update_backtrack_lvl\ k\ S) = clauses\ S$ **and**
 $clauses_update_conflicting[simp]: clauses\ (update_conflicting\ D\ S) = clauses\ S$ **and**
 $clauses_remove_cls[simp]:$
 $clauses\ (remove_cls\ C\ S) = clauses\ S - replicate_mset\ (count\ (clauses\ S)\ C)\ C$ **and**
 $clauses_add_learned_cls[simp]:$
 $no_dup\ (trail\ S) \implies clauses\ (add_learned_cls\ C\ S) = \{\#C\# \} + clauses\ S$ **and**
 $clauses_restart[simp]: clauses\ (restart_state\ S) \subseteq \# clauses\ S$ **and**
 $clauses_init_state[simp]: \bigwedge N. clauses\ (init_state\ N) = N$
prefer 9 using $clauses_def\ learned_clss_restart_state$ **apply** $fastforce$
by $(auto\ simp: ac_simps\ replicate_mset_plus\ clauses_def\ intro: multiset_eqI)$

abbreviation $state :: 'st \Rightarrow ('v, nat, 'v\ clause)\ marked_lit\ list \times 'v\ clauses \times 'v\ clauses$
 $\times nat \times 'v\ clause\ option$ **where**
 $state\ S \equiv (trail\ S, init_clss\ S, learned_clss\ S, backtrack_lvl\ S, conflicting\ S)$

abbreviation $incr_lvl :: 'st \Rightarrow 'st$ **where**
 $incr_lvl\ S \equiv update_backtrack_lvl\ (backtrack_lvl\ S + 1)\ S$

definition $state_eq :: 'st \Rightarrow 'st \Rightarrow bool$ (**infix** ~ 50) **where**
 $S \sim T \longleftrightarrow state\ S = state\ T$

lemma $state_eq_ref[simp, intro]:$
 $S \sim S$
unfolding $state_eq_def$ **by** $auto$

lemma $state_eq_sym:$
 $S \sim T \longleftrightarrow T \sim S$
unfolding $state_eq_def$ **by** $auto$

lemma $state_eq_trans:$
 $S \sim T \implies T \sim U \implies S \sim U$
unfolding $state_eq_def$ **by** $auto$

lemma
shows
 $state_eq_trail: S \sim T \implies trail\ S = trail\ T$ **and**
 $state_eq_init_clss: S \sim T \implies init_clss\ S = init_clss\ T$ **and**
 $state_eq_learned_clss: S \sim T \implies learned_clss\ S = learned_clss\ T$ **and**
 $state_eq_backtrack_lvl: S \sim T \implies backtrack_lvl\ S = backtrack_lvl\ T$ **and**
 $state_eq_conflicting: S \sim T \implies conflicting\ S = conflicting\ T$ **and**
 $state_eq_clauses: S \sim T \implies clauses\ S = clauses\ T$ **and**
 $state_eq_undefined_lit: S \sim T \implies undefined_lit\ (trail\ S)\ L = undefined_lit\ (trail\ T)\ L$
unfolding $state_eq_def\ clauses_def$ **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 $atms_of_ms_learned_clss_restart_state_in_atms_of_ms_learned_clssI[intro]:$
 $x \in atms_of_msu\ (learned_clss\ (restart_state\ S)) \implies x \in atms_of_msu\ (learned_clss\ S)$
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*]:
 $\text{trail } (\text{reduce-trail-to } [] S) = []$
apply (*induction []:: ('v, nat, 'v clause) marked-lits S rule: reduce-trail-to.induct*)
by (*metis length-0-conv reduce-trail-to-length-ne reduce-trail-to-nil*)

lemma *clauses-reduce-trail-to-nil*:
 $\text{clauses } (\text{reduce-trail-to } [] S) = \text{clauses } S$
proof (*induction [] S rule: reduce-trail-to.induct*)
case (*1 Sa*)
then have $\text{clauses } (\text{reduce-trail-to } ([]::'a \text{ list}) (\text{tl-trail } Sa)) = \text{clauses } (\text{tl-trail } Sa)$
 $\vee \text{trail } Sa = []$
by *fastforce*
then show $\text{clauses } (\text{reduce-trail-to } ([]::'a \text{ list}) Sa) = \text{clauses } Sa$
by (*metis (no-types) length-0-conv reduce-trail-to-eq-length clss-tl-trail reduce-trail-to-length-ne*)

qed

lemma *reduce-trail-to-skip-beginning*:
assumes $\text{trail } S = F' @ F$
shows $\text{trail } (\text{reduce-trail-to } F S) = F$
using *assms* **by** (*induction F' arbitrary: S*) (*auto simp: reduce-trail-to-length-ne*)

lemma *clauses-reduce-trail-to*[*simp*]:
 $\text{clauses } (\text{reduce-trail-to } F S) = \text{clauses } S$
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis clss-tl-trail reduce-trail-to.simps*)

lemma *conflicting-update-trail*[*simp*]:
 $\text{conflicting } (\text{reduce-trail-to } F S) = \text{conflicting } S$
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis conflicting-tl-trail reduce-trail-to.simps*)

lemma *backtrack-lvl-update-trial*[simp]:
backtrack-lvl (*reduce-trail-to* *F S*) = *backtrack-lvl S*
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis backtrack-lvl-tl-trail reduce-trail-to.simps*)

lemma *init-clss-update-trial*[simp]:
init-clss (*reduce-trail-to F S*) = *init-clss S*
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis init-clss-tl-trail reduce-trail-to.simps*)

lemma *learned-clss-update-trial*[simp]:
learned-clss (*reduce-trail-to F S*) = *learned-clss S*
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis learned-clss-tl-trail reduce-trail-to.simps*)

lemma *trail-eq-reduce-trail-to-eq*:
trail S = *trail T* \implies *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
apply (*induction F S arbitrary: T rule: reduce-trail-to.induct*)
by (*metis trail-tl-trail reduce-trail-to.simps*)

lemma *reduce-trail-to-state-eq_{NOT}-compatible*:
assumes *ST*: *S* \sim *T*
shows *reduce-trail-to F S* \sim *reduce-trail-to F T*
proof –
have *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
using *trail-eq-reduce-trail-to-eq*[of *S T F*] *ST* **by** *auto*
then show ?thesis **using** *ST* **by** (*auto simp del: state-simp simp: state-eq-def*)
qed

lemma *reduce-trail-to-trail-tl-trail-decomp*[simp]:
trail S = *F' @ Marked K d # F* \implies (*trail* (*reduce-trail-to F S*)) = *F*
apply (*rule reduce-trail-to-skip-beginning*[of - *F' @ Marked K d # []*])
by (*cases F'*) (*auto simp add:tl-append reduce-trail-to-skip-beginning*)

lemma *reduce-trail-to-add-learned-cls*[simp]:
no-dup (*trail S*) \implies
trail (*reduce-trail-to F* (*add-learned-cls C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-add-init-cls*[simp]:
no-dup (*trail S*) \implies
trail (*reduce-trail-to F* (*add-init-cls C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-remove-learned-cls*[simp]:
trail (*reduce-trail-to F* (*remove-cls C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-update-conflicting*[simp]:
trail (*reduce-trail-to F* (*update-conflicting C S*)) = *trail* (*reduce-trail-to F S*)
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-update-backtrack-lvl*[simp]:
trail (*reduce-trail-to F* (*update-backtrack-lvl C S*)) = *trail* (*reduce-trail-to F S*)

```

by (rule trail-eq-reduce-trail-to-eq) auto

lemma in-get-all-marked-decomposition-marked-or-empty:
  assumes  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
  shows  $a = [] \vee (\text{is-marked } (\text{hd } a))$ 
  using assms
proof (induct M arbitrary: a b)
  case Nil then show ?case by simp
next
  case (Cons m M)
  show ?case
  proof (cases m)
    case (Marked l mark)
    then show ?thesis using Cons by auto
  next
    case (Propagated l mark)
    then show ?thesis using Cons by (cases get-all-marked-decomposition M) force+
  qed
qed

lemma in-get-all-marked-decomposition-trail-update-trail[simp]:
  assumes  $H: (L \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
  shows  $\text{trail } (\text{reduce-trail-to } M1 S) = M1$ 
proof -
  obtain K mark where
    L:  $L = \text{Marked } K \text{ mark}$ 
  using H by (cases L) (auto dest!: in-get-all-marked-decomposition-marked-or-empty)
  obtain c where
    tr-S:  $\text{trail } S = c @ M2 @ L \# M1$ 
  using H by auto
  show ?thesis
  by (rule reduce-trail-to-trail-tl-trail-decomp[of - c @ M2 K mark])
  (auto simp: tr-S L)
qed

fun append-trail where
  append-trail [] S = S |
  append-trail (L # M) S = append-trail M (cons-trail L S)

lemma trail-append-trail:
  no-dup (M @ trail S)  $\implies$  trail (append-trail M S) = rev M @ trail S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

lemma init-clss-append-trail:
  no-dup (M @ trail S)  $\implies$  init-clss (append-trail M S) = init-clss S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

lemma learned-clss-append-trail:
  no-dup (M @ trail S)  $\implies$  learned-clss (append-trail M S) = learned-clss S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

lemma conflicting-append-trail:
  no-dup (M @ trail S)  $\implies$  conflicting (append-trail M S) = conflicting S
  by (induction M arbitrary: S) (auto simp: defined-lit-map)

```

lemma *backtrack-lvl-append-trail*:

no-dup ($M @ \text{trail } S$) $\implies \text{backtrack-lvl } (\text{append-trail } M S) = \text{backtrack-lvl } S$
by (*induction* M *arbitrary*: S) (*auto simp*: *defined-lit-map*)

lemma *clauses-append-trail*:

no-dup ($M @ \text{trail } S$) $\implies \text{clauses } (\text{append-trail } M S) = \text{clauses } S$
by (*induction* M *arbitrary*: S) (*auto simp*: *defined-lit-map*)

lemmas *state-access-simp* =

trail-append-trail init-clss-append-trail learned-clss-append-trail backtrack-lvl-append-trail
clauses-append-trail conflicting-append-trail

This function is useful for proofs to speak of a global trail change, but is a bad for programs and code in general.

fun *delete-trail-and-rebuild* **where**

delete-trail-and-rebuild $M S = \text{append-trail } (\text{rev } M) (\text{reduce-trail-to } ([:: 'v \text{ list}] S)$

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 =
state_W *trail* *init-clss* *learned-clss* *backtrack-lvl* *conflicting* *cons-trail* *tl-trail* *add-init-clss*
add-learned-clss *remove-clss* *update-backtrack-lvl* *update-conflicting* *init-state*
restart-state

for

trail :: $'st \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ marked-lits}$ **and**
init-clss :: $'st \Rightarrow 'v \text{ clauses}$ **and**
learned-clss :: $'st \Rightarrow 'v \text{ clauses}$ **and**
backtrack-lvl :: $'st \Rightarrow \text{nat}$ **and**
conflicting :: $'st \Rightarrow 'v \text{ clause option}$ **and**

cons-trail :: $('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
add-init-clss :: $'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ **and**
add-learned-clss :: $'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ **and**
remove-clss :: $'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ **and**
update-backtrack-lvl :: $\text{nat} \Rightarrow 'st \Rightarrow 'st$ **and**
update-conflicting :: $'v \text{ clause option} \Rightarrow 'st \Rightarrow 'st$ **and**

init-state :: $'v \text{ clauses} \Rightarrow 'st$ **and**
restart-state :: $'st \Rightarrow 'st$

begin

inductive *propagate* :: $'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**

propagate-rule[*intro*]:

state $S = (M, N, U, k, \text{None}) \implies C + \{\#L\# \} \in \# \text{ clauses } S \implies M \models_{\text{as}} C \text{Not } C$
 $\implies \text{undefined-lit } (\text{trail } S) L$
 $\implies T \sim \text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S$
 $\implies \text{propagate } S T$

inductive-cases *propagateE*[*elim*]: *propagate* $S T$

thm *propagateE*

inductive *conflict* :: 'st \Rightarrow 'st \Rightarrow bool **where**

conflict-rule[intro]: state $S = (M, N, U, k, \text{None}) \Rightarrow D \in \# \text{ clauses } S \Rightarrow M \models_{\text{as}} \text{CNot } D$
 $\Rightarrow T \sim \text{update-conflicting } (\text{Some } D) S$
 $\Rightarrow \text{conflict } S T$

inductive-cases *conflictE*[elim]: *conflict* $S S'$

inductive *backtrack* :: 'st \Rightarrow 'st \Rightarrow bool **where**

backtrack-rule[intro]: state $S = (M, N, U, k, \text{Some } (D + \{\#L\#}))$
 $\Rightarrow (\text{Marked } K (i+1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$
 $\Rightarrow \text{get-level } M L = k$
 $\Rightarrow \text{get-level } M L = \text{get-maximum-level } M (D + \{\#L\#})$
 $\Rightarrow \text{get-maximum-level } M D = i$
 $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\#}))$
 $\quad (\text{reduce-trail-to } M1$
 $\quad \quad (\text{add-learned-cls } (D + \{\#L\#}))$
 $\quad \quad (\text{update-backtrack-lvl } i$
 $\quad \quad \quad (\text{update-conflicting } \text{None } S)))$
 $\Rightarrow \text{backtrack } S T$

inductive-cases *backtrackE*[elim]: *backtrack* $S S'$

thm *backtrackE*

inductive *decide* :: 'st \Rightarrow 'st \Rightarrow bool **where**

decide-rule[intro]: state $S = (M, N, U, k, \text{None})$
 $\Rightarrow \text{undefined-lit } M L \Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$
 $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L (k+1)) (\text{incr-lvl } S)$
 $\Rightarrow \text{decide } S T$

inductive-cases *decideE*[elim]: *decide* $S S'$

thm *decideE*

inductive *skip* :: 'st \Rightarrow 'st \Rightarrow bool **where**

skip-rule[intro]: state $S = (\text{Propagated } L C' \# M, N, U, k, \text{Some } D) \Rightarrow -L \notin \# D \Rightarrow D \neq \{\#\}$
 $\Rightarrow T \sim \text{tl-trail } S$
 $\Rightarrow \text{skip } S T$

inductive-cases *skipE*[elim]: *skip* $S S'$

thm *skipE*

get-maximum-level $(\text{Propagated } L (C + \{\#L\#}) \# M) D = k \vee k = 0$ is equivalent to
get-maximum-level $(\text{Propagated } L (C + \{\#L\#}) \# M) D = k$

inductive *resolve* :: 'st \Rightarrow 'st \Rightarrow bool **where**

resolve-rule[intro]:
state $S = (\text{Propagated } L (C + \{\#L\#}) \# M, N, U, k, \text{Some } (D + \{\#-L\#}))$
 $\Rightarrow \text{get-maximum-level } (\text{Propagated } L (C + \{\#L\#}) \# M) D = k$
 $\Rightarrow T \sim \text{update-conflicting } (\text{Some } (D \# \cup C)) (\text{tl-trail } S)$
 $\Rightarrow \text{resolve } S T$

inductive-cases *resolveE*[elim]: *resolve* $S S'$

thm *resolveE*

inductive *restart* :: 'st \Rightarrow 'st \Rightarrow bool **where**

restart: state $S = (M, N, U, k, \text{None}) \Rightarrow \neg M \models_{\text{asm}} \text{clauses } S$
 $\Rightarrow T \sim \text{restart-state } S$
 $\Rightarrow \text{restart } S T$

inductive-cases *restartE*[elim]: *restart* $S T$

thm *restartE*

We add the condition $C \notin \# \text{init-clss } S$, to maintain consistency even without the strategy.

inductive *forget* :: 'st \Rightarrow 'st \Rightarrow bool **where**
forget-rule: state $S = (M, N, \{\#C\# \} + U, k, \text{None})$
 $\Rightarrow \neg M \models_{asm} \text{clauses } S$
 $\Rightarrow C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$
 $\Rightarrow C \notin \# \text{init-clss } S$
 $\Rightarrow C \in \# \text{learned-clss } S$
 $\Rightarrow T \sim \text{remove-cl } C \ S$
 $\Rightarrow \text{forget } S \ T$

inductive-cases *forgetE*[elim]: *forget* $S \ T$

inductive *cdcl_W-rf* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S :: 'st$ **where**
restart: *restart* $S \ T \Rightarrow \text{cdcl}_W\text{-rf } S \ T$ |
forget: *forget* $S \ T \Rightarrow \text{cdcl}_W\text{-rf } S \ T$

inductive *cdcl_W-bj* :: 'st \Rightarrow 'st \Rightarrow bool **where**
skip[intro]: *skip* $S \ S' \Rightarrow \text{cdcl}_W\text{-bj } S \ S'$ |
resolve[intro]: *resolve* $S \ S' \Rightarrow \text{cdcl}_W\text{-bj } S \ S'$ |
backtrack[intro]: *backtrack* $S \ S' \Rightarrow \text{cdcl}_W\text{-bj } S \ S'$

inductive-cases *cdcl_W-bjE*: *cdcl_W-bj* $S \ T$

inductive *cdcl_W-o*: 'st \Rightarrow 'st \Rightarrow bool **for** $S :: 'st$ **where**
decide[intro]: *decide* $S \ S' \Rightarrow \text{cdcl}_W\text{-o } S \ S'$ |
bj[intro]: *cdcl_W-bj* $S \ S' \Rightarrow \text{cdcl}_W\text{-o } S \ S'$

inductive *cdcl_W* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S :: 'st$ **where**
propagate: *propagate* $S \ S' \Rightarrow \text{cdcl}_W \ S \ S'$ |
conflict: *conflict* $S \ S' \Rightarrow \text{cdcl}_W \ S \ S'$ |
other: *cdcl_W-o* $S \ S' \Rightarrow \text{cdcl}_W \ S \ S'$ |
rf: *cdcl_W-rf* $S \ S' \Rightarrow \text{cdcl}_W \ S \ S'$

lemma *rtrancp-propagate-is-rtrancp-cdcl_W*:
*propagate*** $S \ S' \Rightarrow \text{cdcl}_W$ ** $S \ S'$
by (induction rule: *rtrancp-induct*) (fastforce dest!: *propagate*)+

lemma *cdcl_W-all-rules-induct*[consumes 1, case-names *propagate conflict forget restart decide skip resolve backtrack*]:

fixes $S :: 'st$

assumes

cdcl_W: *cdcl_W* $S \ S'$ **and**

propagate: $\bigwedge T. \text{propagate } S \ T \Rightarrow P \ S \ T$ **and**

conflict: $\bigwedge T. \text{conflict } S \ T \Rightarrow P \ S \ T$ **and**

forget: $\bigwedge T. \text{forget } S \ T \Rightarrow P \ S \ T$ **and**

restart: $\bigwedge T. \text{restart } S \ T \Rightarrow P \ S \ T$ **and**

decide: $\bigwedge T. \text{decide } S \ T \Rightarrow P \ S \ T$ **and**

skip: $\bigwedge T. \text{skip } S \ T \Rightarrow P \ S \ T$ **and**

resolve: $\bigwedge T. \text{resolve } S \ T \Rightarrow P \ S \ T$ **and**

backtrack: $\bigwedge T. \text{backtrack } S \ T \Rightarrow P \ S \ T$

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: cdclW-o.induct)
    case (decide U)
    then show ?case using assms(6) by auto
  next
    case (bj S')
    then show ?case using assms(7-9) by (induction rule: cdclW-bj.induct) auto
  qed
next
case (rf S')
then show ?case
  by (induct rule: cdclW-rf.induct) (fast dest: forget restart)+
qed

```

lemma *cdcl_W-all-induct*[consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack]:

fixes $S :: 'st$

assumes

cdcl_W: *cdcl_W S S' and*

propagateH: $\bigwedge C L T. C + \{\#L\# \} \in \# \text{ clauses } S \implies \text{trail } S \models_{as} CNot C$

$\implies \text{undefined-lit } (\text{trail } S) L \implies \text{conflicting } S = None$

$\implies T \sim \text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S$

$\implies P S T$ **and**

conflictH: $\bigwedge D T. D \in \# \text{ clauses } S \implies \text{conflicting } S = None \implies \text{trail } S \models_{as} CNot D$

$\implies T \sim \text{update-conflicting } (Some D) S$

$\implies P S T$ **and**

forgetH: $\bigwedge C T. \neg \text{trail } S \models_{asm} \text{clauses } S$

$\implies C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$

$\implies C \notin \# \text{ init-clss } S$

$\implies C \in \# \text{ learned-clss } S$

$\implies \text{conflicting } S = None$

$\implies T \sim \text{remove-cl } C S$

$\implies P S T$ **and**

restartH: $\bigwedge T. \neg \text{trail } S \models_{asm} \text{clauses } S$

$\implies \text{conflicting } S = None$

$\implies T \sim \text{restart-state } S$

$\implies P S T$ **and**

decideH: $\bigwedge L T. \text{conflicting } S = None \implies \text{undefined-lit } (\text{trail } S) L$

$\implies \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$

$\implies T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$

$\implies P S T$ **and**

skipH: $\bigwedge L C' M D T. \text{trail } S = \text{Propagated } L C' \# M$

$\implies \text{conflicting } S = Some D \implies -L \notin \# D \implies D \neq \{\#\}$

$\implies T \sim \text{tl-trail } S$

$\implies P S T$ **and**

resolveH: $\bigwedge L C M D T.$

$\text{trail } S = \text{Propagated } L ((C + \{\#L\# \})) \# M$

$\implies \text{conflicting } S = Some (D + \{\#-L\# \})$

$\implies \text{get-maximum-level } (\text{Propagated } L (C + \{\#L\# \}) \# M) D = \text{backtrack-lvl } S$

```

     $\Rightarrow T \sim (\text{update-conflicting } (\text{Some } (D \# \cup C)) (\text{tl-trail } S))$ 
     $\Rightarrow P \ S \ T$  and
    backtrackH:  $\bigwedge K \ i \ M1 \ M2 \ L \ D \ T.$ 
    (Marked  $K \ (\text{Suc } i) \ \# \ M1, \ M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
     $\Rightarrow \text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S$ 
     $\Rightarrow \text{conflicting } S = \text{Some } (D + \{\#L\# \})$ 
     $\Rightarrow \text{get-maximum-level } (\text{trail } S) \ (D + \{\#L\# \}) = \text{get-level } (\text{trail } S) \ L$ 
     $\Rightarrow \text{get-maximum-level } (\text{trail } S) \ D \equiv i$ 
     $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L \ (D + \{\#L\# \}))$ 
    (reduce-trail-to  $M1$ 
    (add-learned-cls  $(D + \{\#L\# \})$ 
    (update-backtrack-lvl  $i$ 
    (update-conflicting  $\text{None } S$ ))))
     $\Rightarrow P \ S \ T$ 
  shows  $P \ S \ S'$ 
  using cdclW
proof (induct  $S \ S'$  rule: cdclW-all-rules-induct)
  case (propagate  $S'$ )
  then show ?case by (elim propagateE) (frule propagateH; simp)
next
  case (conflict  $S'$ )
  then show ?case by (elim conflictE) (frule conflictH; simp)
next
  case (restart  $S'$ )
  then show ?case by (elim restartE) (frule restartH; simp)
next
  case (decide  $T$ )
  then show ?case by (elim decideE) (frule decideH; simp)
next
  case (backtrack  $S'$ )
  then show ?case by (elim backtrackE) (frule backtrackH; simp del: state-simp add: state-eq-def)
next
  case (forget  $S'$ )
  then show ?case using forgetH by auto
next
  case (skip  $S'$ )
  then show ?case using skipH by auto
next
  case (resolve  $S'$ )
  then show ?case by (elim resolveE) (frule resolveH; simp)
qed

```

lemma cdcl_W-o-induct[consumes 1, case-names decide skip resolve backtrack]:

```

  fixes  $S :: 'st$ 
  assumes cdclW: cdclW-o  $S \ T$  and
  decideH:  $\bigwedge L \ T. \text{conflicting } S = \text{None} \Rightarrow \text{undefined-lit } (\text{trail } S) \ L$ 
     $\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$ 
     $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L \ (\text{backtrack-lvl } S + 1)) \ (\text{incr-lvl } S)$ 
     $\Rightarrow P \ S \ T$  and
  skipH:  $\bigwedge L \ C' \ M \ D \ T. \text{trail } S = \text{Propagated } L \ C' \ \# \ M$ 
     $\Rightarrow \text{conflicting } S = \text{Some } D \Rightarrow -L \notin \# \ D \Rightarrow D \neq \{\#\}$ 
     $\Rightarrow T \sim \text{tl-trail } S$ 
     $\Rightarrow P \ S \ T$  and
  resolveH:  $\bigwedge L \ C \ M \ D \ T.$ 
    trail  $S = \text{Propagated } L \ ( (C + \{\#L\# \})) \ \# \ M$ 

```

```

     $\Rightarrow$  conflicting  $S = \text{Some } (D + \{\#-L\# \})$ 
     $\Rightarrow$  get-maximum-level (Propagated  $L (C + \{\#L\# \}) \# M$ )  $D = \text{backtrack-lvl } S$ 
     $\Rightarrow T \sim \text{update-conflicting } (\text{Some } (D \# \cup C)) (\text{tl-trail } S)$ 
     $\Rightarrow P S T$  and
backtrackH:  $\bigwedge K i M1 M2 L D T.$ 
  (Marked  $K (\text{Suc } i) \# M1, M2 \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ )
   $\Rightarrow$  get-level (trail  $S$ )  $L = \text{backtrack-lvl } S$ 
   $\Rightarrow$  conflicting  $S = \text{Some } (D + \{\#L\# \})$ 
   $\Rightarrow$  get-level (trail  $S$ )  $L = \text{get-maximum-level } (\text{trail } S) (D + \{\#L\# \})$ 
   $\Rightarrow$  get-maximum-level (trail  $S$ )  $D \equiv i$ 
   $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$ 
    (reduce-trail-to  $M1$ 
      (add-learned-cls  $(D + \{\#L\# \})$ 
        (update-backtrack-lvl  $i$ 
          (update-conflicting None  $S$ ))))
   $\Rightarrow P S T$ 
shows  $P S T$ 
using  $\text{cdcl}_W$  apply (induct  $T$  rule:  $\text{cdcl}_W\text{-o.induct}$ )
  using  $\text{assms}(2)$  apply  $\text{auto}[1]$ 
apply (elim  $\text{cdcl}_W\text{-bjE}$  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 $\text{cdcl}_W\text{-o.induct}$

lemma $\text{cdcl}_W\text{-o-all-rules-induct}[\text{consumes } 1, \text{case-names decide backtrack skip resolve}]$:

```

fixes  $S T :: 'st$ 
assumes
   $\text{cdcl}_W\text{-o } S T$  and
   $\bigwedge T. \text{decide } S T \Rightarrow P S T$  and
   $\bigwedge T. \text{backtrack } S T \Rightarrow P S T$  and
   $\bigwedge T. \text{skip } S T \Rightarrow P S T$  and
   $\bigwedge T. \text{resolve } S T \Rightarrow P S T$ 

```

shows $P S T$

using assms **by** (induct T rule: $\text{cdcl}_W\text{-o.induct}$) (auto simp: $\text{cdcl}_W\text{-bj.simps}$)

lemma $\text{cdcl}_W\text{-o-rule-cases}[\text{consumes } 1, \text{case-names decide backtrack skip resolve}]$:

```

fixes  $S T :: 'st$ 
assumes
   $\text{cdcl}_W\text{-o } S T$  and
   $\text{decide } S T \Rightarrow P$  and
   $\text{backtrack } S T \Rightarrow P$  and
   $\text{skip } S T \Rightarrow P$  and
   $\text{resolve } S T \Rightarrow P$ 

```

shows P

using assms **by** (auto simp: $\text{cdcl}_W\text{-o.simps}$ $\text{cdcl}_W\text{-bj.simps}$)

17.4 Invariants

17.4.1 Properties of the trail

We here establish that: * the marks are exactly 1..k where k is the level * the consistency of the trail * the fact that there is no duplicate in the trail.

lemma $\text{backtrack-lit-skipped}$:

assumes L : $\text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S$
and $M1$: $(\text{Marked } K \ (i + 1) \ \# \ M1, \ M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$
and no-dup : $\text{no-dup } (\text{trail } S)$
and bt-l : $\text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$
and order : $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } ([1..<(1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S)))])$
shows $\text{atm-of } L \notin \text{atm-of ' lits-of } M1$
proof
let $?M = \text{trail } S$
assume $L\text{-in-}M1$: $\text{atm-of } L \in \text{atm-of ' lits-of } M1$
obtain c **where** Mc : $\text{trail } S = c \ @ \ M2 \ @ \ \text{Marked } K \ (i + 1) \ \# \ M1$ **using** $M1$ **by** blast
have $\text{atm-of } L \notin \text{atm-of ' lits-of } c$
using $L\text{-in-}M1$ no-dup $\text{mk-disjoint-insert}$ **unfolding** Mc lits-of-def **by** force
have $g\text{-}M\text{-eq-}g\text{-}M1$: $\text{get-level } ?M \ L = \text{get-level } M1 \ L$
using $L\text{-in-}M1$ **unfolding** Mc **by** auto
have g : $\text{get-all-levels-of-marked } M1 = \text{rev } [1..<\text{Suc } i]$
using order **unfolding** Mc
by $(\text{auto simp del: upt-simps dest!: append-cons-eq-upt-length-i simp add: rev-swap[symmetric]})$
then have $\text{Max } (\text{set } (0 \ \# \ \text{get-all-levels-of-marked } (\text{rev } M1))) < \text{Suc } i$ **by** auto
then have $\text{get-level } M1 \ L < \text{Suc } i$
using $\text{get-rev-level-less-max-get-all-levels-of-marked[of rev } M1 \ 0 \ L]$ **by** linarith
moreover have $\text{Suc } i \leq \text{backtrack-lvl } S$ **using** bt-l **by** $(\text{simp add: } Mc \ g)$
ultimately show False **using** $L \ g\text{-}M\text{-eq-}g\text{-}M1$ **by** auto
qed

lemma $\text{cdcl}_W\text{-distinctinv-1}$:

assumes
 $\text{cdcl}_W \ S \ S'$ **and**
 $\text{no-dup } (\text{trail } S)$ **and**
 $\text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ **and**
 $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } [1..<1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$
shows $\text{no-dup } (\text{trail } S')$
using assms
proof $(\text{induct rule: } \text{cdcl}_W\text{-all-induct})$
case $(\text{backtrack } K \ i \ M1 \ M2 \ L \ D \ T)$ **note** $\text{decomp} = \text{this}(1)$ **and** $L = \text{this}(2)$ **and** $T = \text{this}(6)$ **and** $n\text{-d} = \text{this}(7)$
obtain c **where** Mc : $\text{trail } S = c \ @ \ M2 \ @ \ \text{Marked } K \ (i + 1) \ \# \ M1$
using decomp **by** auto
have $\text{no-dup } (M2 \ @ \ \text{Marked } K \ (i + 1) \ \# \ M1)$
using $Mc \ n\text{-d}$ **by** fastforce
moreover have $\text{atm-of } L \notin (\lambda l. \text{atm-of } (\text{lit-of } l)) \ \text{' set } M1$
using $\text{backtrack-lit-skipped[of } S \ L \ K \ i \ M1 \ M2]$ $L \ \text{decomp}$ backtrack.premis
by $(\text{fastforce simp: lits-of-def})$
moreover then have $\text{undefined-lit } M1 \ L$
by $(\text{simp add: defined-lit-map})$
ultimately show $?case$ **using** $\text{decomp } T \ n\text{-d}$ **by** simp
qed $(\text{auto simp: defined-lit-map})$

lemma $\text{cdcl}_W\text{-consistent-inv-2}$:

assumes
 $\text{cdcl}_W \ S \ S'$ **and**
 $\text{no-dup } (\text{trail } S)$ **and**
 $\text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ **and**
 $\text{get-all-levels-of-marked } (\text{trail } S) = \text{rev } [1..<1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$

shows *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 S L K i M1 M2] backtrack(2,7,8,9) decomp*
by (*fastforce simp add: lits-of-def*)
moreover **then** **have** *undefined-lit M1 L*
by (*simp add: defined-lit-map*)
moreover **then** **have** *no-dup (trail T)*
using *T decomp n-d* **by** (*auto simp: defined-lit-map M*)
ultimately **show** *?case*
using *T n-d unfolding M* **by** (*auto dest!: append-cons-eq-upt-length simp del: upt-simps*)
qed *auto*

lemma *cdcl_W-rf-bt*:

assumes
cdcl_W-rf S S' **and**
backtrack-lvl S = length (get-all-levels-of-marked (trail S)) and
get-all-levels-of-marked (trail S) = rev [1.. $(1 + \text{length (get-all-levels-of-marked (trail S))})$])])
shows *backtrack-lvl S' = length (get-all-levels-of-marked (trail S'))*
using *assms* **by** (*induct rule: cdcl_W-rf.induct*) *auto*

lemma *cdcl_W-bt*:

assumes
cdcl_W 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: cdcl_W.induct*) (*auto simp add: cdcl_W-o-bt cdcl_W-rf-bt*)

lemma *cdcl_W-bt-level'*:

assumes
cdcl_W 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 } (\text{get-all-levels-of-marked } (\text{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.. $(1 + \text{length } (\text{get-all-levels-of-marked } ?M))]$ 
  using decide.prem by simp
have k: ?k = length (get-all-levels-of-marked ?M)
  using decide.prem 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.. $(1 + \text{length } (\text{get-all-levels-of-marked } ?M))]$ 
  using H by auto
moreover have ... = rev [Suc 0.. $(1 + \text{length } (\text{get-all-levels-of-marked } ?M))]$ 
  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 S L K i M1 M2] backtrack(2,7,8,9) decomp
  by (fastforce simp add: lits-of-def)
moreover then have undefined-lit M1 L
  by (simp add: defined-lit-map)
then have [simp]: trail T = Propagated L (D + {#L#}) # M1
  using T decomp n-d by auto
obtain c where M: trail S = c @ M2 @ Marked K (i + 1) # M1 using decomp by auto
have get-all-levels-of-marked (rev (trail S))
  = [Suc 0.. $(2 + \text{length } (\text{get-all-levels-of-marked } c) + (\text{length } (\text{get-all-levels-of-marked } M2) + \text{length } (\text{get-all-levels-of-marked } M1)))]$ 
  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 cdclW-consistent-inv:
  fixes S S' :: 'st
  assumes
    cdclW S S' and
    cdclW-M-level-inv S
  shows cdclW-M-level-inv S'
  using assms cdclW-consistent-inv-2 cdclW-distinctinv-1 cdclW-bt cdclW-bt-level'
  unfolding cdclW-M-level-inv-def by meson+

lemma rtrancpl-cdclW-consistent-inv:
  assumes cdclW** S S'
  and cdclW-M-level-inv S
  shows cdclW-M-level-inv S'
  using assms by (induct rule: rtrancpl-induct)
  (auto intro: cdclW-consistent-inv)

lemma trancpl-cdclW-consistent-inv:
  assumes cdclW++ S S'
  and cdclW-M-level-inv S
  shows cdclW-M-level-inv S'
  using assms by (induct rule: trancpl-induct)
  (auto intro: cdclW-consistent-inv)

lemma cdclW-M-level-inv-S0-cdclW[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 (trail S) L ≤ backtrack-lvl S
proof –
  have get-all-levels-of-marked (trail S) = rev [1..L + 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 rev (trail S) 0 L]
    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  $i+1 \in \text{set } (\text{get-all-levels-of-marked } (\text{trail } S))$ 
    using i-S by auto

  then obtain c K c' where tr-S: trail S = c @ Marked K (i + 1) # c'
    using in-get-all-levels-of-marked-iff-decomp[of i+1 trail S] by auto

  obtain M1 M2 where  $(\text{Marked } K (i + 1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
    unfolding tr-S apply (induct c rule: marked-lit-list-induct)

```

```

    apply auto[2]
  apply (rename-tac L m xs,
    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

We generalise the induction principle defined previously: the induction case for *backtrack* now includes the assumption that *undefined-lit M1 L*. This helps the simplifier and thus the automation.

lemma *backtrack-induction-lev*[consumes 1, case-names *M-devel-inv backtrack*]:

```

assumes
  bt: backtrack S T and
  inv: cdclW-M-level-inv S 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 (trail S) L = backtrack-lvl S
     $\implies$  conflicting S = Some (D + {#L#})
     $\implies$  get-level (trail S) L = get-maximum-level (trail S) (D + {#L#})
     $\implies$  get-maximum-level (trail S) D  $\equiv$  i
     $\implies$  undefined-lit M1 L
     $\implies$  T  $\sim$  cons-trail (Propagated L (D + {#L#}))
      (reduce-trail-to M1
        (add-learned-cls (D + {#L#})
          (update-backtrack-lvl i
            (update-conflicting None S))))
     $\implies$  P S T
shows P S T
proof –
obtain K i M1 M2 L D where
  decomp: (Marked K (Suc i) # M1, M2)  $\in$  set (get-all-marked-decomposition (trail S)) and
  L: get-level (trail S) L = backtrack-lvl S and
  confl: conflicting S = Some (D + {#L#}) and
  lev-L: get-level (trail S) L = get-maximum-level (trail S) (D + {#L#}) and
  lev-D: get-maximum-level (trail S) D  $\equiv$  i and
  T: T  $\sim$  cons-trail (Propagated L (D + {#L#}))
    (reduce-trail-to M1
      (add-learned-cls (D + {#L#})
        (update-backtrack-lvl i
          (update-conflicting None S))))
using bt by (elim backtrackE) metis

have atm-of L  $\notin$  ( $\lambda l.$  atm-of (lit-of l)) ‘ set M1
using backtrack-lit-skipped[of S L K i M1 M2] L decomp bt confl lev-L lev-D inv
unfolding cdclW-M-level-inv-def
by (fastforce simp add: lits-of-def)
then have undefined-lit M1 L
by (auto simp: defined-lit-map)
then show ?thesis
using backtrackH[OF decomp L confl lev-L lev-D - T] by simp
qed

```

lemmas *backtrack-induction-lev2* = *backtrack-induction-lev*[consumes 2, case-names *backtrack*]

lemma *cdcl_W-all-induct-lev-full*:

fixes *S* :: 'st

assumes

cdcl_W: *cdcl_W S S'* **and**

inv[simp]: *cdcl_W-M-level-inv S* **and**

propagateH: $\bigwedge C L T. C + \{\#L\# \} \in \# \text{ clauses } S \implies \text{trail } S \models_{as} C \text{Not } C$

$\implies \text{undefined-lit } (\text{trail } S) L \implies \text{conflicting } S = \text{None}$

$\implies T \sim \text{cons-trail } (\text{Propagated } L (C + \{\#L\# \})) S$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$ **and**

conflictH: $\bigwedge D T. D \in \# \text{ clauses } S \implies \text{conflicting } S = \text{None} \implies \text{trail } S \models_{as} C \text{Not } D$

$\implies T \sim \text{update-conflicting } (\text{Some } D) S$

$\implies P S T$ **and**

forgetH: $\bigwedge C T. \neg \text{trail } S \models_{asm} \text{clauses } S$

$\implies C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$

$\implies C \notin \# \text{ init-clss } S$

$\implies C \in \# \text{ learned-clss } S$

$\implies \text{conflicting } S = \text{None}$

$\implies T \sim \text{remove-cl } C S$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$ **and**

restartH: $\bigwedge T. \neg \text{trail } S \models_{asm} \text{clauses } S$

$\implies \text{conflicting } S = \text{None}$

$\implies T \sim \text{restart-state } S$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$ **and**

decideH: $\bigwedge L T. \text{conflicting } S = \text{None} \implies \text{undefined-lit } (\text{trail } S) L$

$\implies \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$

$\implies T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$ **and**

skipH: $\bigwedge L C' M D T. \text{trail } S = \text{Propagated } L C' \# M$

$\implies \text{conflicting } S = \text{Some } D \implies \neg L \notin \# D \implies D \neq \{\#\}$

$\implies T \sim \text{tl-trail } S$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$ **and**

resolveH: $\bigwedge L C M D T.$

$\text{trail } S = \text{Propagated } L ((C + \{\#L\# \})) \# M$

$\implies \text{conflicting } S = \text{Some } (D + \{\#-L\# \})$

$\implies \text{get-maximum-level } (\text{Propagated } L (C + \{\#L\# \}) \# M) D = \text{backtrack-lvl } S$

$\implies T \sim (\text{update-conflicting } (\text{Some } (D \# \cup C)) (\text{tl-trail } S))$

$\implies \text{cdcl}_W\text{-M-level-inv } S$

$\implies P S T$ **and**

backtrackH: $\bigwedge K i M1 M2 L D T.$

$(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$

$\implies \text{get-level } (\text{trail } S) L = \text{backtrack-lvl } S$

$\implies \text{conflicting } S = \text{Some } (D + \{\#L\# \})$

$\implies \text{get-maximum-level } (\text{trail } S) (D + \{\#L\# \}) = \text{get-level } (\text{trail } S) L$

$\implies \text{get-maximum-level } (\text{trail } S) D \equiv i$

$\implies \text{undefined-lit } M1 L$

$\implies T \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \}))$

$(\text{reduce-trail-to } M1$

$(\text{add-learned-cl } (D + \{\#L\# \}))$

```

      (update-backtrack-lvl i
       (update-conflicting None S))))
     $\Rightarrow$  cdclW-M-level-inv S
     $\Rightarrow$  P S T
  shows P S S'
  using cdclW
proof (induct S' rule: cdclW-all-rules-induct)
  case (propagate S')
  then show ?case by (elim propagateE) (frule propagateH; simp)
next
  case (conflict S')
  then show ?case by (elim conflictE) (frule conflictH; simp)
next
  case (restart S')
  then show ?case by (elim restartE) (frule restartH; simp)
next
  case (decide T)
  then show ?case by (elim decideE) (frule decideH; simp)
next
  case (backtrack S')
  then show ?case
    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 cdclW-all-induct-lev2 = cdclW-all-induct-lev-full[consumes 2, case-names propagate conflict
forget restart decide skip resolve backtrack]

lemmas cdclW-all-induct-lev = cdclW-all-induct-lev-full[consumes 1, case-names lev-inv propagate
conflict forget restart decide skip resolve backtrack]

thm cdclW-o-induct
lemma cdclW-o-induct-lev[consumes 1, case-names M-lev decide skip resolve backtrack]:
  fixes S :: 'st
  assumes
    cdclW: cdclW-o S T and
    inv[simp]: cdclW-M-level-inv S and
    decideH:  $\bigwedge L T. \text{conflicting } S = \text{None} \Rightarrow \text{undefined-lit } (\text{trail } S) L$ 
       $\Rightarrow \text{atm-of } L \in \text{atms-of-msu } (\text{init-clss } S)$ 
       $\Rightarrow T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S)$ 
       $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
       $\Rightarrow P S T$  and
    skipH:  $\bigwedge L C' M D T. \text{trail } S = \text{Propagated } L C' \# M$ 
       $\Rightarrow \text{conflicting } S = \text{Some } D \Rightarrow -L \notin \# D \Rightarrow D \neq \{\#\}$ 

```

```

     $\Rightarrow T \sim \text{tl-trail } S$ 
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T$  and
  resolveH:  $\bigwedge L \ C \ M \ D \ T.$ 
    trail S = Propagated L ( (C + {#L#}) ) # M
     $\Rightarrow \text{conflicting } S = \text{Some } (D + \{\#L\# \})$ 
     $\Rightarrow \text{get-maximum-level } (\text{Propagated } L \ (C + \{\#L\# \}) \ # \ M) \ D = \text{backtrack-lvl } S$ 
     $\Rightarrow T \sim \text{update-conflicting } (\text{Some } (D \ \# \cup \ C)) \ (\text{tl-trail } S)$ 
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T$  and
  backtrackH:  $\bigwedge K \ i \ M1 \ M2 \ L \ D \ T.$ 
    (Marked K (Suc i) # M1, M2)  $\in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ 
     $\Rightarrow \text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S$ 
     $\Rightarrow \text{conflicting } S = \text{Some } (D + \{\#L\# \})$ 
     $\Rightarrow \text{get-level } (\text{trail } S) \ L = \text{get-maximum-level } (\text{trail } S) \ (D + \{\#L\# \})$ 
     $\Rightarrow \text{get-maximum-level } (\text{trail } S) \ D \equiv i$ 
     $\Rightarrow \text{undefined-lit } M1 \ L$ 
     $\Rightarrow T \sim \text{cons-trail } (\text{Propagated } L \ (D + \{\#L\# \}))$ 
      (reduce-trail-to M1
        (add-learned-cls (D + {#L#})
          (update-backtrack-lvl i
            (update-conflicting None S))))
     $\Rightarrow \text{cdcl}_W\text{-M-level-inv } S$ 
     $\Rightarrow P \ S \ T$ 
  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:  $cdcl_W$ - $M$ -level-inv  $S$ 
  shows backtrack  $S'$   $T'$ 
  using assms apply (induction rule: backtrack-induction-lev)
  using inv apply simp
  apply (rule backtrack-rule)
  apply auto[5]
  by (auto simp: state-eq-def clauses-def  $cdcl_W$ - $M$ -level-inv-def simp del: state-simp)

lemma decide-state-eq-compatible:
  assumes
    decide  $S$   $T$  and
     $S \sim S'$  and
     $T \sim T'$ 
  shows decide  $S'$   $T'$ 
  using assms apply (elim decideE)
  apply (rule decide-rule)
  by (auto simp: state-eq-def clauses-def simp del: state-simp)

lemma skip-state-eq-compatible:
  assumes
    skip  $S$   $T$  and
     $S \sim S'$  and
     $T \sim T'$ 
  shows skip  $S'$   $T'$ 
  using assms apply (elim skipE)
  apply (rule skip-rule)
  by (auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of - # - trail -]
    simp del: state-simp dest: arg-cong[of - # trail - trail - tl])

lemma resolve-state-eq-compatible:
  assumes
    resolve  $S$   $T$  and
     $S \sim S'$  and
     $T \sim T'$ 
  shows resolve  $S'$   $T'$ 
  using assms apply (elim resolveE)
  apply (rule resolve-rule)
  by (auto simp: state-eq-def clauses-def HOL.eq-sym-conv[of - # - trail -]

```

simp del: state-simp dest: arg-cong[of - # trail - trail - tl])

lemma *forget-state-eq-compatible:*

assumes
forget S T and
S ~ 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-bj S T and cdcl_W-M-level-inv S
S ~ S' and
T ~ T'
shows *cdcl_W-bj S' T'*
using *assms*
by *induction (auto*
intro: skip-state-eq-compatible backtrack-state-eq-compatible resolve-state-eq-compatible)

lemma *trancpl-cdcl_W-bj-state-eq-compatible:*

assumes
cdcl_W-bj⁺⁺ S T and inv: cdcl_W-M-level-inv S and
S ~ S' and
T ~ T'
shows *cdcl_W-bj⁺⁺ S' 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 = this(3)[OF this(4-5)]*
have *cdcl_W⁺⁺ S T*
using *trancpl-mono[of cdcl_W-bj cdcl_W] other step.hyps(1) by blast*
then have *cdcl_W-M-level-inv T*
using *inv trancpl-cdcl_W-consistent-inv by blast*
then have *cdcl_W-bj⁺⁺ T T'*
using *⟨U ~ T⟩ cdcl_W-bj-state-eq-compatible[of T U] ⟨cdcl_W-bj T U⟩ 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
 $cdcl_W S S'$ and
 $inv: cdcl_W\text{-}M\text{-level-inv } S$ and
 $\forall L l. \text{Marked } L l \in \text{set } (trail S) \longrightarrow l > 0$
 shows $\forall L l. \text{Marked } L l \in \text{set } (trail S') \longrightarrow l > 0$
 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
 $cdcl_W\text{-}o S S'$ and
 $inv: cdcl_W\text{-}M\text{-level-inv } S$
 shows $init\text{-}clss S = init\text{-}clss S'$
 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
 $cdcl_W\text{-}o^{++} S S'$ and
 $inv: cdcl_W\text{-}M\text{-level-inv } S$
 shows $init\text{-}clss S = init\text{-}clss S'$
 using assms **apply** (induct rule: *trancpl.induct*)
 by (auto dest: *cdcl_W-o-no-more-init-clss*
 $dest!:: trancpl\text{-}cdcl_W\text{-consistent-inv}$ $dest: trancpl\text{-}mono\text{-explicit}[of cdcl_W\text{-}o - - cdcl_W]$
 $simp: other$)

lemma *rtrancpl-cdcl_W-o-no-more-init-clss*:

assumes
 $cdcl_W\text{-}o^{**} S S'$ and
 $inv: cdcl_W\text{-}M\text{-level-inv } S$
 shows $init\text{-}clss S = init\text{-}clss S'$
 using assms **unfolding** *rtrancpl-unfold* **by** (auto intro: *trancpl-cdcl_W-o-no-more-init-clss*)

lemma *cdcl_W-init-clss*:

$cdcl_W S T \Longrightarrow cdcl_W\text{-}M\text{-level-inv } S \Longrightarrow init\text{-}clss S = init\text{-}clss T$
 by (induct rule: *cdcl_W-all-induct-lev2*) (auto simp: *cdcl_W-M-level-inv-def*)

lemma *rtrancpl-cdcl_W-init-clss*:

$cdcl_W^{**} S T \Longrightarrow cdcl_W\text{-}M\text{-level-inv } S \Longrightarrow init\text{-}clss S = init\text{-}clss T$
 by (induct rule: *rtrancpl-induct*) (auto dest: *cdcl_W-init-clss* *rtrancpl-cdcl_W-consistent-inv*)

lemma *trancpl-cdcl_W-init-clss*:

$cdcl_W^{++} S T \Longrightarrow cdcl_W\text{-}M\text{-level-inv } S \Longrightarrow init\text{-}clss S = init\text{-}clss T$
 using *rtrancpl-cdcl_W-init-clss*[of S T] **unfolding** *rtrancpl-unfold* **by** auto

17.4.5 Learned Clause

This invariant shows that:

- the learned clauses are entailed by the initial set of clauses.

- the conflicting clause is entailed by the initial set of clauses.
- the marks are entailed by the clauses. A more precise version would be to show that either these marked are learned or are in the set of clauses

definition *cdcl_W-learned-clause* ($S :: 'st$) \longleftrightarrow
 $(\text{init-clss } S \models_{\text{psm}} \text{learned-clss } S$
 $\wedge (\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{init-clss } S \models_{\text{pm}} T)$
 $\wedge \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \subseteq \text{set-mset } (\text{clauses } S))$

lemma *cdcl_W-learned-clause-S0-cdcl_W[simp]*:
 $\text{cdcl}_W\text{-learned-clause } (\text{init-state } N)$
unfolding *cdcl_W-learned-clause-def* **by** *auto*

lemma *cdcl_W-learned-clss*:

assumes

$\text{cdcl}_W \ S \ S'$ **and**

$\text{learned}: \text{cdcl}_W\text{-learned-clause } S$ **and**

$\text{lev-inv}: \text{cdcl}_W\text{-M-level-inv } S$

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** $\text{decomp} = \text{this}(1)$ **and** $\text{confl} = \text{this}(3)$ **and** $\text{undef} = \text{this}(6)$
and $T = \text{this}(7)$

show *?case*

using *decomp confl learned undef T lev-inv unfolding cdcl_W-learned-clause-def*

by (*auto dest!: get-all-marked-decomposition-exists-prepend*

simp: clauses-def cdcl_W-M-level-inv-decomp dest: true-clss-clss-left-right)

next

case (*resolve* $L \ C \ M \ D$) **note** $\text{trail} = \text{this}(1)$ **and** $\text{confl} = \text{this}(2)$ **and** $\text{lvl} = \text{this}(3)$ **and**
 $T = \text{this}(4)$

moreover

have $\text{init-clss } S \models_{\text{psm}} \text{learned-clss } S$

using *learned trail unfolding cdcl_W-learned-clause-def clauses-def* **by** *auto*

then have $\text{init-clss } S \models_{\text{pm}} C + \{\#L\#\}$

using *trail learned unfolding cdcl_W-learned-clause-def clauses-def*

by (*auto dest: true-clss-clss-in-imp-true-clss-clss*)

ultimately show *?case*

using *learned*

by (*auto dest: mk-disjoint-insert true-clss-clss-left-right*

simp add: cdcl_W-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 cdcl_W-learned-clause-def

simp del: state-simp

dest: true-clss-clssm-subsetE)

next

case *propagate*

then show *?case* **using** *learned* **by** (*auto simp: cdcl_W-learned-clause-def 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 *rtrancp-cdcl_W-learned-clss:*

```

assumes
  cdclW** S S' and
  cdclW-M-level-inv S
  cdclW-learned-clause S
shows cdclW-learned-clause S'
using assms by induction (auto dest: cdclW-learned-clss intro: rtrancp-cdclW-consistent-inv)

```

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 (*
 $(\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S'))$
 $\wedge (\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S') \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S'))$
 $\wedge \text{atms-of-msu } (\text{learned-clss } S') \subseteq \text{atms-of-msu } (\text{init-clss } S')$
 $\wedge \text{atm-of ' (lits-of } (\text{trail } S')) \subseteq \text{atms-of-msu } (\text{init-clss } S'))$
)

lemma *no-strange-atm-decomp:*

```

assumes no-strange-atm S
shows conflicting S = Some T  $\implies$  atms-of T  $\subseteq$  atms-of-msu (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 atms-of-msu (learned-clss S)  $\subseteq$  atms-of-msu (init-clss S)
and atm-of ' (lits-of (trail S))  $\subseteq$  atms-of-msu (init-clss S)
using assms unfolding no-strange-atm-def by blast+

```

lemma *no-strange-atm-S0 [simp]: no-strange-atm (init-state N)*
unfolding *no-strange-atm-def* **by** *auto*

lemma *cdcl_W-no-strange-atm-explicit:*

```

assumes
  cdclW S S' and
  lev: cdclW-M-level-inv S and
  conf:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S)$  and
  marked:  $\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{atms-of mark} \subseteq \text{atms-of-msu } (\text{init-clss } S)$  and
  learned:  $\text{atms-of-msu } (\text{learned-clss } S) \subseteq \text{atms-of-msu } (\text{init-clss } S)$  and
  trail:  $\text{atm-of ' (lits-of } (\text{trail } S)) \subseteq \text{atms-of-msu } (\text{init-clss } S)$ 
shows  $(\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-msu } (\text{init-clss } S')) \wedge$   

 $(\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S') \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-msu } (\text{init-clss } S')) \wedge$   

 $\text{atms-of-msu } (\text{learned-clss } S') \subseteq \text{atms-of-msu } (\text{init-clss } S') \wedge$   

 $\text{atm-of ' (lits-of } (\text{trail } S')) \subseteq \text{atms-of-msu } (\text{init-clss } S') \text{ (is ?C } S' \wedge ?M S' \wedge ?U S' \wedge ?V S')$ 
using assms(1,2)

```

proof (*induct rule: cdcl_W-all-induct-lev2*)

```

case (propagate C L T) note C-L = this(1) and undef = this(3) and confl = this(4) and T = this(5)
have ?C (cons-trail (Propagated L (C + {#L#}))) S using confl undef by auto
moreover
  have atms-of (C + {#L#})  $\subseteq$  atms-of-msu (init-clss S)
  by (metis (no-types) atms-of-atms-of-ms-mono atms-of-ms-union clauses-def mem-set-mset-iff
    C-L learned set-mset-union sup.orderE)
  then have ?M (cons-trail (Propagated L (C + {#L#}))) S using undef
  by (simp add: marked)
moreover have ?U (cons-trail (Propagated L (C + {#L#}))) S
  using learned undef by auto
moreover have ?V (cons-trail (Propagated L (C + {#L#}))) S
  using C-L learned trail undef unfolding clauses-def
  by (auto simp: in-plus-implies-atm-of-on-atms-of-ms)
ultimately show ?case using T by auto
next
  case (decide L)
  then show ?case using learned marked conf trail unfolding clauses-def by auto
next
  case (skip L C M D)
  then show ?case using learned marked conf trail by auto
next
  case (conflict D T) note T = this(4)
  have D: atm-of ' set-mset D  $\subseteq \bigcup$  (atms-of ' (set-mset (clauses S)))
  using 'D  $\in \#$  clauses S' by (auto simp add: atms-of-def atms-of-ms-def)
  moreover {
    fix xa :: 'v literal
    assume a1: atm-of ' set-mset D  $\subseteq (\bigcup x \in \text{set-mset (init-clss S). atms-of } x)$ 
       $\cup (\bigcup x \in \text{set-mset (learned-clss S). atms-of } x)$ 
    assume a2:  $(\bigcup x \in \text{set-mset (learned-clss S). atms-of } x) \subseteq (\bigcup x \in \text{set-mset (init-clss S). atms-of } x)$ 
    assume xa  $\in \#$  D
    then have atm-of xa  $\in \text{UNION (set-mset (init-clss S)) atms-of}$ 
    using a2 a1 by (metis (no-types) Un-iff atm-of-lit-in-atms-of atms-of-def subset-Un-eq)
    then have  $\exists m \in \text{set-mset (init-clss S). atm-of } xa \in \text{atms-of } m$ 
    by blast
  } note H = this
  ultimately show ?case using conflict.premis T learned marked conf trail
  unfolding atms-of-def atms-of-ms-def clauses-def
  by (auto simp add: H )
next
  case (restart T)
  then show ?case using learned marked conf trail by auto
next
  case (forget C T) note C = this(3) and C-le = this(4) and confl = this(5) and
    T = this(6)
  have H:  $\bigwedge L \text{ mark. Propagated } L \text{ mark} \in \text{set (trail S)} \implies \text{atms-of mark} \subseteq \text{atms-of-msu (init-clss S)}$ 
  using marked by simp
  show ?case unfolding clauses-def apply standard
  using conf T trail C unfolding clauses-def apply (auto dest!: H)[]
  apply standard
  using T trail C apply (auto dest!: H)[]
  apply standard
  using T learned C C-le atms-of-ms-remove-subset[of set-mset (learned-clss S)] apply (auto)[]
  using T trail C apply (auto simp: clauses-def lits-of-def)[]
done
next

```

```

case (backtrack  $K$   $i$   $M1$   $M2$   $L$   $D$   $T$ ) note  $decomp = this(1)$  and  $confl = this(3)$  and  $undef = this(6)$ 
  and  $T = this(7)$ 
have  $?C$   $T$ 
  using  $conf$   $T$   $decomp$   $undef$   $lev$  by (auto simp:  $cdcl_W$ - $M$ -level-inv-decomp)
moreover have  $set\ M1 \subseteq set\ (trail\ S)$ 
  using  $backtrack.hyps(1)$  by auto
then have  $M$ :  $?M\ T$ 
  using  $marked\ conf\ undef\ confl\ T\ decomp\ lev$ 
  by (auto simp: image-subset-iff clauses-def  $cdcl_W$ - $M$ -level-inv-decomp)
moreover have  $?U\ T$ 
  using  $learned\ decomp\ conf\ confl\ T\ undef\ lev$  unfolding clauses-def
  by (auto simp:  $cdcl_W$ - $M$ -level-inv-decomp)
moreover have  $?V\ T$ 
  using  $M\ conf\ confl\ trail\ T\ undef\ decomp\ lev$  by (force simp:  $cdcl_W$ - $M$ -level-inv-decomp)
ultimately show  $?case$  by blast
next
case (resolve  $L$   $C$   $M$   $D$   $T$ ) note  $trail-S = this(1)$  and  $confl = this(2)$  and  $T = this(4)$ 
let  $?T = update-conflicting\ (Some\ (remdups-mset\ (D + C)))\ (tl-trail\ S)$ 
have  $?C\ ?T$ 
  using  $confl\ trail-S\ conf\ marked$  by simp
moreover have  $?M\ ?T$ 
  using  $confl\ trail-S\ conf\ marked$  by auto
moreover have  $?U\ ?T$ 
  using  $trail\ learned$  by auto
moreover have  $?V\ ?T$ 
  using  $confl\ trail-S\ trail$  by auto
ultimately show  $?case$  using  $T$  by auto
qed

```

lemma $cdcl_W$ -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 $rtranclp$ - $cdcl_W$ -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 $rtranclp$ - $cdcl_W$ -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 ($S::'st$)
 $\longleftrightarrow ((\forall T. \text{conflicting}\ S = \text{Some}\ T \longrightarrow \text{distinct-mset}\ T)$
 $\wedge \text{distinct-mset-mset}\ (\text{learned-clss}\ S)$
 $\wedge \text{distinct-mset-mset}\ (\text{init-clss}\ S)$
 $\wedge (\forall L\ mark. (\text{Propagated}\ L\ mark \in \text{set}\ (trail\ S) \longrightarrow \text{distinct-mset}\ (mark))))$

lemma $distinct-cdcl_W$ -state-decomp:
assumes $distinct-cdcl_W$ -state ($S::'st$)
shows $\forall T. \text{conflicting}\ S = \text{Some}\ T \longrightarrow \text{distinct-mset}\ T$
and $\text{distinct-mset-mset}\ (\text{learned-clss}\ S)$
and $\text{distinct-mset-mset}\ (\text{init-clss}\ S)$
and $\forall L\ mark. (\text{Propagated}\ L\ mark \in \text{set}\ (trail\ S) \longrightarrow \text{distinct-mset}\ (mark))$

```

using assms unfolding distinct-cdclW-state-def by blast+

lemma distinct-cdclW-state-decomp-2:
  assumes distinct-cdclW-state (S::'st)
  shows conflicting S = Some T  $\implies$  distinct-mset T
  using assms unfolding distinct-cdclW-state-def by auto

lemma distinct-cdclW-state-S0-cdclW[simp]:
  distinct-mset-mset N  $\implies$  distinct-cdclW-state (init-state N)
  unfolding distinct-cdclW-state-def by auto

lemma distinct-cdclW-state-inv:
  assumes
    cdclW S S' and
    cdclW-M-level-inv S and
    distinct-cdclW-state S
  shows distinct-cdclW-state S'
  using assms
proof (induct rule: cdclW-all-induct-lev2)
  case (backtrack K i M1 M2 L D)
  then show ?case
    unfolding distinct-cdclW-state-def
    by (fastforce dest: get-all-marked-decomposition-incl simp: cdclW-M-level-inv-decomp)
next
  case restart
  then show ?case unfolding distinct-cdclW-state-def distinct-mset-set-def clauses-def
  using learned-clss-restart-state[of S] by auto
next
  case resolve
  then show ?case
    by (auto simp add: distinct-cdclW-state-def distinct-mset-set-def clauses-def
      distinct-mset-single-add
      intro!: distinct-mset-union-mset)
qed (auto simp add: distinct-cdclW-state-def distinct-mset-set-def clauses-def)

lemma rtanclp-distinct-cdclW-state-inv:
  assumes
    cdclW** S S' and
    cdclW-M-level-inv S and
    distinct-cdclW-state S
  shows distinct-cdclW-state S'
  using assms apply (induct rule: rtanclp-induct)
  using distinct-cdclW-state-inv rtanclp-cdclW-consistent-inv by blast+

```

17.4.8 Conflicts and co

This invariant shows that each mark contains a contradiction only related to the previously defined variable.

abbreviation *every-mark-is-a-conflict* :: *'st \Rightarrow bool* **where**
every-mark-is-a-conflict S \equiv
 $\forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$
 $\longrightarrow (b \models_{as} CNot \ (\text{mark} - \{ \#L \# \}) \wedge L \in \# \text{ mark})$

definition *cdcl_W-conflicting* *S \equiv*
 $(\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot \ T)$

\wedge every-mark-is-a-conflict S

lemma *backtrack-atms-of-D-in-M1*:

fixes $M1 :: ('v, nat, 'v \text{ clause}) \text{ marked-lits}$

assumes

inv : $cdcl_W$ -M-level-inv S **and**

$undef$: undefined-lit $M1$ L **and**

i : $get\text{-}maximum\text{-}level$ ($trail\ S$) $D = i$ **and**

$decomp$: ($Marked\ K$ ($Suc\ i$) $\#$ $M1$, $M2$)

\in set ($get\text{-}all\text{-}marked\text{-}decomposition$ ($trail\ S$)) **and**

$S\text{-}lvl$: $backtrack\text{-}lvl\ S = get\text{-}maximum\text{-}level$ ($trail\ S$) ($D + \{\#L\# \}$) **and**

$S\text{-}confl$: $conflicting\ S = Some\ (D + \{\#L\# \})$ **and**

$undef$: undefined-lit $M1$ L **and**

T : $T \sim (cons\text{-}trail\ (Propagated\ L\ (D + \{\#L\# \})))$

$(reduce\text{-}trail\text{-}to\ M1$

$(add\text{-}learned\text{-}cls\ (D + \{\#L\# \})$

$(update\text{-}backtrack\text{-}lvl\ i$

$(update\text{-}conflicting\ None\ S))))$ **and**

$confl$: $\forall T. conflicting\ S = Some\ T \longrightarrow trail\ S \models_{as} CNot\ T$

shows $atms\text{-}of\ D \subseteq atm\text{-}of\ ' \text{ lits-of } (tl\ (trail\ T))$

proof ($rule\ ccontr$)

let $?k = get\text{-}maximum\text{-}level$ ($trail\ S$) ($D + \{\#L\# \}$)

have $trail\ S \models_{as} CNot\ D$ **using** $confl\ S\text{-}confl$ **by** $auto$

then have $vars\text{-}of\text{-}D$: $atms\text{-}of\ D \subseteq atm\text{-}of\ ' \text{ lits-of } (trail\ S)$ **unfolding** $atms\text{-}of\text{-}def$

by ($meson\ image\text{-}subsetI\ mem\text{-}set\text{-}mset\text{-}iff\ true\text{-}annots\text{-}CNot\text{-}all\text{-}atms\text{-}defined$)

obtain $M0$ **where** M : $trail\ S = M0 @ M2 @ Marked\ K\ (Suc\ i) \# M1$

using $decomp$ **by** $auto$

have max : $get\text{-}maximum\text{-}level$ ($trail\ S$) ($D + \{\#L\# \}$)

$= length$ ($get\text{-}all\text{-}levels\text{-}of\text{-}marked$ ($M0 @ M2 @ Marked\ K\ (Suc\ i) \# M1$))

using inv **unfolding** $cdcl_W$ -M-level-inv-def $S\text{-}lvl\ M$ **by** $simp$

assume a : $\neg ?thesis$

then obtain L' **where**

L' : $L' \in atms\text{-}of\ D$ **and**

$L'\text{-notin}\text{-}M1$: $L' \notin atm\text{-}of\ ' \text{ lits-of } M1$

using $T\ undef\ decomp\ inv$ **by** ($auto\ simp$: $cdcl_W$ -M-level-inv-decomp)

then have $L'\text{-in}$: $L' \in atm\text{-}of\ ' \text{ lits-of } (M0 @ M2 @ Marked\ K\ (i + 1) \# [])$

using $vars\text{-}of\text{-}D$ **unfolding** M **by** $force$

then obtain L'' **where**

$L'' \in \# D$ **and**

L'' : $L' = atm\text{-}of\ L''$

using $L'\ L'\text{-notin}\text{-}M1$ **unfolding** $atms\text{-}of\text{-}def$ **by** $auto$

have $lev\text{-}L''$:

$get\text{-}level$ ($trail\ S$) $L'' = get\text{-}rev\text{-}level$ ($Marked\ K\ (Suc\ i) \# rev\ M2 @ rev\ M0$) ($Suc\ i$) L''

using $L'\text{-notin}\text{-}M1\ L''\ M$ **by** ($auto\ simp\ del$: $get\text{-}rev\text{-}level.simps$)

have $get\text{-}all\text{-}levels\text{-}of\text{-}marked$ ($trail\ S$) $= rev\ [1..<1+?k]$

using $inv\ S\text{-}lvl$ **unfolding** $cdcl_W$ -M-level-inv-def **by** $auto$

then have $get\text{-}all\text{-}levels\text{-}of\text{-}marked$ ($M0 @ M2$)

$= rev\ [Suc\ (Suc\ i)..<Suc\ (get\text{-}maximum\text{-}level\ (trail\ S)\ (D + \{\#L\# \}))]$

unfolding M **by** ($auto\ simp$: $rev\text{-}swap[symmetric]\ dest$!: $append\text{-}cons\text{-}eq\text{-}upt\text{-}length\text{-}i\text{-}end$)

then have M : $get\text{-}all\text{-}levels\text{-}of\text{-}marked\ M0 @ get\text{-}all\text{-}levels\text{-}of\text{-}marked\ M2$

$= rev\ [Suc\ (Suc\ i)..<Suc\ (length\ (get\text{-}all\text{-}levels\text{-}of\text{-}marked\ (M0 @ M2 @ Marked\ K\ (Suc\ i) \# M1)))]$

unfolding max **unfolding** M **by** $simp$

```

have get-rev-level (Marked K (Suc i) # rev (M0 @ M2)) (Suc i) L''
  ≥ Min (set ((Suc i) # get-all-levels-of-marked (Marked K (Suc i) # rev (M0 @ M2))))
  using get-rev-level-ge-min-get-all-levels-of-marked[of L''
    rev (M0 @ M2 @ [Marked K (Suc i)]) Suc i] L'-in
  unfolding L'' by (fastforce simp add: lits-of-def)
also have Min (set ((Suc i) # get-all-levels-of-marked (Marked K (Suc i) # rev (M0 @ M2))))
  = Min (set ((Suc i) # get-all-levels-of-marked (rev (M0 @ M2)))) by auto
also have ... = Min (set ((Suc i) # get-all-levels-of-marked M0 @ get-all-levels-of-marked M2))
  by (simp add: Un-commute)
also have ... = Min (set ((Suc i) # [Suc (Suc i)..<2 + length (get-all-levels-of-marked M0)
  + (length (get-all-levels-of-marked M2) + length (get-all-levels-of-marked M1))]))
  unfolding M by (auto simp add: Un-commute)
also have ... = Suc i by (auto intro: Min-eqI)
finally have get-rev-level (Marked K (Suc i) # rev (M0 @ M2)) (Suc i) L'' ≥ Suc i .
then have get-level (trail S) L'' ≥ i + 1
  using lev-L'' by simp
then have get-maximum-level (trail S) D ≥ i + 1
  using get-maximum-level-ge-get-level[OF ‹L'' ∈ # D›, of trail S] by auto
then show False using i by auto
qed

```

lemma *distinct-atms-of-incl-not-in-other:*

```

assumes
  a1: no-dup (M @ M') and a2:
  atms-of D ⊆ atm-of ' lits-of M'
shows ∀ x ∈ atms-of D. x ∉ atm-of ' lits-of M
proof -
  { fix aa :: 'a
    have ff1: ∧ l ms. undefined-lit ms l ∨ atm-of l
      ∈ set (map (λm. atm-of (lit-of (m::('a, 'b, 'c) marked-lit))) ms)
      by (simp add: defined-lit-map)
    have ff2: ∧ a. a ∉ atms-of D ∨ a ∈ atm-of ' lits-of M'
      using a2 by (meson subsetCE)
    have ff3: ∧ a. a ∉ set (map (λm. atm-of (lit-of m)) M')
      ∨ a ∉ set (map (λm. atm-of (lit-of m)) M)
      using a1 by (metis (lifting) IntI distinct-append empty-iff map-append)
    have ∀ L a f. ∃ l. ((a::'a) ∉ f ' L ∨ (l::'a literal) ∈ L) ∧ (a ∉ f ' L ∨ f l = a)
      by blast
    then have aa ∉ atms-of D ∨ aa ∉ atm-of ' lits-of M
      using ff3 ff2 ff1 by (metis (no-types) Marked-Propagated-in-iff-in-lits-of) }
    then show ?thesis
      by blast
  }
qed

```

lemma *cdcl_W-propagate-is-conclusion:*

```

assumes
  cdclW S S' and
  inv: cdclW-M-level-inv S and
  decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
  learned: cdclW-learned-clause S and
  conf: ∀ T. conflicting S = Some T ⟶ trail S ⊨as CNot T and
  alien: no-strange-atm S
shows all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))
using asms(1,2)

```

```

proof (induct rule: cdclW-all-induct-lev2)
  case restart
  then show ?case by auto
next
  case forget
  then show ?case using decomp by auto
next
  case conflict
  then show ?case using decomp by auto
next
  case (resolve L C M D) note tr = this(1) and T = this(4)
  let ?decomp = get-all-marked-decomposition M
  have M: set ?decomp = insert (hd ?decomp) (set (tl ?decomp))
    by (cases ?decomp) auto
  show ?case
    using decomp tr T unfolding all-decomposition-implies-def
    by (cases hd (get-all-marked-decomposition M))
      (auto simp: M)
next
  case (skip L C' M D) note tr = this(1) and T = this(5)
  have M: set (get-all-marked-decomposition M)
    = insert (hd (get-all-marked-decomposition M)) (set (tl (get-all-marked-decomposition M)))
    by (cases get-all-marked-decomposition M) auto
  show ?case
    using decomp tr T unfolding all-decomposition-implies-def
    by (cases hd (get-all-marked-decomposition M))
      (auto simp add: M)
next
  case decide note S = this(1) and undef = this(2) and T = this(4)
  show ?case using decomp T undef unfolding S all-decomposition-implies-def by auto
next
  case (propagate C L T) note propa = this(2) and undef = this(3) and T = this(5)
  obtain a y where ay: hd (get-all-marked-decomposition (trail S)) = (a, y)
    by (cases hd (get-all-marked-decomposition (trail S)))
  then have M: trail S = y @ a using get-all-marked-decomposition-decomp by blast
  have M': set (get-all-marked-decomposition (trail S))
    = insert (a, y) (set (tl (get-all-marked-decomposition (trail S))))
    using ay by (cases get-all-marked-decomposition (trail S)) auto
  have unmark a ∪ set-mset (init-clss S) ⊨ps unmark y
    using decomp ay unfolding all-decomposition-implies-def
    by (cases get-all-marked-decomposition (trail S)) fastforce+
  then have a-Un-N-M: unmark a ∪ set-mset (init-clss S)
    ⊨ps unmark (trail S)
    unfolding M by (auto simp add: all-in-true-clss-clss image-Un)

  have unmark 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  $\models_{ps}$  CNot C
    using a-Un-N-M true-clss-clss-left-right true-clss-clss-union-l-r by blast
  qed
  moreover have  $\bigwedge aa\ b.$ 
     $\forall (Ls, seen) \in set\ (get\_all\_marked\_decomposition\ (y\ @\ a)).$ 
     $unmark\ Ls \cup set\_mset\ (init\_clss\ S) \models_{ps} unmark\ seen$ 
 $\implies (aa, b) \in set\ (tl\ (get\_all\_marked\_decomposition\ (y\ @\ a)))$ 
 $\implies unmark\ aa \cup set\_mset\ (init\_clss\ S) \models_{ps} unmark\ b$ 
  by (metis (no-types, lifting) case-prod-conv get-all-marked-decomposition-never-empty-sym
    list.collapse list.set-intros(2))

  ultimately show ?case
  using decomp T undef unfolding ay all-decomposition-implies-def
  using M  $\langle unmark\ a \cup set\_mset\ (init\_clss\ S) \models_{ps} unmark\ y \rangle$ 
  ay by auto
next
case (backtrack K i M1 M2 L D T) note decomp' = this(1) and lev-L = this(2) and conf = this(3)
and
  undef = this(6) and T = this(7)
have  $\forall l \in set\ M2. \neg is\_marked\ l$ 
  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  $\in set\ (get\_all\_marked\_decomposition\ (trail\ T))$ 
  then have x: x  $\in set\ (get\_all\_marked\_decomposition\ (Propagated\ L\ ((D + \{\#L\#\})) \# M1))$ 
    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  $\vee$  x  $\in set\ ?tl$ 
    using x by (cases ?m) auto
  moreover {
    assume x  $\in set\ ?tl$ 
    then have x  $\in set\ (get\_all\_marked\_decomposition\ (trail\ S))$ 
      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)  $\Rightarrow unmark\ Ls$ 
       $\cup set\_mset\ (init\_clss\ (T))$ 
       $\models_{ps} unmark\ seen$ 
      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  $\langle x = ?hd \rangle$  by auto
    have (M1', M1'')  $\in set\ (get\_all\_marked\_decomposition\ (trail\ S))$ 
      using M1[symmetric] hd-get-all-marked-decomposition-skip-some[OF M1[symmetric],
        of M0 @ M2 - i + 1] unfolding M by fastforce
    then have 1: unmark M1'  $\cup set\_mset\ (init\_clss\ S)$ 

```

```

     $\models_{ps}$  unmark  $M1''$ 
    using decomp unfolding all-decomposition-implies-def by auto
  moreover
    have  $trail\ S \models_{as} CNot\ D$  using conf confl by auto
    then have  $vars-of-D: atms-of\ D \subseteq atm-of\ 'lits-of\ (trail\ S)$ 
      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] backtrack inv conf confl
      by (auto simp: cdclW-M-level-inv-decomp)
    have no-dup ( $trail\ S$ ) using inv by (auto simp: cdclW-M-level-inv-decomp)
    then have vars-in-M1:
       $\forall x \in atms-of\ D. x \notin atm-of\ 'lits-of\ (M0\ @\ M2\ @\ Marked\ K\ (i + 1)\ \# \ [])$ 
      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
    have  $M1 = M1'' @ M1'$  by (simp add: M1 get-all-marked-decomposition-decomp)
    have  $TT: unmark\ M1' \cup set-mset\ (init-clss\ S) \models_{ps} CNot\ D$ 
      using true-annots-true-clss-cl [OF (M1 \models_{as} CNot D)] true-clss-clss-left-right [OF 1,
        of CNot D] unfolding ( $M1 = M1'' @ M1'$ ) by (auto simp add: inf-sup-aci(5,7))
    have  $init-clss\ S \models_{pm} D + \{\#L\# \}$ 
      using conf learned cdclW-learned-clause-def confl by blast
    then have  $T': unmark\ M1' \cup set-mset\ (init-clss\ S) \models_p D + \{\#L\# \}$  by auto
    have  $atms-of\ (D + \{\#L\# \}) \subseteq atms-of-msu\ (clauses\ S)$ 
      using alien conf unfolding no-strange-atm-def clauses-def by auto
    then have  $unmark\ M1' \cup set-mset\ (init-clss\ S) \models_p \{\#L\# \}$ 
      using true-clss-clss-plus-CNot [OF T' TT] by auto
  ultimately
    have case x of (Ls, seen)  $\Rightarrow$  unmark Ls
       $\cup set-mset\ (init-clss\ T)$ 
       $\models_{ps}$  unmark seen using  $T'\ T\ decomp'\ undef\ inv$  unfolding  $x'$ 
      by (simp add: cdclW-M-level-inv-decomp)
  }
  ultimately show case x of (Ls, seen)  $\Rightarrow$  unmark Ls  $\cup set-mset\ (init-clss\ T)$ 
     $\models_{ps}$  unmark seen using T by auto
qed
qed

```

lemma *cdcl_W-propagate-is-false:*

```

  assumes
    cdclW S S' and
    lev: cdclW-M-level-inv S and
    learned: cdclW-learned-clause S and
    decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) and
    confl:  $\forall T. conflicting\ S = Some\ T \longrightarrow trail\ S \models_{as} CNot\ T$  and
    alien: no-strange-atm S and
    mark-confl: every-mark-is-a-conflict S
  shows every-mark-is-a-conflict S'
  using assms(1,2)
proof (induct rule: cdclW-all-induct-lev2)
  case (propagate C L T) note undef = this(3) and  $T = this(5)$ 
  show ?case

```

```

proof (intro allI impI)
  fix L' mark a b
  assume a @ Propagated L' mark # b = trail T
  then have (a=[] ∧ L = L' ∧ mark = C + {#L#} ∧ b = trail S)
    ∨ tl a @ Propagated L' mark # b = trail S
    using T undef by (cases a) fastforce+
  moreover {
    assume tl a @ Propagated L' mark # b = trail S
    then have b ⊨as CNot ( mark - {#L'#} ) ∧ L' ∈# mark
    using mark-confl by auto
  }
  moreover {
    assume a=[] and L = L' and mark = C + {#L#} and b = trail S
    then have b ⊨as CNot ( mark - {#L'#} ) ∧ L ∈# mark
    using (trail S ⊨as CNot C) by auto
  }
  ultimately show b ⊨as CNot ( mark - {#L'#} ) ∧ L' ∈# mark by blast
qed
next
case (decide L) note undef[simp] = this(2) and T = this(4)
have ∧a La mark b. a @ Propagated La mark # b = Marked L (backtrack-lvl S+1) # trail S
  ⇒ tl a @ Propagated La mark # b = trail S by (case-tac a, auto)
then show ?case using mark-confl T unfolding decide.hyps(1) by fastforce
next
case (skip L C' M D T) note tr = this(1) and T = this(5)
show ?case
proof (intro allI impI)
  fix L' mark a b
  assume a @ Propagated L' mark # b = trail T
  then have a @ Propagated L' mark # b = M using tr T by simp
  then have (Propagated L C' # a) @ Propagated L' mark # b = Propagated L C' # M by auto
  moreover have ∀ La mark a b. a @ Propagated La mark # b = Propagated L C' # M
    → b ⊨as CNot ( mark - {#La#} ) ∧ La ∈# mark
    using mark-confl unfolding skip.hyps(1) by simp
  ultimately show b ⊨as CNot ( mark - {#L'#} ) ∧ L' ∈# 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 ⊨as CNot ( mark - {#L'#} ) ∧ L' ∈# 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 None 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  $\models_{\text{as}}$  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  $\models_{\text{as}}$  CNot D
      using vars-in-M1 true-annots-remove-if-notin-vars[of M0 @ M2 @ Marked K (i + 1) # [] M1
        CNot D] (trail S  $\models_{\text{as}}$  CNot D) unfolding M lits-of-def by simp
    then have b  $\models_{\text{as}}$  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  $\models_{\text{as}}$  CNot (mark - {#La#})  $\wedge$  La  $\in$  # mark
      using mark-confl by blast
  }
  ultimately show b  $\models_{\text{as}}$  CNot (mark - {#La#})  $\wedge$  La  $\in$  # mark by fast
qed
qed

```

lemma cdcl_W-conflicting-is-false:
 assumes
 cdcl_W S S' and
 M-lev: cdcl_W-M-level-inv S and

$\text{confl-inv: } \forall T. \text{ conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T$ **and**
 $\text{marked-conf: } \forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$
 $\longrightarrow (b \models_{\text{as}} \text{CNot } (\text{mark} - \{\#L\})) \wedge L \in \# \text{ mark}$ **and**
 $\text{dist: distinct-cdcl}_W\text{-state } S$
shows $\forall T. \text{ conflicting } S' = \text{Some } T \longrightarrow \text{trail } S' \models_{\text{as}} \text{CNot } T$
using $\text{assms}(1,2)$
proof ($\text{induct rule: cdcl}_W\text{-all-induct-lev2}$)
case ($\text{skip } L \ C' \ M \ D$) **note** $\text{tr-}S = \text{this}(1)$ **and** $T = \text{this}(5)$
then have $\text{Propagated } L \ C' \ \# \ M \models_{\text{as}} \text{CNot } D$ **using** assms skip **by** auto
moreover
have $L \notin \# \ D$
proof (rule ccontr)
assume $\neg \text{?thesis}$
then have $-L \in \text{lits-of } M$
using $\text{in-CNot-implies-uminus}(2)[\text{of } D \ L \ \text{Propagated } L \ C' \ \# \ M]$
 $\langle \text{Propagated } L \ C' \ \# \ M \models_{\text{as}} \text{CNot } D \rangle$ **by** simp
then show False
by ($\text{metis } M\text{-lev } \text{cdcl}_W\text{-M-level-inv-decomp}(1) \text{ consistent-interp-def insert-iff}$
 $\text{lits-of-cons marked-lit.sel}(2) \text{ skip.hyps}(1))$
qed
ultimately show ?case
using $\text{skip.hyps}(1-3) \text{ true-annots-CNot-lit-of-notin-skip } T$ **unfolding** $\text{cdcl}_W\text{-M-level-inv-def}$
by fastforce
next
case ($\text{resolve } L \ C \ M \ D \ T$) **note** $\text{tr} = \text{this}(1)$ **and** $\text{confl} = \text{this}(2)$ **and** $T = \text{this}(4)$
show ?case
proof (intro allI impI)
fix T'
have $\text{tl } (\text{trail } S) \models_{\text{as}} \text{CNot } C$ **using** $\text{tr assms}(4)$ **by** fastforce
moreover
have $\text{distinct-mset } (D + \{\#-L\# \})$ **using** confl dist
unfolding $\text{distinct-cdcl}_W\text{-state-def}$ **by** auto
then have $-L \notin \# \ D$ **unfolding** distinct-mset-def **by** auto
have $M \models_{\text{as}} \text{CNot } D$
proof $-$
have $\text{Propagated } L \ ((C + \{\#L\# \})) \ \# \ M \models_{\text{as}} \text{CNot } D \cup \text{CNot } \{\#-L\# \}$
using $\text{confl tr confl-inv}$ **by** force
then show ?thesis
using $M\text{-lev } \langle -L \notin \# \ D \rangle \text{ tr true-annots-lit-of-notin-skip}$
unfolding $\text{cdcl}_W\text{-M-level-inv-def}$ **by** force
qed
moreover assume $\text{conflicting } T = \text{Some } T'$
ultimately
show $\text{trail } T \models_{\text{as}} \text{CNot } T'$
using $\text{tr } T$ **by** auto
qed
qed ($\text{auto simp: assms}(2) \text{ cdcl}_W\text{-M-level-inv-decomp}$)

lemma $\text{cdcl}_W\text{-conflicting-decomp}$:
assumes $\text{cdcl}_W\text{-conflicting } S$
shows $\forall T. \text{ conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T$
and $\forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$
 $\longrightarrow (b \models_{\text{as}} \text{CNot } (\text{mark} - \{\#L\# \})) \wedge L \in \# \text{ mark}$
using assms **unfolding** $\text{cdcl}_W\text{-conflicting-def}$ **by** blast+

lemma *cdcl_W-conflicting-decomp2*:
assumes *cdcl_W-conflicting S* **and** *conflicting S = Some T*
shows *trail S ⊨_{as} CNot T*
using *assms* **unfolding** *cdcl_W-conflicting-def* **by** *blast+*

lemma *cdcl_W-conflicting-decomp2'*:
assumes
cdcl_W-conflicting S **and**
conflicting S = Some D
shows *trail S ⊨_{as} CNot D*
using *assms* **unfolding** *cdcl_W-conflicting-def* **by** *auto*

lemma *cdcl_W-conflicting-S0-cdcl_W[simp]*:
cdcl_W-conflicting (init-state N)
unfolding *cdcl_W-conflicting-def* **by** *auto*

17.4.9 Putting all the invariants together

lemma *cdcl_W-all-inv*:
assumes *cdcl_W: cdcl_W S S'* **and**
1: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) **and**
2: cdcl_W-learned-clause S **and**
4: cdcl_W-M-level-inv S **and**
5: no-strange-atm S **and**
7: distinct-cdcl_W-state S **and**
8: cdcl_W-conflicting S
shows *all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))*
and *cdcl_W-learned-clause S'*
and *cdcl_W-M-level-inv S'*
and *no-strange-atm S'*
and *distinct-cdcl_W-state S'*
and *cdcl_W-conflicting S'*
proof –
show *S1: all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))*
using *cdcl_W-propagate-is-conclusion[OF cdcl_W 4 1 2 - 5]* *8* **unfolding** *cdcl_W-conflicting-def*
by *blast*
show *S2: cdcl_W-learned-clause S' using cdcl_W-learned-clss[OF cdcl_W 2 4]* .
show *S4: cdcl_W-M-level-inv S' using cdcl_W-consistent-inv[OF cdcl_W 4]* .
show *S5: no-strange-atm S' using cdcl_W-no-strange-atm-inv[OF cdcl_W 5 4]* .
show *S7: distinct-cdcl_W-state S' using distinct-cdcl_W-state-inv[OF cdcl_W 4 7]* .
show *S8: cdcl_W-conflicting S'*
using *cdcl_W-conflicting-is-false[OF cdcl_W 4 - - 7]* *8* *cdcl_W-propagate-is-false[OF cdcl_W 4 2 1 - 5]*
unfolding *cdcl_W-conflicting-def* **by** *fast*
qed

lemma *rtrancp-cdcl_W-all-inv*:
assumes
cdcl_W: rtrancp cdcl_W S S' **and**
1: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) **and**
2: cdcl_W-learned-clause S **and**
4: cdcl_W-M-level-inv S **and**
5: no-strange-atm S **and**
7: distinct-cdcl_W-state S **and**
8: cdcl_W-conflicting S
shows

all-decomposition-implies-m (*init-clss* S') (*get-all-marked-decomposition* (*trail* S')) **and**
cdcl_W-learned-clause S' **and**
cdcl_W-M-level-inv S' **and**
no-strange-atm S' **and**
distinct-cdcl_W-state S' **and**
cdcl_W-conflicting S'
using *assms*
proof (*induct rule: rtrancpl-induct*)
case *base*
case 1 **then** **show** ?*case* **by** *blast*
case 2 **then** **show** ?*case* **by** *blast*
case 3 **then** **show** ?*case* **by** *blast*
case 4 **then** **show** ?*case* **by** *blast*
case 5 **then** **show** ?*case* **by** *blast*
case 6 **then** **show** ?*case* **by** *blast*
next
case (*step* S' S'') **note** $H = \text{this}$
case 1 **with** $H(3-7)[\text{OF this}(1-6)]$ **show** ?*case* **using** *cdcl_W-all-inv*[*OF* $H(2)$]
 H **by** *presburger*
case 2 **with** $H(3-7)[\text{OF this}(1-6)]$ **show** ?*case* **using** *cdcl_W-all-inv*[*OF* $H(2)$]
 H **by** *presburger*
case 3 **with** $H(3-7)[\text{OF this}(1-6)]$ **show** ?*case* **using** *cdcl_W-all-inv*[*OF* $H(2)$]
 H **by** *presburger*
case 4 **with** $H(3-7)[\text{OF this}(1-6)]$ **show** ?*case* **using** *cdcl_W-all-inv*[*OF* $H(2)$]
 H **by** *presburger*
case 5 **with** $H(3-7)[\text{OF this}(1-6)]$ **show** ?*case* **using** *cdcl_W-all-inv*[*OF* $H(2)$]
 H **by** *presburger*
case 6 **with** $H(3-7)[\text{OF this}(1-6)]$ **show** ?*case* **using** *cdcl_W-all-inv*[*OF* $H(2)$]
 H **by** *presburger*
qed

lemma *all-invariant-S0-cdcl_W*:
assumes *distinct-mset-mset* N
shows *all-decomposition-implies-m* (*init-clss* (*init-state* N))
(*get-all-marked-decomposition* (*trail* (*init-state* N))))
and *cdcl_W-learned-clause* (*init-state* N)
and $\forall T. \text{conflicting}(\text{init-state } N) = \text{Some } T \longrightarrow (\text{trail}(\text{init-state } N)) \models_{\text{as}} \text{CNot } T$
and *no-strange-atm* (*init-state* N)
and *consistent-interp* (*lits-of* (*trail* (*init-state* N))))
and $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = \text{trail}(\text{init-state } N) \longrightarrow$
($b \models_{\text{as}} \text{CNot}(\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark}$)
and *distinct-cdcl_W-state* (*init-state* N)
using *assms* **by** *auto*

lemma *cdcl_W-only-propagated-vars-unsat*:
assumes
marked: $\forall x \in \text{set } M. \neg \text{is-marked } x$ **and**
DN: $D \in \# \text{ clauses } S$ **and**
D: $M \models_{\text{as}} \text{CNot } D$ **and**
inv: *all-decomposition-implies-m* N (*get-all-marked-decomposition* M) **and**
state: *state* $S = (M, N, U, k, C)$ **and**
learned-cl: *cdcl_W-learned-clause* S **and**
atm-incl: *no-strange-atm* S
shows *unsatisfiable* (*set-mset* N)

```

proof (rule ccontr)
  assume  $\neg$  unsatisfiable (set-mset N)
  then obtain I where
    I: I  $\models_s$  set-mset N and
    cons: consistent-interp I and
    tot: total-over-m I (set-mset N)
    unfolding satisfiable-def by auto
  have atms-of-msu N  $\cup$  atms-of-msu U = atms-of-msu N
    using atm-incl state unfolding total-over-m-def no-strange-atm-def
    by (auto simp add: clauses-def)
  then have total-over-m I (set-mset N) using tot unfolding total-over-m-def by auto
  moreover have N  $\models_{psm}$  U using learned-cl state unfolding cdclW-learned-clause-def by auto
  ultimately have I-D: I  $\models$  D
    using I DN cons state unfolding true-clss-clss-def true-clss-def Ball-def
  by (metis Un-iff  $\langle$ atms-of-msu N  $\cup$  atms-of-msu U = atms-of-msu N $\rangle$  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 atms-of-ms (set-mset N  $\cup$  unmark M) = atms-of-msu N
    using atm-incl state unfolding no-strange-atm-def by auto
  then have total-over-m I (set-mset N  $\cup$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ' (set M))
    using tot unfolding total-over-m-def by auto
  then have I  $\models_s$  ( $\lambda a. \{\#lit\text{-of } a\# \}$ ) ' (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  $\models_s$  unmark M by auto
  {
    fix K
    assume K  $\in \#$  D
    then have  $-K \in \text{lits-of } M$ 
      using D unfolding true-annots-def Ball-def CNot-def true-annot-def true-clss-def true-lit-def
      Bex-mset-def by (metis (mono-tags, lifting) count-single less-not-refl mem-Collect-eq)
    then have  $-K \in I$  using IM true-clss-singleton-lit-of-implies-incl lits-of-def by fastforce
  }
  then have  $\neg$  I  $\models$  D using cons unfolding true-clss-def true-lit-def consistent-interp-def by auto
  then show False using I-D by blast
qed

```

We have actually a much stronger theorem, namely *all-decomposition-implies ?N* (*get-all-marked-decomposition ?M*) \implies *?N* \cup $\{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } ?M\} \models_{ps}$ unmark ?M, that show that the only choices we made are marked in the formula

```

lemma
  assumes all-decomposition-implies-m N (get-all-marked-decomposition M)
  and  $\forall m \in \text{set } M. \neg \text{is-marked } m$ 
  shows set-mset N  $\models_{ps}$  unmark M
proof -
  have T:  $\{\{\#lit\text{-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } M\} = \{\}$  using assms(2) by auto
  then show ?thesis
    using all-decomposition-implies-propagated-lits-are-implied[OF assms(1)] unfolding T by simp
qed

```

```

lemma conflict-with-false-implies-unsat:
  assumes
    cdclW: cdclW S S' and

```

```

    lev: cdclW-M-level-inv S and
    [simp]: conflicting S' = Some {#} 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 = Some {#}
  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
    cdclW S S' and
    lev: cdclW-M-level-inv S and
    conflicting: cdclW-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: cdclW-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: cdclW-M-level-inv-decomp)
  moreover
    have trail S  $\models_{as}$  CNot (D + {#L#})
    using conflicting confl unfolding cdclW-conflicting-def by auto
    then have lits-of (trail S)  $\models_s$  CNot (D + {#L#}) using true-annots-true-clss by blast
    ultimately have  $\neg \text{tautology } (D + \{ \#L\# \})$  using consistent-CNot-not-tautology by blast
    then show ?case using backtrack no-tauto
      by (auto simp: cdclW-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-cdclW-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-cdclW-state (S:: 'st)
 $\longleftrightarrow$  (trail S  $\models_{asm}$  init-clss S
 $\vee ((\forall L \in \text{atms-of-msu } (\text{init-clss } S). L \in \text{atm-of ' lits-of } (\text{trail } S)))$ 

```

$\wedge (\exists C \in \# \text{ init-clss } S. \text{ trail } S \models_{as} C \text{Not } C)))$

17.5 CDCL Strong Completeness

fun *mapi* :: ('a \Rightarrow nat \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a list \Rightarrow 'b list **where**
mapi - - [] = [] |
mapi *f* *n* (*x* # *xs*) = *f* *x* *n* # *mapi* *f* (*n* - 1) *xs*

lemma *mark-not-in-set-mapi*[simp]: $L \notin \text{set } M \implies \text{Marked } L \ k \notin \text{set } (\text{mapi } \text{Marked } i \ M)$
by (induct *M* arbitrary: *i*) auto

lemma *propagated-not-in-set-mapi*[simp]: $L \notin \text{set } M \implies \text{Propagated } L \ k \notin \text{set } (\text{mapi } \text{Marked } i \ M)$
by (induct *M* arbitrary: *i*) auto

lemma *image-set-mapi*:
 $f \text{ ' set } (\text{mapi } g \ i \ M) = \text{set } (\text{mapi } (\lambda x \ i. f \ (g \ x \ i)) \ i \ M)$
by (induction *M* arbitrary: *i*) auto

lemma *mapi-map-convert*:
 $\forall x \ i \ j. f \ x \ i = f \ x \ j \implies \text{mapi } f \ i \ M = \text{map } (\lambda x. f \ x \ 0) \ M$
by (induction *M* arbitrary: *i*) auto

lemma *defined-lit-mapi*: $\text{defined-lit } (\text{mapi } \text{Marked } i \ M) \ L \longleftrightarrow \text{atm-of } L \in \text{atm-of ' set } M$
by (induction *M*) (auto simp: defined-lit-map image-set-mapi mapi-map-convert)

lemma *cdcl_W-can-do-step*:

assumes

consistent-interp (*set* *M*) **and**

distinct *M* **and**

atm-of ' (set *M*) \subseteq *atms-of-msu* *N*

shows $\exists S. \text{rtrancp } \text{cdcl}_W \ (\text{init-state } N) \ S$

$\wedge \text{state } S = (\text{mapi } \text{Marked } (\text{length } M) \ M, N, \{\#\}, \text{length } M, \text{None})$

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*, *None*)

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* $\langle \text{distinct } (L \# M) \rangle$ *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 - ?S₀]*)
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*, *None*) **and**
rtranclp cdcl_W (*init-state N*) *S* **and**
final-cdcl_W-state S

proof –

obtain *S* **where**
st: rtranclp cdcl_W (*init-state N*) *S* **and**
S: state S = (*mapi Marked* (*length M*) *M*, *N*, {*#*}, *length M*, *None*)
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-cls* **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 conflict S S' \longleftrightarrow *conflict S S'*

proof –

have *tranclp conflict S S'* \implies *conflict S S'*
unfolding *full1-def* **by** (*induct rule: tranclp.induct*) *force+*
then have *tranclp conflict S S'* \implies *conflict S S'* **by** (*meson rtranclpD*)
then show *?thesis* **unfolding** *full1-def* **by** (*metis conflictE option.simps(3)*)
conflicting-update-conflicting state-eq-conflicting tranclp.intros(1)

qed

inductive *cdcl_W-cp* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**

conflict'[intro]: conflict S S' \implies cdcl_W-cp S S' |

propagate': propagate S S' \implies cdcl_W-cp S S'

lemma *rtranclp-cdcl_W-cp-rtranclp-cdcl_W*:

*cdcl_W-cp** S T \implies 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

cdcl_W-cp S T **and**

S \sim *S'* **and**

T \sim *T'*

```

shows  $cdcl_W\text{-}cp\ S'\ T'$ 
using assms
apply (induction)
  using conflict-state-eq-compatible apply auto[1]
using propagate' propagate-state-eq-compatible by auto

lemma tranclp-cdclW-cp-state-eq-compatible:
  assumes
     $cdcl_W\text{-}cp^{++}\ S\ T$  and
     $S \sim S'$  and
     $T \sim T'$ 
  shows  $cdcl_W\text{-}cp^{++}\ S'\ T'$ 
  using assms
proof induction
  case base
  then show ?case
    using cdclW-cp-state-eq-compatible by blast
next
  case (step  $U\ V$ )
  obtain  $ss :: 'st$  where
     $cdcl_W\text{-}cp\ S\ ss \wedge cdcl_W\text{-}cp^{**}\ ss\ U$ 
  by (metis (no-types) step(1) tranclpD)
  then show ?case
    by (meson cdclW-cp-state-eq-compatible rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp2
      state-eq-ref step(2) step(4) step(5))
qed

lemma option-full-cdclW-cp:
   $conflicting\ S \neq None \implies full\ cdcl_W\text{-}cp\ S\ S$ 
unfolding full-def rtranclp-unfold tranclp-unfold by (auto simp add: cdclW-cp.simps)

lemma skip-unique:
   $skip\ S\ T \implies skip\ S\ T' \implies T \sim T'$ 
  by (fastforce simp: state-eq-def simp del: state-simp)

lemma resolve-unique:
   $resolve\ S\ T \implies resolve\ S\ T' \implies T \sim T'$ 
  by (fastforce simp: state-eq-def simp del: state-simp)

lemma cdclW-cp-no-more-clauses:
  assumes  $cdcl_W\text{-}cp\ S\ S'$ 
  shows  $clauses\ S = clauses\ S'$ 
  using assms by (induct rule: cdclW-cp.induct) (auto elim!: conflictE propagateE)

lemma tranclp-cdclW-cp-no-more-clauses:
  assumes  $cdcl_W\text{-}cp^{++}\ S\ S'$ 
  shows  $clauses\ S = clauses\ S'$ 
  using assms by (induct rule: tranclp.induct) (auto dest: cdclW-cp-no-more-clauses)

lemma rtranclp-cdclW-cp-no-more-clauses:
  assumes  $cdcl_W\text{-}cp^{**}\ S\ S'$ 
  shows  $clauses\ S = clauses\ S'$ 
  using assms by (induct rule: rtranclp-induct) (fastforce dest: cdclW-cp-no-more-clauses) +

lemma no-conflict-after-conflict:

```

conflict S $T \implies \neg \text{conflict } T$ U
by *fastforce*

lemma *no-propagate-after-conflict*:
conflict S $T \implies \neg \text{propagate } T$ U
by *fastforce*

lemma *tranclp-cdcl_W-cp-propagate-with-conflict-or-not*:
assumes *cdcl_W-cp*⁺⁺ S U
shows (*propagate*⁺⁺ S $U \wedge \text{conflicting } U = \text{None}$)
 $\vee (\exists T D. \text{propagate}^{**} S T \wedge \text{conflict } T U \wedge \text{conflicting } U = \text{Some } D)$
proof –
have *propagate*⁺⁺ S $U \vee (\exists T. \text{propagate}^{**} S T \wedge \text{conflict } T U)$
using *assms* **by** *induction*
(force simp: cdcl_W-cp.simps tranclp-into-rtranclp dest: no-conflict-after-conflict
no-propagate-after-conflict)+
moreover
have *propagate*⁺⁺ S $U \implies \text{conflicting } U = \text{None}$
unfolding *tranclp-unfold-end* **by** *auto*
moreover
have $\bigwedge T. \text{conflict } T U \implies \exists D. \text{conflicting } U = \text{Some } D$
by *auto*
ultimately show *?thesis* **by** *meson*
qed

lemma *cdcl_W-cp-conflicting-not-empty[simp]*: *conflicting* $S = \text{Some } D \implies \neg \text{cdcl}_W\text{-cp } S$ S'
proof
assume *cdcl_W-cp* S S' **and** *conflicting* $S = \text{Some } D$
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 $S' \implies \text{cdcl}_W\text{-stgy } S$ $S' \mid$
other': *cdcl_W-o* S $S' \implies \text{no-step cdcl}_W\text{-cp } S \implies \text{full cdcl}_W\text{-cp } S' S'' \implies \text{cdcl}_W\text{-stgy } S$ S''

17.6.2 Invariants

These are the same invariants as before, but lifted

lemma *cdcl_W-cp-learned-clause-inv*:
assumes *cdcl_W-cp* S S'
shows *learned-clss* $S = \text{learned-clss } S'$
using *assms* **by** (*induct rule: cdcl_W-cp.induct*) *fastforce*+

lemma *rtranclp-cdcl_W-cp-learned-clause-inv*:
assumes *cdcl_W-cp*^{**} S S'
shows *learned-clss* $S = \text{learned-clss } S'$

using *assms* **by** (*induct* rule: *rtranclp-induct*) (*fastforce* dest: *cdcl_W-cp-learned-clause-inv*) +

lemma *tranclp-cdcl_W-cp-learned-clause-inv*:
assumes *cdcl_W-cp⁺⁺ S S'*
shows *learned-clss S = learned-clss S'*
using *assms* **by** (*simp* add: *rtranclp-cdcl_W-cp-learned-clause-inv tranclp-into-rtranclp*)

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 *rtranclp-cdcl_W-cp-backtrack-lvl*:
assumes *cdcl_W-cp^{**} S S'*
shows *backtrack-lvl S = backtrack-lvl S'*
using *assms* **by** (*induct* rule: *rtranclp-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'.hypos(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 *rtrancpl-cdcl_W-stgy-consistent-inv*:
assumes *cdcl_W-stgy** S S'*
and *cdcl_W-M-level-inv S*
shows *cdcl_W-M-level-inv S'*
using *assms* **by** *induction (auto dest!: cdcl_W-stgy-consistent-inv)*

lemma *cdcl_W-cp-no-more-init-clss*:
assumes *cdcl_W-cp S S'*
shows *init-clss S = init-clss S'*
using *assms* **by** (*induct rule: cdcl_W-cp.induct*) *auto*

lemma *trancpl-cdcl_W-cp-no-more-init-clss*:
assumes *cdcl_W-cp⁺⁺ S S'*
shows *init-clss S = init-clss S'*
using *assms* **by** (*induct rule: trancpl.induct*) (*auto dest: cdcl_W-cp-no-more-init-clss*)

lemma *cdcl_W-stgy-no-more-init-clss*:
assumes *cdcl_W-stgy S S' and cdcl_W-M-level-inv S*
shows *init-clss S = init-clss S'*
using *assms*
apply (*induct rule: cdcl_W-stgy.induct*)
unfolding *full1-def full-def* **apply** (*blast dest: trancpl-cdcl_W-cp-no-more-init-clss trancpl-cdcl_W-o-no-more-init-clss*)
by (*metis cdcl_W-o-no-more-init-clss rtrancpl-unfold trancpl-cdcl_W-cp-no-more-init-clss*)

lemma *rtrancpl-cdcl_W-stgy-no-more-init-clss*:
assumes *cdcl_W-stgy** S S' and cdcl_W-M-level-inv S*
shows *init-clss S = init-clss S'*
using *assms*
apply (*induct rule: rtrancpl-induct, simp*)
using *cdcl_W-stgy-no-more-init-clss* **by** (*simp add: rtrancpl-cdcl_W-stgy-consistent-inv*)

lemma *cdcl_W-cp-dropWhile-trail'*:
assumes *cdcl_W-cp S S'*
obtains *M where trail S' = M @ trail S and (∀ l ∈ set M. ¬is-marked l)*
using *assms* **by** *induction fastforce+*

lemma *rtrancpl-cdcl_W-cp-dropWhile-trail'*:
assumes *cdcl_W-cp** S S'*
obtains *M :: ('v, nat, 'v clause) marked-lit list where trail S' = M @ trail S and ∀ l ∈ set M. ¬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 *rtrancpl-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 $\text{length}(\text{trail } S) = \text{card}(\text{atm-of} \text{ ‘lits-of } (\text{trail } S))$

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 $(\lambda S. \text{card}(\text{atms-of-msu}(\text{init-clss } S)) - \text{length}(\text{trail } S))$

$+ (\text{if } \text{conflicting } S = \text{None} \text{ then } 1 \text{ else } 0)) S$

$> (\lambda S. \text{card}(\text{atms-of-msu}(\text{init-clss } S)) - \text{length}(\text{trail } S))$

$+ (\text{if } \text{conflicting } S = \text{None} \text{ then } 1 \text{ else } 0)) T$

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 *cdcl_W-no-strange-atm-inv alien M-lev* **apply** (*meson cdcl_W cdcl_W.simps cdcl_W-cp.cases*)

using *M-lev cdcl_W cdcl_W-cp-consistent-inv cdcl_W-M-level-inv-def* **apply** *blast*

using *cdcl_W* **by** (*auto simp: cdcl_W-cp.simps*)

with *assms*

show *?thesis* **by** *induction (auto split: split-if-asm)+*

qed

lemma *cdcl_W-cp-wf*: *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 - -*

$(\lambda S. \text{card}(\text{atms-of-msu}(\text{init-clss } S)) - \text{length}(\text{trail } S))$

$+ (\text{if } \text{conflicting } S = \text{None} \text{ then } 1 \text{ else } 0))$ ’)

apply *simp*

using *cdcl_W-cp-decreasing-measure* **unfolding** *less-than-iff* **by** *blast*

lemma *rtrancp-cdcl_W-all-struct-inv-cdcl_W-cp-iff-rtrancp-cdcl_W-cp*:

assumes

lev: *cdcl_W-M-level-inv* S **and**

alien: *no-strange-atm* S

shows $(\lambda a \ b. (\text{cdcl}_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a) \wedge \text{cdcl}_W\text{-cp } a \ b)^{**} S T$

$\longleftrightarrow \text{cdcl}_W\text{-cp}^{**} S T$

(*is ?I S T* \longleftrightarrow *?C S T*)

proof

assume

?I S T

```

then show ?C S T by induction auto
next
assume
  ?C S T
then show ?I S T
proof induction
  case base
  then show ?case by simp
next
case (step T U) note st = this(1) and cp = this(2) and IH = this(3)
have cdclW** S T
  by (metis rtrancpl-unfold cdclW-cp-conflicting-not-empty cp st
    rtrancpl-propagate-is-rtrancpl-cdclW trancpl-cdclW-cp-propagate-with-conflict-or-not)
then have
  cdclW-M-level-inv T and
  no-strange-atm T
  using ⟨cdclW** S T⟩ apply (simp add: assms(1) rtrancpl-cdclW-consistent-inv)
  using ⟨cdclW** S T⟩ alien rtrancpl-cdclW-no-strange-atm-inv lev by blast
then have (λa b. (cdclW-M-level-inv a ∧ no-strange-atm a)
  ∧ cdclW-cp a b)** T U
  using cp by auto
then show ?case using IH by auto
qed
qed

lemma cdclW-cp-normalized-element:
  assumes
    lev: cdclW-M-level-inv S and
    no-strange-atm S
  obtains T where full cdclW-cp S T
proof -
  let ?inv = λa. (cdclW-M-level-inv a ∧ no-strange-atm a)
  obtain T where T: full (λa b. ?inv a ∧ cdclW-cp a b) S T
  using cdclW-cp-wf wf-exists-normal-form[of λa b. ?inv a ∧ cdclW-cp a b]
  unfolding full-def by blast
  then have cdclW-cp** S T
    using rtrancpl-cdclW-all-struct-inv-cdclW-cp-iff-rtrancpl-cdclW-cp assms unfolding full-def
    by blast
  moreover
  then have cdclW** S T
    using rtrancpl-cdclW-cp-rtrancpl-cdclW by blast
  then have
    cdclW-M-level-inv T and
    no-strange-atm T
    using ⟨cdclW** S T⟩ apply (simp add: assms(1) rtrancpl-cdclW-consistent-inv)
    using ⟨cdclW** S T⟩ assms(2) rtrancpl-cdclW-no-strange-atm-inv lev by blast
  then have no-step cdclW-cp T
    using T unfolding full-def by auto
  ultimately show thesis using that unfolding full-def by blast
qed

lemma in-atms-of-implies-atm-of-on-atms-of-ms:
  C + {#L#} ∈# A ⟹ x ∈ atms-of C ⟹ x ∈ 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-cdclW-cp-step:
  assumes no-strange-atm S
  shows  $\exists S''.$  full cdclW-cp S S''
  using assms
proof (induct card (atms-of-msu (init-clss S) - atm-of 'lits-of (trail S)) arbitrary: S)
  case 0 note card = this(1) and alien = this(2)
  then have atm: atms-of-msu (init-clss S) = atm-of ' lits-of (trail S)
    unfolding no-strange-atm-def by auto
  { assume a:  $\exists S'. \text{conflict } S S'$ 
    then obtain S' where S': conflict S S' by metis
    then have  $\forall S''. \neg \text{cdcl}_W\text{-cp } S' S''$  by auto
    then have ?case using a S' cdclW-cp.conflict' unfolding full-def by blast
  }
  moreover {
    assume a:  $\exists S'. \text{propagate } S S'$ 
    then obtain S' where propagate S S' by blast
    then obtain M N U k C L where S: state S = (M, N, U, k, None)
    and S': state S' = (Propagated L ( (C + {#L#})) # M, N, U, k, None)
    and C + {#L#}  $\in\#$  clauses S
    and M  $\models_{as}$  CNot C
    and undefined-lit M L
    using propagate by auto
    have atms-of-msu U  $\subseteq$  atms-of-msu N using alien S unfolding no-strange-atm-def by auto
    then have atm-of L  $\in$  atms-of-msu (init-clss S)
      using  $\langle C + \{ \#L\# \} \in\# \text{ clauses } S \rangle S$  unfolding atms-of-ms-def clauses-def by force+
    then have False using  $\langle \text{undefined-lit } M L \rangle S$  unfolding atm unfolding lits-of-def
      by (auto simp add: defined-lit-map)
  }
  ultimately show ?case by (metis cdclW-cp.cases full-def rtranclp.rtrancl-refl)
next
  case (Suc n) note IH = this(1) and card = this(2) and alien = this(3)
  { assume a:  $\exists S'. \text{conflict } S S'$ 
    then obtain S' where S': conflict S S' by metis
    then have  $\forall S''. \neg \text{cdcl}_W\text{-cp } S' S''$  by auto
    then have ?case unfolding full-def Ex-def using S' cdclW-cp.conflict' by blast
  }
  moreover {
    assume a:  $\exists S'. \text{propagate } S S'$ 
    then obtain S' where propagate: propagate S S' by blast
    then obtain M N U k C L where
      S: state S = (M, N, U, k, None) and
      S': state S' = (Propagated L ( (C + {#L#})) # M, N, U, k, None) and
      C + {#L#}  $\in\#$  clauses S and
      M  $\models_{as}$  CNot C and
      undefined-lit M L
  }

```

```

    by fastforce
  then have atm-of L  $\notin$  atm-of ' lits-of M
    unfolding lits-of-def by (auto simp add: defined-lit-map)
  moreover
    have no-strange-atm S' using alien propagate propagate-no-strange-atm by blast
    then have atm-of L  $\in$  atms-of-msu N using S' unfolding no-strange-atm-def by auto
    then have  $\bigwedge A. \{atm-of L\} \subseteq atms-of-msu N - A \vee atm-of L \in A$  by force
  moreover have Suc n - card {atm-of L} = n by simp
  moreover have card (atms-of-msu N - atm-of ' lits-of M) = Suc n
    using card S S' by simp
  ultimately
    have card (atms-of-msu N - atm-of ' insert L (lits-of M)) = n
      by (metis (no-types) Diff-insert card-Diff-subset finite.emptyI finite.insertI image-insert)
    then have n = card (atms-of-msu (init-clss S') - atm-of ' lits-of (trail S'))
      using card S S' by simp
  then have a1: Ex (full cdclW-cp S') using IH ⟨no-strange-atm S'⟩ by blast
  have ?case
    proof -
      obtain S'' :: 'st where
        ff1: cdclW-cp** S' S''  $\wedge$  no-step cdclW-cp S''
        using a1 unfolding full-def by blast
      have cdclW-cp** S S''
        using ff1 cdclW-cp.intros(2)[OF propagate]
        by (metis (no-types) converse-rtranclp-into-rtranclp)
      then have  $\exists S''. cdcl_{W\text{-}cp}^{**} S S'' \wedge (\forall S'''. \neg cdcl_{W\text{-}cp} S'' S''')$ 
        using ff1 by blast
      then show ?thesis unfolding full-def
        by meson
    qed
  }
  ultimately show ?case unfolding full-def by (metis cdclW-cp.cases rtranclp.rtrancl-refl)
qed

```

17.6.3 Literal of highest level in conflicting clauses

One important property of the *local.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

$\lambda S. (\text{conflicting } S = \text{None} \longrightarrow (\forall D \in \# \text{ clauses } S. \neg \text{trail } S \models_{as} CNot D))$

abbreviation *conflict-is-false-with-level* :: 'st \Rightarrow bool **where**

conflict-is-false-with-level S $\equiv \forall D. \text{conflicting } S = \text{Some } D \longrightarrow D \neq \{\#\}$

$\longrightarrow (\exists L \in \# D. \text{get-level } (\text{trail } S) L = \text{backtrack-lvl } S)$

lemma *not-conflict-not-any-negated-init-clss*:

assumes $\forall S'. \neg \text{conflict } S S'$

shows *no-clause-is-false* S

using *assms state-eq-ref* **by** blast

lemma *full-cdcl_W-cp-not-any-negated-init-clss*:

assumes *full cdcl_W-cp* S S'

shows *no-clause-is-false* S'

using *assms not-conflict-not-any-negated-init-clss* **unfolding** *full-def* **by** blast

```

lemma full1-cdclW-cp-not-any-negated-init-clss:
  assumes full1 cdclW-cp S S'
  shows no-clause-is-false S'
  using assms not-conflict-not-any-negated-init-clss unfolding full1-def by blast

lemma cdclW-stgy-not-non-negated-init-clss:
  assumes cdclW-stgy S S'
  shows no-clause-is-false S'
  using assms apply (induct rule: cdclW-stgy.induct)
  using full1-cdclW-cp-not-any-negated-init-clss full-cdclW-cp-not-any-negated-init-clss by metis+

lemma rtrancp-cdclW-stgy-not-non-negated-init-clss:
  assumes cdclW-stgy** S S' and no-clause-is-false S
  shows no-clause-is-false S'
  using assms by (induct rule: rtrancp-induct) (auto simp: cdclW-stgy-not-non-negated-init-clss)

lemma cdclW-stgy-conflict-ex-lit-of-max-level:
  assumes cdclW-cp S S'
  and no-clause-is-false S
  and cdclW-M-level-inv S
  shows conflict-is-false-with-level S'
  using assms
proof (induct rule: cdclW-cp.induct)
  case conflict'
  then show ?case by auto
next
  case propagate'
  then show ?case by auto
qed

lemma no-chained-conflict:
  assumes conflict S S'
  and conflict S' S''
  shows False
  using assms by fastforce

lemma rtrancp-cdclW-cp-propa-or-propa-confl:
  assumes cdclW-cp** S U
  shows propagate** S U  $\vee$  ( $\exists T. \text{propagate** } S T \wedge \text{conflict } T U$ )
  using assms
proof induction
  case base
  then show ?case by auto
next
  case (step U V) note SU = this(1) and UV = this(2) and IH = this(3)
  consider (confl) T where propagate** S T and conflict T U
  | (propa) propagate** S U using IH by auto
  then show ?case
  proof cases
    case confl
    then have False using UV by auto
    then show ?thesis by fast
  next
    case propa

```

```

    also have conflict  $U \vee V$  propagate  $U \vee V$  using  $UV$  by (auto simp add: cdclW-cp.simps)
    ultimately show ?thesis by force
qed
qed

lemma rtranclp-cdclW-co-conflict-ex-lit-of-max-level:
  assumes full: full cdclW-cp  $S \ U$ 
  and cls-f: no-clause-is-false  $S$ 
  and conflict-is-false-with-level  $S$ 
  and lev: cdclW-M-level-inv  $S$ 
  shows conflict-is-false-with-level  $U$ 
proof (intro allI impI)
  fix  $D$ 
  assume confl: conflicting  $U = \text{Some } D$  and
     $D: D \neq \{\#\}$ 
  consider ( $CT$ ) conflicting  $S = \text{None}$  | ( $SD$ )  $D'$  where conflicting  $S = \text{Some } D'$ 
  by (cases conflicting  $S$ ) auto
  then show  $\exists L \in \#D. \text{get-level}(\text{trail } U) \ L = \text{backtrack-lvl } U$ 
  proof cases
    case  $SD$ 
    then have  $S = U$ 
      by (metis (no-types) assms(1) cdclW-cp-conflicting-not-empty full-def rtranclpD tranclpD)
    then show ?thesis using assms(3) confl D by blast
  next
    case  $CT$ 
    have init-clss  $U = \text{init-clss } S$  and learned-clss  $U = \text{learned-clss } S$ 
      using assms(1) unfolding full-def
      apply (metis (no-types) rtranclpD tranclp-cdclW-cp-no-more-init-clss)
      by (metis (mono-tags, lifting) assms(1) full-def rtranclp-cdclW-cp-learned-clause-inv)
    obtain  $T$  where propagate**  $S \ T$  and  $TU: \text{conflict } T \ U$ 
    proof –
      have  $f5: U \neq S$ 
        using confl CT by force
      then have cdclW-cp++  $S \ U$ 
        by (metis full full-def rtranclpD)
      have  $\bigwedge p \text{ pa. } \neg \text{propagate } p \text{ pa} \vee \text{conflicting } \text{pa} =$ 
        (None::'v literal multiset option)
        by auto
      then show ?thesis
        using  $f5$  that tranclp-cdclW-cp-propagate-with-conflict-or-not[OF (cdclW-cp++  $S \ U$ )]
        full confl CT unfolding full-def by auto
    qed
    have init-clss  $T = \text{init-clss } S$  and learned-clss  $T = \text{learned-clss } S$ 
      using  $TU$  (init-clss  $U = \text{init-clss } S$ ) (learned-clss  $U = \text{learned-clss } S$ ) by auto
    then have  $D \in \# \text{clauses } S$ 
      using  $TU \text{ confl}$  by (fastforce simp: clauses-def)
    then have  $\neg \text{trail } S \models_{\text{as}} CNot \ D$ 
      using cls-f CT by simp
  moreover
    obtain  $M$  where tr-U: trail  $U = M @ \text{trail } S$  and nm:  $\forall m \in \text{set } M. \neg \text{is-marked } m$ 
      by (metis (mono-tags, lifting) assms(1) full-def rtranclp-cdclW-cp-dropWhile-trail)
    have trail  $U \models_{\text{as}} CNot \ D$ 
      using  $TU \text{ confl}$  by auto
  ultimately obtain  $L$  where  $L \in \# D$  and  $\neg L \in \text{lits-of } M$ 
    unfolding tr-U CNot-def true-annots-def Ball-def true-annot-def true-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: rtrancplp-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: ( $\lambda l. \text{atm-of (lit-of } l)$ ) 'set M
       $\cap (\lambda l. \text{atm-of (lit-of } l))$  'set (trail S) = \{\}
    moreover assume a4:  $x \in \text{set } M$ 
    moreover assume a5:  $xb \in \text{set (trail S)}$ 
    moreover have atm-of ( $\neg L$ ) = atm-of L
      by auto
    ultimately have False
      by auto
  }
  then have LS: atm-of L  $\notin \text{atm-of ' lits-of (trail S)}$ 
    using  $\langle \neg L \in \text{lits-of } M \rangle \langle \text{no-dup (trail U)} \rangle$  unfolding tr-U lits-of-def by auto
ultimately have get-level (trail U) L = backtrack-lvl U
proof (cases get-all-levels-of-marked (trail S)  $\neq []$ , goal-cases)
  case 2 note LD = this(1) and LM = this(2) and inv-U = this(3) and US = this(4) and
    LS = this(5) and ne = this(6)
  have backtrack-lvl S = 0
    using lev ne unfolding cdclW-M-level-inv-def by auto
  moreover have get-rev-level (rev M) 0 L = 0
    using nm by auto
  ultimately show ?thesis using LS ne US unfolding tr-U
    by (simp add: get-all-levels-of-marked-nil-iff-not-is-marked lits-of-def)
next
  case 1 note LD = this(1) and LM = this(2) and inv-U = this(3) and US = this(4) and
    LS = this(5) and ne = this(6)

  have hd (get-all-levels-of-marked (trail S)) = backtrack-lvl S
    using ne lev unfolding cdclW-M-level-inv-def
    by (cases get-all-levels-of-marked (trail S)) auto
  moreover have atm-of L  $\in \text{atm-of ' lits-of } M$ 
    using  $\langle \neg L \in \text{lits-of } M \rangle$  by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def)
  ultimately show ?thesis
    using nm ne unfolding tr-U
    using get-level-skip-beginning-hd-get-all-levels-of-marked[OF LS, of M]
      get-level-skip-in-all-not-marked[of rev M L backtrack-lvl S]
    unfolding lits-of-def US
    by auto
  qed
then show  $\exists L \in \#D. \text{get-level (trail U) } L = \text{backtrack-lvl } U$ 
  using  $\langle L \in \# D \rangle$  by blast
qed

```

qed

17.6.4 Literal of highest level in marked literals

definition *mark-is-false-with-level* :: 'st \Rightarrow bool **where**

mark-is-false-with-level $S' \equiv$

$\forall D\ M1\ M2\ L. \ M1 \text{ @ } \text{Propagated } L\ D \# M2 = \text{trail } S' \longrightarrow D - \{\#L\# \} \neq \{\#\}$
 $\longrightarrow (\exists L. L \in \# \ D \wedge \text{get-level } (\text{trail } S')\ L = \text{get-maximum-possible-level } M1)$

definition *no-more-propagation-to-do*:: 'st \Rightarrow bool **where**

no-more-propagation-to-do $S \equiv$

$\forall D\ M\ M'\ L. D + \{\#L\#\} \in \# \text{ clauses } S \longrightarrow \text{trail } S = M' \text{ @ } M \longrightarrow M \models_{as} CNot\ D$
 $\longrightarrow \text{undefined-lit } M\ L \longrightarrow \text{get-maximum-possible-level } M < \text{backtrack-lvl } S$
 $\longrightarrow (\exists L. L \in \# \ D \wedge \text{get-level } (\text{trail } S)\ L = \text{get-maximum-possible-level } M)$

lemma *propagate-no-more-propagation-to-do*:

assumes *propagate*: *propagate* $S\ S'$

and H : *no-more-propagation-to-do* S

and M : *cdcl_W-M-level-inv* S

shows *no-more-propagation-to-do* S'

using *assms*

proof –

obtain $M\ N\ U\ k\ C\ L$ **where**

S : *state* $S = (M, N, U, k, None)$ **and**

S' : *state* $S' = (\text{Propagated } L\ ((C + \{\#L\#\}))) \# M, N, U, k, None)$ **and**

$C + \{\#L\#\} \in \# \text{ clauses } S$ **and**

$M \models_{as} CNot\ C$ **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 $D-L$: $D + \{\#L'\#\} \in \# \text{ clauses } S'$

and $\text{trail } S' = M2 \text{ @ } M1$

and *get-max*: *get-maximum-possible-level* $M1 < \text{backtrack-lvl } S'$

and $M1 \models_{as} CNot\ D$

and *undef*: *undefined-lit* $M1\ L'$

have $tl\ M2 \text{ @ } M1 = \text{trail } S \vee (M2 = [] \wedge M1 = \text{Propagated } L\ ((C + \{\#L\#\}))) \# M$

using $\langle \text{trail } S' = M2 \text{ @ } M1 \rangle S'\ S$ **by** (*cases* $M2$) *auto*

moreover {

assume $tl\ M2 \text{ @ } M1 = \text{trail } S$

moreover **have** $D + \{\#L'\#\} \in \# \text{ clauses } S$ **using** $D-L\ S\ S'$ **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 $(\text{trail } S)\ L' = \text{get-maximum-possible-level } M1$

using $H\ \langle M1 \models_{as} CNot\ D \rangle \text{undef}$ **unfolding no-more-propagation-to-do-def** **by** *metis*

moreover

{ **have** *cdcl_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 $\langle L' \in \# \ D \rangle \langle M1 \models_{as} CNot\ D \rangle$ **by** (*metis atm-of-uminus image-eq1 in-CNot-implies-uminus*(2))

```

    then have atm-of  $L' \in \text{atm-of } ( \text{ lits-of } M )$ 
    using  $\langle \text{tl } M2 @ M1 = \text{trail } S \rangle S$  by auto
  ultimately have  $\text{atm-of } L \neq \text{atm-of } L'$  unfolding lits-of-def by auto
}
ultimately have  $\exists L' \in \# D. \text{get-level } (\text{trail } S') L' = \text{get-maximum-possible-level } M1$ 
  using  $S S'$  by auto
}
moreover {
  assume  $M2 = []$  and  $M1: M1 = \text{Propagated } L ( (C + \{\#L\# \}) ) \# M$ 
  have  $\text{cdcl}_W\text{-}M\text{-level-inv } S'$ 
    using  $\text{cdcl}_W\text{-consistent-inv}[OF - M] \text{cdcl}_W.\text{propagate}[OF \text{ propagate}]$  by blast
  then have  $\text{get-all-levels-of-marked } (\text{trail } S') = \text{rev } ([\text{Suc } 0..<(\text{Suc } 0+k)])$ 
    using  $S'$  unfolding  $\text{cdcl}_W\text{-}M\text{-level-inv-def}$  by auto
  then have  $\text{get-maximum-possible-level } M1 = \text{backtrack-lvl } S'$ 
    using  $\text{get-maximum-possible-level-max-get-all-levels-of-marked}[of M1] S' M1$ 
    by (auto intro: Max-eqI)
  then have False using get-max by auto
}
ultimately show  $\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{get-maximum-possible-level } M1$  by fast
qed
qed

```

lemma *conflict-no-more-propagation-to-do*:

```

  assumes conflict: conflict  $S S'$ 
  and  $H: \text{no-more-propagation-to-do } S$ 
  and  $M: \text{cdcl}_W\text{-}M\text{-level-inv } S$ 
  shows no-more-propagation-to-do  $S'$ 
  using assms unfolding no-more-propagation-to-do-def conflict.simps by force

```

lemma *cdcl_W-cp-no-more-propagation-to-do*:

```

  assumes conflict: cdclW-cp  $S S'$ 
  and  $H: \text{no-more-propagation-to-do } S$ 
  and  $M: \text{cdcl}_W\text{-}M\text{-level-inv } S$ 
  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 *cdcl_W-then-exists-cdcl_W-stgy-step*:

```

  assumes
     $o: \text{cdcl}_W\text{-}o \ S S'$  and
    alien: no-strange-atm  $S$  and
    lev:  $\text{cdcl}_W\text{-}M\text{-level-inv } S$ 
  shows  $\exists S'. \text{cdcl}_W\text{-stgy } S S'$ 
  proof -
    obtain  $S''$  where full  $\text{cdcl}_W\text{-cp}$   $S' S''$ 
    using always-exists-full-cdclW-cp-step alien cdclW-no-strange-atm-inv cdclW-o-no-more-init-clss
       $o$  other lev by (meson  $\text{cdcl}_W\text{-consistent-inv}$ )
    then show ?thesis

```

using *assms* by (*metis* *always-exists-full-cdcl_W-cp-step* *cdcl_W-stgy.conflict'* *full-unfold* *other'*)
qed

lemma *backtrack-no-decomp*:

assumes *S*: *state* *S* = (*M*, *N*, *U*, *k*, *Some* (*D* + {#*L*#}))
and *L*: *get-level* *M* *L* = *k*
and *D*: *get-maximum-level* *M* *D* < *k*
and *M-L*: *cdcl_W-M-level-inv* *S*
shows $\exists S'. \text{cdcl}_W\text{-o } S S'$

proof –

have *L-D*: *get-level* *M* *L* = *get-maximum-level* *M* (*D* + {#*L*#})
using *L D* by (*simp* *add: get-maximum-level-plus*)
let ?*i* = *get-maximum-level* *M* *D*
obtain *K M1 M2* **where** *K*: (*Marked* *K* (?*i* + 1) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* *M*)
using *backtrack-ex-decomp*[*OF M-L*, *of ?i*] *D S* by *auto*
show ?*thesis* using *backtrack-rule*[*OF S K L L-D*] by (*meson* *bj cdcl_W-bj.simps* *state-eq-ref*)
qed

lemma *cdcl_W-stgy-final-state-conclusive*:

assumes *termi*: $\forall S'. \neg \text{cdcl}_W\text{-stgy } S S'$
and *decomp*: *all-decomposition-implies-m* (*init-clss* *S*) (*get-all-marked-decomposition* (*trail* *S*))
and *learned*: *cdcl_W-learned-clause* *S*
and *level-inv*: *cdcl_W-M-level-inv* *S*
and *alien*: *no-strange-atm* *S*
and *no-dup*: *distinct-cdcl_W-state* *S*
and *confl*: *cdcl_W-conflicting* *S*
and *confl-k*: *conflict-is-false-with-level* *S*
shows (*conflicting* *S* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss* *S*)))
∨ (*conflicting* *S* = *None* ∧ *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* = *Some* {#}
∨ *conflicting* *S* = *None*
∨ ($\exists D L. \text{conflicting } S = \text{Some } (D + \{\#L\# \})$)
apply (*cases* *conflicting* *S*, *auto*)
by (*rename-tac* *C*, *case-tac* *C*, *auto*)
moreover {
assume *conflicting* *S* = *Some* {#}
then have *unsatisfiable* (*set-mset* (*init-clss* *S*))
using *assms*(β) **unfolding** *cdcl_W-learned-clause-def* *true-clss-cls-def*
by (*metis* (*no-types*, *lifting*) *Un-insert-right* *atms-of-empty* *satisfiable-def*
sup-bot.right-neutral *total-over-m-insert* *total-over-set-empty* *true-clss-empty*)
}
moreover {
assume *conflicting* *S* = *None*
{ **assume** $\neg ?M \models_{asm} ?N$
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  $\neg ?B \subseteq ?A$ 
    then obtain  $l$  where  $l \in ?B$  and  $l \notin ?A$  by auto
    then have undefined-lit  $?M$  (Pos  $l$ )
      using  $\langle l \notin ?A \rangle$  unfolding lits-of-def by (auto simp add: defined-lit-map)
    then have  $\exists S'. \text{cdcl}_W\text{-o } S S'$ 
      using  $\text{cdcl}_W\text{-o.decide decide.intros } \langle l \in ?B \rangle$  no-strange-atm-def
      by (metis  $\langle \text{conflicting } S = \text{None} \rangle$  literal.sel(1) state-eq-def)
    then show False
      using termi  $\text{cdcl}_W\text{-then-exists-cdcl}_W\text{-stgy-step}[OF - \text{alien}]$  level-inv by blast
  qed
  qed
  obtain  $D$  where  $\neg ?M \models_a D$  and  $D \in \# ?N$ 
    using  $\langle \neg ?M \models_{asm} ?N \rangle$  unfolding lits-of-def true-annots-def Ball-def by auto
  have  $\text{atms-of } D \subseteq \text{atm-of } ' (lits\text{-of } ?M)$ 
    using  $\langle D \in \# ?N \rangle$  unfolding  $\langle \text{atm-of } ' (lits\text{-of } ?M) = \text{atms-of-msu } ?N \rangle$  atms-of-ms-def
    by (auto simp add: atms-of-def)
  then have  $a1: \text{atm-of } ' \text{set-mset } D \subseteq \text{atm-of } ' \text{lits-of } (\text{trail } S)$ 
    by (auto simp add: atms-of-def lits-of-def)
  have total-over-m  $(lits\text{-of } ?M) \{D\}$ 
    using  $\langle \text{atms-of } D \subseteq \text{atm-of } ' (lits\text{-of } ?M) \rangle$  atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
    by (fastforce simp: total-over-set-def)
  then have  $?M \models_{as} CNot D$ 
    using total-not-true-cls-true-clss-CNot  $\langle \neg \text{trail } S \models_a D \rangle$  true-annot-def
    true-annots-true-cls by fastforce
  then have False
    proof -
      obtain  $S'$  where
        f2: full  $\text{cdcl}_W\text{-cp } S S'$ 
        by (meson alien always-exists-full-cdclW-cp-step level-inv)
      then have  $S' = S$ 
        using  $\text{cdcl}_W\text{-stgy.conflict'}$ [of  $S$ ] by (metis (no-types) full-unfold termi)
      then show ?thesis
        using f2  $\langle D \in \# \text{init-clss } S \rangle$   $\langle \text{conflicting } S = \text{None} \rangle$   $\langle \text{trail } S \models_{as} CNot D \rangle$ 
        clauses-def full-cdclW-cp-not-any-negated-init-clss by auto
    qed
  }
  then have  $?M \models_{asm} ?N$  by blast
}
moreover {
  assume  $\exists D L. \text{conflicting } S = \text{Some } (D + \{\#L\# \})$ 
  then obtain  $D L$  where  $LD: \text{conflicting } S = \text{Some } (D + \{\#L\# \})$  and  $\text{lev-L: get-level } ?M L = ?k$ 
    by (metis (mono-tags) bex-msetE confl-k insert-DiffM2 multi-self-add-other-not-self
        union-eq-empty)
  let  $?D = D + \{\#L\# \}$ 
  have  $?D \neq \{\# \}$  by auto
  have  $?M \models_{as} CNot ?D$  using confl  $LD$  unfolding  $\text{cdcl}_W\text{-conflicting-def}$  by auto
  then have  $?M \neq []$  unfolding true-annots-def Ball-def true-annot-def true-cls-def by force
  { have  $M: ?M = \text{hd } ?M \# \text{tl } ?M$  using  $\langle ?M \neq [] \rangle$  list.collapse by fastforce
    assume marked: is-marked (hd  $?M$ )
    then obtain  $k'$  where  $k': k' + 1 = ?k$ 
      using level-inv  $M$  unfolding  $\text{cdcl}_W\text{-M-level-inv-def}$ 
      by (cases hd (trail  $S$ ); cases trail  $S$ ) auto
    obtain  $L' l'$  where  $L': \text{hd } ?M = \text{Marked } L' l'$  using marked by (cases hd  $?M$ ) auto
    have marked-hd-tl: get-all-levels-of-marked (hd (trail  $S$ ) # tl (trail  $S$ ))
      = rev  $[1..<1 + \text{length } (\text{get-all-levels-of-marked } ?M)]$ 
  }
}

```

```

using level-inv lev-L M unfolding cdclW-M-level-inv-def M[symmetric]
by blast
then have l'-tl: l' # 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..length (get-all-levels-of-marked ?M)]
  using M Suc-le-mono calculation by (fastforce simp add: upt.simps(2))
finally have
  l' = ?k and
  g-r: get-all-levels-of-marked (tl (trail S))
    = rev [1..length (get-all-levels-of-marked (trail S))]
  using level-inv lev-L M unfolding cdclW-M-level-inv-def by auto
have *:  $\bigwedge \text{list. no-dup list} \implies$ 
  -  $L \in \text{lits-of list} \implies \text{atm-of } L \in \text{atm-of ' lits-of list}$ 
  by (metis atm-of-uminus imageI)
have L' = -L
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  moreover have  $-L \in \text{lits-of } ?M$  using confl LD unfolding cdclW-conflicting-def by auto
  ultimately have get-level (hd (trail S) # tl (trail S)) L = get-level (tl ?M) L
    using cdclW-M-level-inv-decomp(1)[OF level-inv] unfolding L' consistent-interp-def
    by (metis (no-types, lifting) L' M atm-of-eq-atm-of get-level-skip-beginning insert-iff
      lits-of-cons marked-lit.sel(1))

  moreover
    have length (get-all-levels-of-marked (trail S)) = ?k
      using level-inv unfolding cdclW-M-level-inv-def by auto
    then have Max (set (0 # get-all-levels-of-marked (tl (trail S)))) = ?k - 1
      unfolding g-r by (auto simp add: Max-n-upt)
    then have get-level (tl ?M) L < ?k
      using get-maximum-possible-level-ge-get-level[of tl ?M L]
      by (metis One-nat-def add.right-neutral add-Suc-right diff-add-inverse2
        get-maximum-possible-level-max-get-all-levels-of-marked k' le-imp-less-Suc
        list.simps(15))
    finally show False using lev-L M by auto
  qed
have L: hd ?M = Marked (-L) ?k using l' = ?k L' = -L L' by auto

have g-a-l: get-all-levels-of-marked ?M = rev [1..1 + ?k]
  using level-inv lev-L M unfolding cdclW-M-level-inv-def by auto
have g-k: get-maximum-level (trail S) D ≤ ?k
  using get-maximum-possible-level-ge-get-maximum-level[of ?M]
  get-maximum-possible-level-max-get-all-levels-of-marked[of ?M]
  by (auto simp add: Max-n-upt g-a-l)
have get-maximum-level (trail S) D < ?k
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then have get-maximum-level (trail S) D = ?k using M g-k unfolding L by auto
  then obtain L' where L' ∈ # D and L-k: get-level ?M L' = ?k
    using get-maximum-level-exists-lit[of ?k ?M D] unfolding k'[symmetric] by auto
  have  $L \neq L'$  using no-dup 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  $\neg ?thesis$ 
then have  $get\_level\ ?M\ L' = get\_level\ (tl\ ?M)\ L'$ 
  using  $M\ \langle L \neq L' \rangle\ get\_level\_skip\_beginning[of\ L'\ hd\ ?M\ tl\ ?M]$  unfolding  $L$ 
  by  $(auto\ simp: atm\_of\_eq\_atm\_of)$ 
moreover have  $\dots < ?k$ 
proof -
  { assume  $a1: get\_level\ (tl\ (trail\ S))\ L' = backtrack\_lvl\ S$ 
    assume  $a2: rev\ (get\_all\_levels\_of\_marked\ (tl\ (trail\ S))) =$ 
       $[Suc\ 0..<backtrack\_lvl\ S]$ 
    have  $k' + Suc\ 0 = backtrack\_lvl\ S$ 
    using  $k'$  by presburger
    then have False
    using  $a2\ a1$  by  $(metis\ (no\_types)\ Max\_n\_upt\ Zero\_neq\_Suc\ add\_diff\_cancel\_left'\$ 
       $add\_diff\_cancel\_right'\ diff\_is\_0\_eq$ 
       $get\_all\_levels\_of\_marked\_rev\_eq\_rev\_get\_all\_levels\_of\_marked$ 
       $get\_rev\_level\_less\_max\_get\_all\_levels\_of\_marked\ list.set(2)\ set\_upt)$ 
    }
  then show  $?thesis$ 
    using  $g\_r\ get\_rev\_level\_less\_max\_get\_all\_levels\_of\_marked[of\ rev\ (tl\ ?M)\ 0\ L]$ 
     $l'\text{-}tl\ calculation[symmetric]\ g\_a\_l\ L\_k$ 
    by  $(auto\ simp: Max\_n\_upt\ cdcl\_W\text{-}M\text{-}level\_inv\_def\ rev\_swap[symmetric])$ 
  qed
  finally show False using  $L\_k$  by simp
qed
then have  $taut: tautology\ (D + \{\#L\# \})$ 
  using  $\langle L' \in \# D \rangle$  by  $(metis\ add.commute\ mset\_leD\ mset\_le\_add\_left\ multi\_member\_this$ 
     $tautology\_minus)$ 
have consistent\_interp  $(lits\_of\ ?M)$ 
  using level\_inv unfolding  $cdcl\_W\text{-}M\text{-}level\_inv\_def$  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  $cdcl\_W\text{-}conflicting\_def$  by auto
ultimately show False by blast
qed
then have False
  using backtrack-no-decomp  $[OF\ -\ \langle get\_level\ (trail\ S)\ L = backtrack\_lvl\ S \rangle\ -\ level\_inv]$ 
   $LD\ alien\ termi$  by  $(metis\ cdcl\_W\text{-}then\_exists\_cdcl\_W\text{-}stgy\_step\ level\_inv)$ 
}
moreover {
  assume  $\neg is\_marked\ (hd\ ?M)$ 
  then obtain  $L'\ C$  where  $L'C: hd\ ?M = Propagated\ L'\ C$  by  $(cases\ hd\ ?M,\ auto)$ 
  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  $cdcl\_W\text{-}conflicting\_def$  by  $(metis\ append\_Nil\ diff\_single\_eq\_union)$ 
  { assume  $-L' \notin \# ?D$ 
    then have False
    using  $bj[OF\ cdcl\_W\text{-}bj.skip[OF\ skip\_rule[OF\ -\ \langle -L' \notin \# ?D \rangle\ \langle ?D \neq \{\#\} \rangle,\ of\ S\ C\ tl\ (trail\ S)\ -$ 
       $]]]$ 
     $termi\ M$  by  $(metis\ LD\ alien\ cdcl\_W\text{-}then\_exists\_cdcl\_W\text{-}stgy\_step\ state\_eq\_def\ level\_inv)$ 
  }
}
moreover {
  assume  $-L' \in \# ?D$ 
  then obtain  $D'$  where  $D': ?D = D' + \{\#-L'\#\}$  by  $(metis\ insert\_DiffM2)$ 

```

```

have g-r: get-all-levels-of-marked (Propagated L' C # tl (trail S))
  = rev [Suc 0..<Suc (length (get-all-levels-of-marked (trail S)))]
  using level-inv M unfolding cdclW-M-level-inv-def by auto
have Max (insert 0 (set (get-all-levels-of-marked (Propagated L' C # tl (trail S))))) = ?k
  using level-inv M unfolding g-r cdclW-M-level-inv-def set-rev
  by (auto simp add:Max-n-upt)
then have get-maximum-level (Propagated L' C # tl ?M) D' ≤ ?k
  using get-maximum-possible-level-ge-get-maximum-level[of Propagated L' C # tl ?M]
  unfolding get-maximum-possible-level-max-get-all-levels-of-marked by auto
then have get-maximum-level (Propagated L' C # tl ?M) D' = ?k
  ∨ get-maximum-level (Propagated L' C # tl ?M) D' < ?k
  using le-neq-implies-less by blast
moreover {
  assume g-D'-k: get-maximum-level (Propagated L' C # tl ?M) D' = ?k
  have False
  proof -
    have f1: get-maximum-level (trail S) D' = backtrack-lvl S
      using M g-D'-k by auto
    have (trail S, init-clss S, learned-clss S, backtrack-lvl S, Some (D + {#L#}))
      = state S
      by (metis (no-types) LD)
    then have cdclW-o S (update-conflicting (Some (D' #U 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 (Propagated L' C # tl ?M) D' < ?k
  then have False
  proof -
    assume a1: get-maximum-level (Propagated L' C # tl (trail S)) D' < backtrack-lvl S
    obtain mm :: 'v literal multiset and ll :: 'v literal where
      f2: conflicting S = Some (mm + {#ll#})
      get-level (trail S) ll = backtrack-lvl S
      using LD (get-level (trail S) L = backtrack-lvl S) by blast
    then have f3: get-maximum-level (trail S) D' ≤ get-level (trail S) ll
      using M a1 by force
    have lev-neq: get-level (trail S) ll ≠ get-maximum-level (trail S) D'
      using f2 M calculation(2) by presburger
    have f1: trail S = Propagated L' C # tl (trail S)
      conflicting S = Some (D' + {#- L'#})
      using D' LD M by force+
    have f2: conflicting S = Some (mm + {#ll#})
      get-level (trail S) ll = backtrack-lvl S
      using f2 by force+
    have ll = - L'
      by (metis (no-types) D' LD lev-neq option.inject f2 f3 le-antisym
        get-maximum-level-ge-get-level insert-noteq-member)
    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'  $\implies$  cdclW++ S S'
  apply (induct rule: cdclW-cp.induct)
  by (meson cdclW.conflict cdclW.propagate tranclp.r-into-trancl tranclp.trancl-into-trancl) +

```

```

lemma tranclp-cdclW-cp-tranclp-cdclW:
  cdclW-cp++ S S'  $\implies$  cdclW++ S S'
  apply (induct rule: tranclp.induct)
  apply (simp add: cdclW-cp-tranclp-cdclW)
  by (meson cdclW-cp-tranclp-cdclW tranclp-trans)

```

```

lemma cdclW-stgy-tranclp-cdclW:
  cdclW-stgy S S'  $\implies$  cdclW++ S S'
proof (induct rule: cdclW-stgy.induct)
  case conflict'
  then show ?case
    unfolding full1-def by (simp add: tranclp-cdclW-cp-tranclp-cdclW)
next
  case (other' S' S'')
  then have S' = S''  $\vee$  cdclW-cp++ S' S''
    by (simp add: rtranclp-unfold full-def)
  then show ?case
    using other' by (meson cdclW.other cdclW-axioms tranclp.r-into-trancl
      tranclp-cdclW-cp-tranclp-cdclW tranclp-trans)
qed

```

```

lemma tranclp-cdclW-stgy-tranclp-cdclW:
  cdclW-stgy++ S S'  $\implies$  cdclW++ S S'
  apply (induct rule: tranclp.induct)
  using cdclW-stgy-tranclp-cdclW apply blast
  by (meson cdclW-stgy-tranclp-cdclW tranclp-trans)

```

```

lemma rtranclp-cdclW-stgy-rtranclp-cdclW:
  cdclW-stgy** S S'  $\implies$  cdclW** S S'
  using rtranclp-unfold[of cdclW-stgy S S'] tranclp-cdclW-stgy-tranclp-cdclW[of S S'] by auto

```

```

lemma cdclW-o-conflict-is-false-with-level-inv:
  assumes
    cdclW-o S S' and
    lev: cdclW-M-level-inv S and
    cnfl-inv: conflict-is-false-with-level S and
    n-d: distinct-cdclW-state S and
    conflicting: cdclW-conflicting S
  shows conflict-is-false-with-level S'
  using assms(1,2)
proof (induct rule: cdclW-o-induct-lev2)

```

```

case (resolve L C M D T) note tr-S = this(1) and confl = this(2) and T = this(4)
have -L  $\notin\#$  D using n-d confl unfolding distinct-cdclW-state-def distinct-mset-def by auto
moreover have L  $\notin\#$  D
proof (rule ccontr)
  assume  $\neg$  ?thesis
  moreover have Propagated L (C + {#L#}) # M  $\models_{as}$  CNot D
    using conflicting confl tr-S unfolding cdclW-conflicting-def by auto
  ultimately have -L  $\in$  lits-of (Propagated L (C + {#L#})) # M
    using in-CNot-implies-uminus(2) by blast
  moreover have no-dup (Propagated L (C + {#L#})) # M
    using lev tr-S unfolding cdclW-M-level-inv-def by auto
  ultimately show False unfolding lits-of-def by (metis consistent-interp-def image-eqI
    list.set-intros(1) lits-of-def marked-lit.sel(2) distinctconsistent-interp)
qed

ultimately
  have g-D: get-maximum-level (Propagated L (C + {#L#}) # M) D
    = get-maximum-level M D
  proof -
    have  $\forall a f L. ((a::'v) \in f \text{ ' } L) = (\exists l. (l::'v \text{ literal}) \in L \wedge a = f l)$ 
      by blast
    then show ?thesis
      using get-maximum-level-skip-first[of L D (C + {#L#}) M] unfolding atms-of-def
      by (metis (no-types)  $\langle - L \notin\# D \rangle \langle L \notin\# D \rangle$  atm-of-eq-atm-of mem-set-mset-iff)
    qed
  { assume
      get-maximum-level (Propagated L (C + {#L#}) # M) D = backtrack-lvl S and
      backtrack-lvl S > 0
    then have D: get-maximum-level M D = backtrack-lvl S unfolding g-D by blast
    then have ?case
      using tr-S  $\langle$ backtrack-lvl S > 0 $\rangle$  get-maximum-level-exists-lit[of backtrack-lvl S M D] T
      by auto
    }
  moreover {
    assume [simp]: backtrack-lvl S = 0
    have  $\bigwedge L. \text{get-level } M L = 0$ 
    proof -
      fix L
      have atm-of L  $\notin$  atm-of ' (lits-of M)  $\implies$  get-level M L = 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.. $\leq$ Suc (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 M L = 0
          using get-maximum-possible-level-ge-get-level[of M L]
          unfolding get-maximum-possible-level-max-get-all-levels-of-marked by auto
      }
    ultimately show get-level M L = 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 ∈# D and get-level (Propagated L C' # M) La = backtrack-lvl S
  using skip confl-inv by auto
moreover
have atm-of La ≠ atm-of L
proof (rule ccontr)
  assume ¬ ?thesis
  then have La: La = L using ⟨La ∈# D⟩ ⟨- L ∉# D⟩ by (auto simp add: atm-of-eq-atm-of)
  have Propagated L C' # M ⊨as CNot D
    using conflicting tr-S D unfolding cdclW-conflicting-def by auto
  then have -L ∈ lits-of M
    using ⟨La ∈# 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 (Propagated L C' # M) La = get-level M La 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 ∨ (∃ S'. propagate** S S' ∧ conflict S' T)*
using *assms* **by** *induction blast+*

lemma *rtrancpl-cdcl_W-cp-propagate-confl*:
assumes *cdcl_W-cp** S T*
shows *propagate** S T ∨ (∃ S'. propagate** S S' ∧ conflict S' T)*
by (*simp add: assms rtrancpl-cdcl_W-cp-propa-or-propa-confl*)

lemma *cdcl_W-cp-propagate-completeness*:
assumes *MN: set M ⊨_s set-mset N* **and**
cons: consistent-interp (set M) **and**
tot: total-over-m (set M) (set-mset N) **and**
lits-of (trail S) ⊆ set M **and**
init-clss S = N **and**
*propagate** S S'* **and**
learned-clss S = {#}
shows *length (trail S) ≤ length (trail S') ∧ lits-of (trail S') ⊆ set M*
using *assms(6,4,5,7)*

proof (*induction rule: rtrancpl-induct*)

case *base*
then show ?case **by** *auto*

next

case (*step Y Z*)
note *st = this(1)* **and** *propa = this(2)* **and** *IH = this(3)* **and** *lits' = this(4)* **and** *NS = this(5)* **and**
learned = this(6)
then have *len: length (trail S) ≤ length (trail Y)* **and** *LM: lits-of (trail Y) ⊆ set M*
by *blast+*

obtain *M' N' U k C L* **where**

Y: state Y = (M', N', U, k, None) **and**
Z: state Z = (Propagated L (C + {#L#}) # M', N', U, k, None) **and**
C: C + {#L#} ∈# clauses Y **and**

```

   $M'-C: M' \models_{as} CNot\ C$  and
  undefined-lit (trail  $Y$ )  $L$ 
  using propa by auto
  have init-clss  $S = init-clss\ Y$ 
  using st by induction auto
  then have [simp]:  $N' = N$  using NS  $Y\ Z$  by simp
  have learned-clss  $Y = \{\#\}$ 
  using st learned by induction auto
  then have [simp]:  $U = \{\#\}$  using  $Y$  by auto
  have set  $M \models_s CNot\ C$ 
  using  $M'-C\ LM\ Y$  unfolding true-annot-def Ball-def true-annot-def true-clss-def true-clss-def
  by force
  moreover
  have set  $M \models C + \{\#L\#$ 
  using MN  $C\ learned\ Y$  unfolding true-clss-def clauses-def
  by (metis NS  $\langle init-clss\ S = init-clss\ Y \rangle \langle learned-clss\ Y = \{\#\} \rangle$  add.right-neutral
    mem-set-mset-iff)
  ultimately have  $L \in set\ M$  by (simp add: cons consistent-CNot-not)
  then show ?case using LM len  $Y\ Z$  by auto
qed

```

lemma *completeness-is-a-full1-propagation:*

```

  fixes  $S :: 'st$  and  $M :: 'v$  literal list
  assumes MN: set  $M \models_s set-mset\ N$ 
  and cons: consistent-interp (set  $M$ )
  and tot: total-over-m (set  $M$ ) (set-mset  $N$ )
  and alien: no-strange-atm  $S$ 
  and learned: learned-clss  $S = \{\#\}$ 
  and clsS[simp]: init-clss  $S = N$ 
  and lits: lits-of (trail  $S$ )  $\subseteq set\ M$ 
  shows  $\exists S'. propagate^{**}\ S\ S' \wedge full\ cdcl_W-cp\ S\ S'$ 
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)  $\exists X. propagate^{**}\ S\ X \wedge conflict\ X\ S'$ 
  using rtranclp-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-annot-def true-annot-def true-clss-def true-cls-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 $cdcl_W-cp^{**}\ ?S\ ?S' \implies \exists M. trail\ ?S' = M @ trail\ ?S \wedge (\forall l \in set\ M. \neg\ is-marked\ l)$

lemma *rtrancp-propagate-is-trail-append*:
 $propagate^{**}\ S\ T \implies \exists c. trail\ T = c @ trail\ S$
by (*induction rule: rtrancp-induct*) *auto*

lemma *rtrancp-propagate-is-update-trail*:
 $propagate^{**}\ S\ T \implies cdcl_W-M-level-inv\ S \implies T \sim delete-trail-and-rebuild\ (trail\ T)\ S$
proof (*induction rule: rtrancp-induct*)
case *base*
then show $?case$ **unfolding** *state-eq-def* **by** (*auto simp: cdcl_W-M-level-inv-decomp state-access-simp*)
next
case (*step* $T\ U$) **note** $IH=this(3)[OF\ this(4)]$
moreover have $cdcl_W-M-level-inv\ U$
using *rtrancp-cdcl_W-consistent-inv* $\langle propagate^{**}\ S\ T \rangle \langle propagate\ T\ U \rangle$
rtrancp-mono[*of* *propagate* $cdcl_W$] $cdcl_W-cp-consistent-inv\ propagate'$
rtrancp-propagate-is-rtrancp-cdcl_W *step.prem*s **by** *blast*
then have *no-dup* ($trail\ U$) **unfolding** $cdcl_W-M-level-inv-def$ **by** *auto*
ultimately show $?case$ **using** $\langle propagate\ T\ U \rangle$ **unfolding** *state-eq-def*
by (*fastforce simp: state-access-simp*)
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**
 $length: n \leq length\ M$
shows
 $\exists M' k\ S. length\ M' \geq n \wedge$
 $lits-of\ M' \subseteq set\ M \wedge$
 $no-dup\ M' \wedge$
 $S \sim update-backtrack-lvl\ k\ (append-trail\ (rev\ M')\ (init-state\ N)) \wedge$
 $cdcl_W-stgy^{**}\ (init-state\ N)\ S$
using *length*
proof (*induction n*)
case 0
have $update-backtrack-lvl\ 0\ (append-trail\ (rev\ [])\ (init-state\ N)) \sim init-state\ N$
by (*auto simp: state-eq-def simp del: state-simp*)
moreover have
 $0 \leq length\ []$ **and**
 $lits-of\ [] \subseteq set\ M$ **and**
 $cdcl_W-stgy^{**}\ (init-state\ N)\ (init-state\ N)$
and *no-dup* []

```

    by (auto simp: state-eq-def simp del: state-simp)
ultimately show ?case using state-eq-sym by blast
next
case (Suc n) note IH = this(1) and n = this(2)
then obtain M' k S where
  l-M': length M' ≥ n and
  M': lits-of M' ⊆ set M and
  n-d[simp]: no-dup M' and
  S: S ∼ update-backtrack-lvl k (append-trail (rev M') (init-state N)) and
  st: cdclW-stgy** (init-state N) S
  by auto
have
  M: cdclW-M-level-inv S and
  alien: no-strange-atm S
  using rtrancplp-cdclW-consistent-inv[OF rtrancplp-cdclW-stgy-rtrancplp-cdclW[OF st]]
  rtrancplp-cdclW-no-strange-atm-inv[OF rtrancplp-cdclW-stgy-rtrancplp-cdclW[OF st]]
  S unfolding state-eq-def cdclW-M-level-inv-def no-strange-atm-def by auto
{ assume no-step: ¬no-step propagate S

  obtain S' where S': propagate** S S' and full: full cdclW-cp S S'
  using completeness-is-a-full1-propagation[OF assms(1-3), of S] alien M' S
  by (auto simp: state-access-simp)
  have lev: cdclW-M-level-inv S'
  using M S' rtrancplp-cdclW-consistent-inv rtrancplp-propagate-is-rtrancplp-cdclW by blast
  then have n-d'[simp]: no-dup (trail S')
  unfolding cdclW-M-level-inv-def by auto
  have length (trail S) ≤ length (trail S') ∧ lits-of (trail S') ⊆ set M
  using S' full cdclW-cp-propagate-completeness[OF assms(1-3), of S] M' S
  by (auto simp: state-access-simp)
  moreover
  have full: full1 cdclW-cp S S'
  using full no-step no-step-cdclW-cp-no-conflict-no-propagate(2) unfolding full1-def full-def
  rtrancplp-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 rtrancplpD trancplpD)
  have trail S = M' using S by (auto simp: state-access-simp)
  with propa have length (trail S') > n
  using l-M' propa by (induction rule: trancplp.induct) auto
  moreover
  have stS': cdclW-stgy** (init-state N) S'
  using st cdclW-stgy.conflict'[OF full] by auto
  then have init-clss S' = N using stS' rtrancplp-cdclW-stgy-no-more-init-clss by fastforce
  moreover
  have
    [simp]: learned-clss S' = {#} and
    [simp]: init-clss S' = init-clss S and
    [simp]: conflicting S' = None
  using trancplp-into-rtrancplp[OF ⟨propagate++ S S'⟩] S
  rtrancplp-propagate-is-update-trail[of S S'] S M unfolding state-eq-def
  by (auto simp: state-access-simp)
  have S-S': S' ∼ update-backtrack-lvl (backtrack-lvl S')
  (append-trail (rev (trail S')) (init-state N)) using S
  by (auto simp: state-eq-def state-access-simp simp del: state-simp)
  have cdclW-stgy** (init-state (init-clss S')) S'

```

```

    apply (rule rtrancp.rtrancI-into-rtrancI)
    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  $S-S'$  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-confI: no-step conflict  $S$ 
    proof -
      { fix  $D$ 
        assume  $D \in \# N$  and  $M' \models_{as} CNot\ D$ 
        then have set  $M \models D$  using MN unfolding true-clss-def by auto
        moreover have set  $M \models_s CNot\ D$ 
          using ⟨ $M' \models_{as} CNot\ D$ ⟩  $M'$ 
          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: conflict.simps true-clss-def state-access-simp)
  qed
  have lenM: length  $M = \text{card } (\text{set } M)$  using distM by (induction  $M$ ) auto
  have no-dup  $M'$  using  $S\ M$  unfolding cdclW-M-level-inv-def by auto
  then have card (lits-of  $M'$ ) = length  $M'$ 
    by (induction  $M'$ ) (auto simp add: lits-of-def card-insert-if)
  then have lits-of  $M' \subset \text{set } M$ 
    using  $n\ M'\ n'\ \text{lenM}$  by auto
  then obtain  $m$  where  $m$ :  $m \in \text{set } M$  and undef- $m$ :  $m \notin \text{lits-of } M'$  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  $m \in \text{atms-of-msu } (\text{init-clss } S)$ 
    using atm-incl calculation  $S$  by (auto simp: state-access-simp)
  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 simp: state-access-simp)
  let ? $S'$  = cons-trail (Marked  $m\ (k+1)$ ) (incr-lvl  $S$ )
  have lits-of (trail ? $S'$ )  $\subseteq \text{set } M$  using  $m\ M'\ S\ \text{undef}$  by (auto simp: state-access-simp)
  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 ? $S'$ ]  $S\ \text{undef}$ 
    by (auto simp: state-access-simp)
  have cdclW-M-level-inv ? $S'$ 
    using  $M$  dec rtrancp-mono[of decide cdclW] by (meson cdclW-consistent-inv decide other)

```

```

then have lev'': cdclW-M-level-inv S''
  using S'' rtrancpl-cdclW-consistent-inv rtrancpl-propagate-is-rtrancpl-cdclW by blast
then have n-d'': no-dup (trail S'')
  unfolding cdclW-M-level-inv-def by auto
have length (trail ?S') ≤ length (trail S'') ∧ lits-of (trail S'') ⊆ set M
  using S'' full cdclW-cp-propagate-completeness[OF assms(1-3), of ?S' S''] m M' S undef
  by (simp add: state-access-simp)
then have Suc n ≤ length (trail S'') ∧ lits-of (trail S'') ⊆ set M
  using l-M' S undef by (auto simp: state-access-simp)
moreover
  have cdclW-M-level-inv (cons-trail (Marked m (Suc (backtrack-lvl S))))
    (update-backtrack-lvl (Suc (backtrack-lvl S)) S))
    using S (cdclW-M-level-inv (cons-trail (Marked m (k + 1)) (incr-lvl S))) by auto
  then have S'': S'' ∼ update-backtrack-lvl (backtrack-lvl S'')
    (append-trail (rev (trail S'')) (init-state N))
    using rtrancpl-propagate-is-update-trail[OF S''] S undef n-d'' lev''
    by (auto simp del: state-simp simp: state-eq-def state-access-simp)
  then have cdclW-stgy** (init-state N) S''
    using cdclW-stgy.intros(2)[OF decide[OF dec] - full] no-step no-confl st
    by (auto simp: cdclW-cp.simps)
  ultimately show ?thesis using S'' n-d'' by blast
qed
}
ultimately show ?case by blast
qed

```

lemma cdcl_W-stgy-strong-completeness:

```

assumes MN: set M ⊨s set-mset N
and cons: consistent-interp (set M)
and tot: total-over-m (set M) (set-mset N)
and atm-incl: atm-of ' (set M) ⊆ atms-of-msu N
and distM: distinct M

```

shows

```

  ∃ M' k S.
    lits-of M' = set M ∧
    S ∼ update-backtrack-lvl k (append-trail (rev M') (init-state N)) ∧
    cdclW-stgy** (init-state N) S ∧
    final-cdclW-state S

```

proof –

```

from cdclW-stgy-strong-completeness-n[OF assms, of length M]
obtain M' k T where
  l: length M ≤ length M' and
  M'-M: lits-of M' ⊆ set M and
  no-dup: no-dup M' and
  T: T ∼ update-backtrack-lvl k (append-trail (rev M') (init-state N)) and
  st: cdclW-stgy** (init-state N) T
  by auto
have card (set M) = length M using distM by (simp add: distinct-card)
moreover
  have cdclW-M-level-inv T
    using rtrancpl-cdclW-stgy-consistent-inv[OF st] T by auto
  then have card (set ((map (λl. atm-of (lit-of l)) M'))) = length M'
    using distinct-card no-dup by fastforce
moreover have card (lits-of M') = card (set ((map (λl. atm-of (lit-of l)) M')))
  using no-dup unfolding lits-of-def apply (induction M') by (auto simp add: card-insert-if)

```


ultimately have $\text{card } (\text{set } M) \leq \text{card } (\text{lits-of } M')$ using l unfolding lits-of-def by *auto*
 then have $\text{set } M = \text{lits-of } M'$
 using $M'-M$ *card-seteq* by *blast*
 moreover
 then have $M' \models_{asm} N$
 using MN unfolding *true-annots-def* *Ball-def* *true-annot-def* *true-cls-def* by *auto*
 then have *final-cdcl_W-state* T
 using T *no-dup* unfolding *final-cdcl_W-state-def* by (*auto simp: state-access-simp*)
 ultimately show *?thesis* using *st T* by *blast*
 qed

17.6.6 No conflict with only variables of level less than backtrack level

This invariant is stronger than the previous argument in the sense that it is a property about all possible conflicts.

definition *no-smaller-confl* ($S::'st$) \equiv
 $(\forall M K i M' D. M' @ \text{Marked } K i \# M = \text{trail } S \longrightarrow D \in \# \text{ clauses } S$
 $\longrightarrow \neg M \models_{as} \text{CNot } D)$

lemma *no-smaller-confl-init-sate*[*simp*]:
no-smaller-confl (*init-state* N) unfolding *no-smaller-confl-def* by *auto*

lemma *cdcl_W-o-no-smaller-confl-inv*:

fixes $S S' :: 'st$
assumes
cdcl_W-o $S S'$ **and**
lev: *cdcl_W-M-level-inv* S **and**
max-lev: *conflict-is-false-with-level* S **and**
smaller: *no-smaller-confl* S **and**
no-f: *no-clause-is-false* S
shows *no-smaller-confl* S'
 using *assms*(1,2) unfolding *no-smaller-confl-def*
proof (*induct rule: cdcl_W-o-induct-lev2*)
 case (*decide* $L T$) **note** *confl* = *this*(1) **and** *undef* = *this*(2) **and** $T = \text{this}(4)$
 have [*simp*]: *clauses* $T = \text{clauses } S$
 using T *undef* by *auto*
 show *?case*
proof (*intro allI impI*)
 fix $M'' K i M' Da$
 assume $M'' @ \text{Marked } K i \# M' = \text{trail } T$
 and $D: Da \in \# \text{ local.clauses } T$
 then have $tl M'' @ \text{Marked } K i \# M' = \text{trail } S$
 $\vee (M'' = [] \wedge \text{Marked } K i \# M' = \text{Marked } L (\text{backtrack-lvl } S + 1) \# \text{trail } S)$
 using T *undef* by (*cases* M'') *auto*
 moreover {
 assume $tl M'' @ \text{Marked } K i \# M' = \text{trail } S$
 then have $\neg M' \models_{as} \text{CNot } Da$
 using $D T$ *undef no-f confl smaller* unfolding *no-smaller-confl-def smaller* by *fastforce*
 }
 moreover {
 assume $\text{Marked } K i \# M' = \text{Marked } L (\text{backtrack-lvl } S + 1) \# \text{trail } S$
 then have $\neg M' \models_{as} \text{CNot } Da$ using *no-f D confl T* by *auto*
 }
 ultimately show $\neg M' \models_{as} \text{CNot } Da$ by *fast*
 qed

```

next
  case resolve
  then show ?case using smaller no-f max-lev unfolding no-smaller-confl-def by auto
next
  case skip
  then show ?case using smaller no-f max-lev unfolding no-smaller-confl-def by auto
next
  case (backtrack K i M1 M2 L D T) note decomp = this(1) and confl = this(3) and undef = this(6)
    and T = this(7)
  obtain c where M: trail S = c @ M2 @ Marked K (i+1) # M1
    using decomp by auto

  show ?case
  proof (intro allI impI)
    fix M ia K' M' Da
    assume M' @ Marked K' ia # M = trail T
    then have tl M' @ Marked K' ia # M = M1
      using T decomp undef lev by (cases M') (auto simp: cdclW-M-level-inv-decomp)
    assume D: Da ∈ # clauses T
    moreover{
      assume Da ∈ # clauses S
      then have  $\neg M \models_{as} CNot\ Da$  using  $\langle tl\ M' @\ Marked\ K'\ ia\ \# \ M =\ M1 \rangle\ M\ confl\ undef\ smaller$ 
        unfolding no-smaller-confl-def by auto
    }
    moreover {
      assume Da: Da = D + {#L#}
      have  $\neg M \models_{as} CNot\ Da$ 
      proof (rule ccontr)
        assume  $\neg ?thesis$ 
        then have  $-L \in lits\ of\ M$  unfolding Da by auto
        then have  $-L \in lits\ of\ (Propagated\ L\ ((D + \{ \#L\# \})) \# M1)$ 
          using UnI2  $\langle tl\ M' @\ Marked\ K'\ ia\ \# \ M =\ M1 \rangle$ 
          by auto
        moreover
          have backtrack S
            (cons-trail (Propagated L (D + {#L#}))
              (reduce-trail-to M1 (add-learned-cls (D + {#L#}))
                (update-backtrack-lvl i (update-conflicting None 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 None 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 $M': M'' @ \text{Marked } K i \# M' = \text{trail } S'$
and $D \in \# \text{ clauses } S'$
obtain $M N U k C L$ **where**
 S : *state* $S = (M, N, U, k, \text{None})$ **and**
 S' : *state* $S' = (\text{Propagated } L ((C + \{\#L\# \})) \# M, N, U, k, \text{None})$ **and**
 $C + \{\#L\# \} \in \# \text{ clauses } S$ **and**
 $M \models_{\text{as}} C \text{Not } C$ **and**
undefined-lit $M L$
using *propagate* **by** *auto*
have $\text{tl } M'' @ \text{Marked } K i \# M' = \text{trail } S$ **using** $M' S S'$
by (*metis Pair-inject list.inject list.sel(3) marked-lit.distinct(1) self-append-conv2*
tl-append2)
then have $\neg M' \models_{\text{as}} C \text{Not } D$
using $\langle D \in \# \text{ clauses } S' \rangle$ *n-l* $S S'$ *clauses-def* **unfolding** *no-smaller-conflict-def* **by** *auto*
then show $\neg M' \models_{\text{as}} C \text{Not } D$ **by** *auto*
qed

lemma *cdcl_W-cp-no-smaller-conflict-inv*:
assumes *propagate*: *cdcl_W-cp* $S S'$
and *n-l*: *no-smaller-conflict* S
shows *no-smaller-conflict* S'
using *assms*
proof (*induct rule: cdcl_W-cp.induct*)
case (*conflict'* $S S'$)
then show ?*case* **using** *conflict-no-smaller-conflict-inv*[*of* $S S'$] **by** *blast*
next
case (*propagate'* $S S'$)
then show ?*case* **using** *propagate-no-smaller-conflict-inv*[*of* $S S'$] **by** *fastforce*
qed

lemma *rtrancp-cdcl_W-cp-no-smaller-conflict-inv*:
assumes *propagate*: *cdcl_W-cp*^{**} $S S'$
and *n-l*: *no-smaller-conflict* S
shows *no-smaller-conflict* S'
using *assms*
proof (*induct rule: rtrancp-induct*)
case *base*
then show ?*case* **by** *simp*
next
case (*step* $S' S''$)

then show ?case **using** *cdcl_W-cp-no-smaller-conflict-inv*[of *S' S''*] **by** *fast*
qed

lemma *trancp-cdcl_W-cp-no-smaller-conflict-inv*:

assumes *propagate*: *cdcl_W-cp⁺⁺ S S'*

and *n-l*: *no-smaller-conflict S*

shows *no-smaller-conflict S'*

using *assms*

proof (*induct rule*: *trancp.induct*)

case (*r-into-tranc1 S S'*)

then show ?case **using** *cdcl_W-cp-no-smaller-conflict-inv*[of *S S'*] **by** *blast*

next

case (*tranc1-into-tranc1 S S' S''*)

then show ?case **using** *cdcl_W-cp-no-smaller-conflict-inv*[of *S' S''*] **by** *fast*

qed

lemma *full-cdcl_W-cp-no-smaller-conflict-inv*:

assumes *full cdcl_W-cp S S'*

and *n-l*: *no-smaller-conflict S*

shows *no-smaller-conflict S'*

using *assms* **unfolding** *full-def*

using *rtrancp-cdcl_W-cp-no-smaller-conflict-inv*[of *S S'*] **by** *blast*

lemma *full1-cdcl_W-cp-no-smaller-conflict-inv*:

assumes *full1 cdcl_W-cp S S'*

and *n-l*: *no-smaller-conflict S*

shows *no-smaller-conflict S'*

using *assms* **unfolding** *full1-def*

using *trancp-cdcl_W-cp-no-smaller-conflict-inv*[of *S S'*] **by** *blast*

lemma *cdcl_W-stgy-no-smaller-conflict-inv*:

assumes *cdcl_W-stgy S S'*

and *n-l*: *no-smaller-conflict S*

and *conflict-is-false-with-level S*

and *cdcl_W-M-level-inv S*

shows *no-smaller-conflict S'*

using *assms*

proof (*induct rule*: *cdcl_W-stgy.induct*)

case (*conflict' S'*)

then show ?case **using** *full1-cdcl_W-cp-no-smaller-conflict-inv*[of *S S'*] **by** *blast*

next

case (*other' S' S''*)

have *no-smaller-conflict S'*

using *cdcl_W-o-no-smaller-conflict-inv*[OF *other'.hyps(1) other'.prems(3,2,1)*]

not-conflict-not-any-negated-init-clss other'.hyps(2) **by** *blast*

then show ?case **using** *full-cdcl_W-cp-no-smaller-conflict-inv*[of *S' S''*] *other'.hyps* **by** *blast*

qed

lemma *conflict-conflict-is-no-clause-is-false-test*:

assumes *conflict S S'*

and ($\forall D \in \# \text{init-clss } S + \text{learned-clss } S. \text{trail } S \models_{\text{as}} \text{CNot } D$

$\rightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S) L = \text{backtrack-lvl } S))$

shows $\forall D \in \# \text{init-clss } S' + \text{learned-clss } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$

$\rightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{backtrack-lvl } S')$

using *assms* by *auto*

lemma *is-conflicting-exists-conflict*:

assumes $\neg(\forall D \in \# \text{init-clss } S' + \text{learned-clss } S'. \neg \text{trail } S' \models_{\text{as}} \text{CNot } D)$
and *conflicting* $S' = \text{None}$
shows $\exists S''. \text{conflict } S' S''$
using *assms* *clauses-def* *not-conflict-not-any-negated-init-clss* **by** *fastforce*

lemma *cdcl_W-o-conflict-is-no-clause-is-false*:

fixes $S S' :: 'st$
assumes
cdcl_W-o $S S'$ **and**
lev: *cdcl_W-M-level-inv* S **and**
max-lev: *conflict-is-false-with-level* S **and**
no-f: *no-clause-is-false* S **and**
no-l: *no-smaller-confl* S
shows *no-clause-is-false* S'
 $\vee (\text{conflicting } S' = \text{None}$
 $\longrightarrow (\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{backtrack-lvl } S')))$

using *assms*(1,2)

proof (*induct* rule: *cdcl_W-o-induct-lev2*)

case (*decide* $L T$) **note** $S = \text{this}(1)$ **and** *undef* = *this*(2) **and** $T = \text{this}(4)$

show ?*case*

proof (rule *HOL.disjI2*, *clarify*)

fix D

assume $D: D \in \# \text{clauses } T$ **and** $M-D: \text{trail } T \models_{\text{as}} \text{CNot } D$

let ? $M = \text{trail } S$

let ? $M' = \text{trail } T$

let ? $k = \text{backtrack-lvl } S$

have $\neg ?M \models_{\text{as}} \text{CNot } D$

using *no-f* $D S T \text{undef}$ **by** *auto*

have $-L \in \# D$

proof (rule *ccontr*)

assume $\neg ?thesis$

have ? $M \models_{\text{as}} \text{CNot } D$

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 $M-D x T \text{undef}$ **unfolding** *true-annots-def* *Ball-def* *true-annot-def* *CNot-def* *true-cls-def* *Bex-mset-def* **by** *auto*

show $\exists L \in \# x. \text{lits-of } ?M \models_l L$ **unfolding** *Bex-mset-def*

by (*metis* $\langle - L \notin \# D \rangle \langle L'' \in \# x \rangle L' \langle \text{lits-of } (\text{Marked } L (\text{?k} + 1) \# ?M) \models_l L' \rangle$
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** $\langle \neg ?M \models_{\text{as}} \text{CNot } D \rangle$ **by** *auto*

qed

have *atm-of* $L \notin \text{atm-of } \langle \text{lits-of } ?M \rangle$

using *undef* *defined-lit-map* **unfolding** *lits-of-def* **by** *fastforce*

then have *get-level* (*Marked* L ($?k + 1$) $\#$? M) $(-L) = ?k + 1$ **by** *simp*

```

    then show  $\exists La. La \in \# D \wedge \text{get-level } ?M' La = \text{backtrack-lvl } T$ 
      using  $\langle \neg L \in \# D \rangle T \text{ undef by auto}$ 
  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  $\text{decomp} = \text{this}(1)$  and  $\text{undef} = \text{this}(6)$  and  $T = \text{this}(7)$ 
  show  $?case$ 
  proof (rule HOL.disjI2, clarify)
    fix  $Da$ 
    assume  $Da: Da \in \# \text{ clauses } T$ 
    and  $M-D: \text{trail } T \models_{as} CNot \text{ } Da$ 
    obtain  $c$  where  $M: \text{trail } S = c @ M2 @ \text{Marked } K (i + 1) \# M1$ 
      using  $\text{decomp}$  by auto
    have  $tr-T: \text{trail } T = \text{Propagated } L (D + \{\#L\}) \# M1$ 
      using  $T \text{ decomp undef lev by (auto simp: cdcl}_W\text{-M-level-inv-decomp)}$ 
    have  $\text{backtrack } S T$ 
      using  $\text{backtrack.intros backtrack.hyps } T \text{ by (force simp del: state-simp simp: state-eq-def)}$ 
    then have  $\text{lev': cdcl}_W\text{-M-level-inv } T$ 
      using  $\text{cdcl}_W\text{-consistent-inv lev other by blast}$ 
    then have  $\neg L \notin \text{lits-of } M1$ 
      unfolding  $\text{cdcl}_W\text{-M-level-inv-def lits-of-def}$ 
      proof –
        have  $\text{consistent-interp } (\text{lits-of } (\text{trail } S)) \wedge \text{no-dup } (\text{trail } S)$ 
           $\wedge \text{backtrack-lvl } S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ 
           $\wedge \text{get-all-levels-of-marked } (\text{trail } S)$ 
           $= \text{rev } [1..<1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$ 
        using  $\text{lev cdcl}_W\text{-M-level-inv-def by blast}$ 
        then show  $\neg L \notin \text{lit-of ' set } M1$ 
          by (metis (no-types) One-nat-def add.right-neutral add-Suc-right
            atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set backtrack.hyps(2)
            cdcl}_W\text{.backtrack-lit-skipped cdcl}_W\text{-axioms decomp lits-of-def})
      qed
    { assume  $Da \in \# \text{ clauses } S$ 
      then have  $\neg M1 \models_{as} CNot \text{ } Da$  using no-l M unfolding no-smaller-confl-def by auto
    }
    moreover {
      assume  $Da: Da = D + \{\#L\}$ 
      have  $\neg M1 \models_{as} CNot \text{ } Da$  using  $\langle \neg L \notin \text{lits-of } M1 \rangle$  unfolding  $Da$  by simp
    }
    ultimately have  $\neg M1 \models_{as} CNot \text{ } Da$ 
      using  $Da T \text{ undef decomp lev by (fastforce simp: cdcl}_W\text{-M-level-inv-decomp)}$ 
    then have  $\neg L \in \# Da$ 
      using  $M-D \langle \neg L \notin \text{lits-of } M1 \rangle \text{ in-CNot-implies-uminus}(2)$ 
      true-annots-CNot-lit-of-notin-skip T unfolding tr-T
      by (smt insert-iff lits-of-cons marked-lit.sel(2))
    have  $g-M1: \text{get-all-levels-of-marked } M1 = \text{rev } [1..<i+1]$ 
      using  $\text{lev lev' } T \text{ decomp undef unfolding cdcl}_W\text{-M-level-inv-def}$  by auto
    have  $\text{no-dup } (\text{Propagated } L (D + \{\#L\}) \# M1)$ 
      using  $\text{lev lev' } T \text{ decomp undef unfolding cdcl}_W\text{-M-level-inv-def}$  by auto
    then have  $L: \text{atm-of } L \notin \text{atm-of ' lits-of } M1$  unfolding lits-of-def by auto
  
```

```

have get-level (Propagated L ((D + {#L#})) # M1) (-L) = 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  $\exists La. La \in \# Da \wedge \text{get-level } (\text{trail } T) La = \text{backtrack-lvl } T$ 
  using  $\langle -L \in \# Da \rangle T \text{ decomp undef lev}$  by (auto simp: cdclW-M-level-inv-def)
qed
qed

lemma full1-cdclW-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-confl: conflicting S = None
  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-confl)
then show ?thesis
  proof cases
    case confl
    then show ?thesis by blast
  next
    case propa
    then have conflicting U = None
      using no-confl by induction auto
    moreover have [simp]: learned-clss U = learned-clss S and
      [simp]: init-clss U = init-clss S
      using propa by induction auto
    moreover
      obtain D where D:  $D \in \# \text{clauses } U$  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 rtranclp-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  $\langle \text{conflicting } U = None \rangle D \text{ clauses-def not-conflict-not-any-negated-init-clss tr-U}$ 
        by blast
      then have False using full cdclW-cp.conflict' unfolding full-def by blast
      then show ?thesis by fast
  qed
qed

lemma full1-cdclW-cp-exists-conflict-full1-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-confl: conflicting S = None
  shows  $\exists T D. \text{propagate}^{**} S T \wedge \text{conflict } T U$ 
   $\wedge \text{trail } T \models_{as} CNot D \wedge \text{conflicting } U = \text{Some } D \wedge D \in \# \text{clauses } S$ 
proof -
  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 conf by induction auto
obtain D where trail T  $\models_{as}$  CNot D  $\wedge$  conflicting U = Some D  $\wedge$  D  $\in \#$  clauses S
  using conf p c by (fastforce simp: clauses-def)
then show ?thesis
  using propa conf by blast
qed

```

```

lemma cdclW-stgy-no-smaller-confl:
  assumes cdclW-stgy S S'
  and n-l: no-smaller-confl S
  and conflict-is-false-with-level S
  and cdclW-M-level-inv S
  and no-clause-is-false S
  and distinct-cdclW-state S
  and cdclW-conflicting S
  shows no-smaller-confl S'
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict' S')
  show no-smaller-confl S'
    using conflict'.hyps conflict'.prems(1) full1-cdclW-cp-no-smaller-confl-inv by blast
next
  case (other' S' S'')
  have lev': cdclW-M-level-inv S'
    using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
  show no-smaller-confl S''
    using cdclW-stgy-no-smaller-confl-inv[OF cdclW-stgy.other'[OF other'.hyps(1-3)]]
    other'.prems(1-3) by blast
qed

```

```

lemma cdclW-stgy-ex-lit-of-max-level:
  assumes cdclW-stgy S S'
  and n-l: no-smaller-confl S
  and conflict-is-false-with-level S
  and cdclW-M-level-inv S
  and no-clause-is-false S
  and distinct-cdclW-state S
  and cdclW-conflicting S
  shows conflict-is-false-with-level S'
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict' S')
  have no-smaller-confl S'
    using conflict'.hyps conflict'.prems(1) full1-cdclW-cp-no-smaller-confl-inv by blast
  moreover have conflict-is-false-with-level S'
    using conflict'.hyps conflict'.prems(2-4)
    rtranclp-cdclW-co-conflict-ex-lit-of-max-level[of S S']
    unfolding full-def full1-def rtranclp-unfold by presburger
  then show ?case by blast
next
  case (other' S' S'')
  have lev': cdclW-M-level-inv S'

```



```

using cdclW-consistent-inv other other'.hyps(1) other'.prems(3) by blast
moreover
  have no-clause-is-false S'
     $\vee$  (conflicting S' = None  $\longrightarrow$  ( $\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{as} CNot D$ 
       $\longrightarrow$  ( $\exists L. L \in \# D \wedge \text{get-level}(\text{trail } S') L = \text{backtrack-lvl } S')$ ))
    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' = None
    then have conflict-is-false-with-level S' by auto
    moreover have full cdclW-cp S' S''
      by (metis (no-types) other'.hyps(3))
    ultimately have conflict-is-false-with-level S''
      using rtranclp-cdclW-co-conflict-ex-lit-of-max-level[of S' S''] lev' <no-clause-is-false S'
      by blast
  }
moreover
  {
    assume c: conflicting S'  $\neq$  None
    have conflicting S  $\neq$  None using other'.hyps(1) c
      by (induct rule: cdclW-o-induct) auto
    then have conflict-is-false-with-level S'
      using cdclW-o-conflict-is-false-with-level-inv[OF other'.hyps(1)]
      other'.prems(3,5,6,2) by blast
    moreover have cdclW-cp** S' S'' using other'.hyps(3) unfolding full-def by auto
    then have S' = S'' using c
      by (induct rule: rtranclp-induct)
      (fastforce intro: option.exhaust)+
    ultimately have conflict-is-false-with-level S'' by auto
  }
  ultimately have conflict-is-false-with-level S'' by blast
}
moreover {
  assume
    confl: conflicting S' = None and
    D-L:  $\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{as} CNot D$ 
       $\longrightarrow$  ( $\exists L. L \in \# D \wedge \text{get-level}(\text{trail } S') L = \text{backtrack-lvl } S')$ 
  { assume  $\forall D \in \# \text{clauses } S'. \neg \text{trail } S' \models_{as} CNot D$ 
    then have no-clause-is-false S' using confl by simp
    then have conflict-is-false-with-level S'' using calculation(3) by presburger
  }
moreover {
  assume  $\neg(\forall D \in \# \text{clauses } S'. \neg \text{trail } S' \models_{as} CNot D)$ 
  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'' = Some D
    using full1-cdclW-cp-exists-conflict-full1-decompose[OF - - confl]
    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 rtranclp-cdclW-cp-dropWhile-trail other'(3) unfolding full-def by meson
  }
}

```

```

have btS: backtrack-lvl S'' = backtrack-lvl S'
  using other'.hyps(3) unfolding full-def by (metis 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'.hyps(3))
then have nd: no-dup (trail S'')
  by (metis (no-types) cdclW-M-level-inv-decomp(2))
have conflict-is-false-with-level S''
proof cases
  assume trail S' ⊨as CNot D
  moreover then obtain L where
    L ∈ # D and
    lev-L: get-level (trail S') L = backtrack-lvl S'
    using D-L D by blast
  moreover
    have LS': -L ∈ lits-of (trail S')
      using ⟨trail S' ⊨as CNot D⟩ ⟨L ∈ # D⟩ in-CNot-implies-uminus(2) by blast
    { fix x :: ('v, nat, 'v literal multiset) marked-lit and
      xb :: ('v, nat, 'v literal multiset) marked-lit
      assume a1: x ∈ set (trail S') and
        a2: xb ∈ set M and
        a3: (λl. atm-of (lit-of l)) ' set M ∩ (λl. atm-of (lit-of l)) ' set (trail S')
          = {} and
        a4: - L = lit-of x and
        a5: atm-of L = atm-of (lit-of xb)
      moreover have atm-of (lit-of x) = atm-of L
        using a4 by (metis (no-types) atm-of-uminus)
      ultimately have False
        using a5 a3 a2 a1 by auto
    }
  then have atm-of L ∉ atm-of ' lits-of M
    using nd LS' unfolding M by (auto simp add: lits-of-def)
  then have get-level (trail S'') L = get-level (trail S') L
    unfolding M by (simp add: lits-of-def)
  ultimately show ?thesis using btS ⟨conflicting S'' = Some D⟩ by auto
next
  assume ¬trail S' ⊨as CNot D
  then obtain L where L ∈ # D and LM: -L ∈ lits-of M
    using ⟨trail S'' ⊨as CNot D⟩
    by (auto simp add: CNot-def true-cls-def M true-annots-def true-annot-def
        split: split-if-asm)
  { fix x :: ('v, nat, 'v literal multiset) marked-lit and
    xb :: ('v, nat, 'v literal multiset) marked-lit
    assume a1: xb ∈ set (trail S') and
      a2: x ∈ set M and
      a3: atm-of L = atm-of (lit-of xb) and
      a4: - L = lit-of x and
      a5: (λl. atm-of (lit-of l)) ' set M ∩ (λl. atm-of (lit-of l)) ' set (trail S')
        = {}
    moreover have atm-of (lit-of xb) = atm-of (- L)
      using a3 by simp
    ultimately have False
      by auto
  }
  then have LS': atm-of L ∉ atm-of ' lits-of (trail S')
    using nd ⟨L ∈ # D⟩ LM unfolding M by (auto simp add: lits-of-def)

```

```

show ?thesis
proof cases
  assume ne: get-all-levels-of-marked (trail S') = []
  have backtrack-lvl S'' = 0
    using inv ne nm unfolding cdclW-M-level-inv-def M
    by (simp add: get-all-levels-of-marked-nil-iff-not-is-marked)
  moreover
    have a1: get-level M L = 0
      using nm by auto
    then have get-level (M @ trail S') L = 0
      by (metis LS' get-all-levels-of-marked-nil-iff-not-is-marked
        get-level-skip-beginning-not-marked lits-of-def ne)
    ultimately show ?thesis using ⟨conflicting S'' = Some D⟩ ⟨L ∈ # D⟩ unfolding M
      by auto
  next
    assume ne: get-all-levels-of-marked (trail S') ≠ []
    have hd (get-all-levels-of-marked (trail S')) = backtrack-lvl S'
      using ne lev' M nm unfolding cdclW-M-level-inv-def
      by (cases get-all-levels-of-marked (trail S'))
        (simp-all add: get-all-levels-of-marked-nil-iff-not-is-marked[symmetric])
    moreover have atm-of L ∈ atm-of ' lits-of M
      using ⟨¬L ∈ lits-of M⟩
      by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def)
    ultimately show ?thesis
      using nm ne ⟨L ∈ # D⟩ ⟨conflicting S'' = Some D⟩
        get-level-skip-beginning-hd-get-all-levels-of-marked[OF LS', of M]
        get-level-skip-in-all-not-marked[of rev M L backtrack-lvl S']
      unfolding lits-of-def btS M
      by auto
  qed
qed
}
ultimately have conflict-is-false-with-level S'' by blast
}
moreover
{
  assume conflicting S' ≠ None
  have no-clause-is-false S' using ⟨conflicting S' ≠ None⟩ by auto
  then have conflict-is-false-with-level S'' using calculation(3) by presburger
}
ultimately show ?case by fast
qed

```

lemma rtrancp-cdcl_W-stgy-no-smaller-confl-inv:

assumes

cdcl_W-stgy** S S' **and**

n-l: no-smaller-confl S **and**

cls-false: conflict-is-false-with-level S **and**

lev: cdcl_W-M-level-inv S **and**

no-f: no-clause-is-false S **and**

dist: distinct-cdcl_W-state S **and**

conflicting: cdcl_W-conflicting S **and**

decomp: all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S)) **and**

learned: cdcl_W-learned-clause S **and**

alien: no-strange-atm S

shows *no-smaller-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** $st = \text{this}(1)$ **and** $cdcl = \text{this}(2)$ **and** $IH = \text{this}(3)$
have *no-smaller-confl* S' **and** *conflict-is-false-with-level* S'
using IH **by** *blast+*
moreover have *cdcl_W-M-level-inv* S'
using st *lev rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*
by (*blast intro: rtranclp-cdcl_W-consistent-inv*)+
moreover have *no-clause-is-false* S'
using st *no-f rtranclp-cdcl_W-stgy-not-non-negated-init-clss* **by** *presburger*
moreover have *distinct-cdcl_W-state* S'
using *rtanclp-distinct-cdcl_W-state-inv*[*of* $S S'$] *lev rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*[*OF* st]
dist **by** *auto*
moreover have *cdcl_W-conflicting* S'
using *rtranclp-cdcl_W-all-inv*(6)[*of* $S S'$] st *alien conflicting decomp dist learned lev*
rtranclp-cdcl_W-stgy-rtranclp-cdcl_W **by** *blast*
ultimately show ?*case*
using *cdcl_W-stgy-no-smaller-confl*[*OF* $cdcl$] *cdcl_W-stgy-ex-lit-of-max-level*[*OF* $cdcl$] **by** *fast*
qed

17.6.7 Final States are Conclusive

lemma *full-cdcl_W-stgy-final-state-conclusive-non-false*:
fixes $S' :: 'st$
assumes *full: full cdcl_W-stgy (init-state N) S'*
and *no-d: distinct-mset-mset N*
and *no-empty: $\forall D \in \#N. D \neq \{\#\}$*
shows (*conflicting* $S' = \text{Some } \{\#\} \wedge$ *unsatisfiable (set-mset (init-clss S'))*)
 \vee (*conflicting* $S' = \text{None} \wedge$ *trail S' \models_{asm} init-clss S'*)
proof –
let ? $S = \text{init-state } N$
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: cdcl_W-learned-clause S' and
level-inv: cdcl_W-M-level-inv S' and
alien: no-strange-atm S' and
no-dup: distinct-cdcl_W-state S' and
confl: cdcl_W-conflicting S' and
decomp: all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))
using *no-d tranclp-cdcl_W-stgy-tranclp-cdcl_W*[*of* ? $S S'$] *step rtranclp-cdcl_W-all-inv*(1–6)[*of* ? $S S'$]
unfolding *rtranclp-unfold* **by** *auto*
moreover
have $\forall D \in \#N. \neg \square \models_{as} CNot D$ **using** *no-empty* **by** *auto*
then have *confl-k: conflict-is-false-with-level S'*
using *rtranclp-cdcl_W-stgy-no-smaller-confl-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 confl*
confl-k] .
qed

lemma *conflict-is-full1-cdcl_W-cp*:
 assumes *cp*: *conflict S S'*
 shows *full1 cdcl_W-cp S S'*
proof –
 have *cdcl_W-cp S S'* and *conflicting S' ≠ None* using *cp cdcl_W-cp.intros* by *auto*
 then have *cdcl_W-cp⁺⁺ S S'* by *blast*
 moreover have *no-step cdcl_W-cp S'*
 using *⟨conflicting S' ≠ None⟩* by (*metis cdcl_W-cp-conflicting-not-empty option.exhaust*)
 ultimately show *full1 cdcl_W-cp S S'* unfolding *full1-def* by *blast+*
qed

lemma *cdcl_W-cp-fst-empty-conflicting-false*:
 assumes *cdcl_W-cp S S'*
 and *trail S = []*
 and *conflicting S ≠ None*
 shows *False*
 using *assms* by (*induct rule: cdcl_W-cp.induct*) *auto*

lemma *cdcl_W-o-fst-empty-conflicting-false*:
 assumes *cdcl_W-o S S'*
 and *trail S = []*
 and *conflicting S ≠ None*
 shows *False*
 using *assms* by (*induct rule: cdcl_W-o.induct*) *auto*

lemma *cdcl_W-stgy-fst-empty-conflicting-false*:
 assumes *cdcl_W-stgy S S'*
 and *trail S = []*
 and *conflicting S ≠ None*
 shows *False*
 using *assms* apply (*induct rule: cdcl_W-stgy.induct*)
 using *trancpD cdcl_W-cp-fst-empty-conflicting-false* unfolding *full1-def* apply *metis*
 using *cdcl_W-o-fst-empty-conflicting-false* by *blast*
thm *cdcl_W-cp.induct[split-format(complete)]*

lemma *cdcl_W-cp-conflicting-is-false*:
cdcl_W-cp S S' ⟹ conflicting S = Some {#} ⟹ False
 by (*induction rule: cdcl_W-cp.induct*) *auto*

lemma *rtrancp-cdcl_W-cp-conflicting-is-false*:
cdcl_W-cp⁺⁺ S S' ⟹ conflicting S = Some {#} ⟹ False
 apply (*induction rule: trancp.induct*)
 by (*auto dest: cdcl_W-cp-conflicting-is-false*)

lemma *cdcl_W-o-conflicting-is-false*:
cdcl_W-o S S' ⟹ conflicting S = Some {#} ⟹ False
 by (*induction rule: cdcl_W-o.induct*) *auto*

lemma *cdcl_W-stgy-conflicting-is-false*:
cdcl_W-stgy S S' ⟹ conflicting S = Some {#} ⟹ False
 apply (*induction rule: cdcl_W-stgy.induct*)
 unfolding *full1-def* apply (*metis (no-types) cdcl_W-cp-conflicting-not-empty trancpD*)

unfolding *full-def* **by** (*metis conflict-with-false-implies-terminated other*)

lemma *rtrancpl-cdcl_W-stgy-conflicting-is-false*:
*cdcl_W-stgy^{**} S S' \implies conflicting S = Some {#} \implies S' = S*
apply (*induction rule: rtrancpl-induct*)
apply *simp*
using *cdcl_W-stgy-conflicting-is-false* **by** *blast*

lemma *full-cdcl_W-init-clss-with-false-normal-form*:
assumes
 $\forall m \in \text{set } M. \neg \text{is-marked } m$ **and**
 $E = \text{Some } D$ **and**
 $\text{state } S = (M, N, U, 0, E)$
full *cdcl_W-stgy S S'* **and**
all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))
cdcl_W-learned-clause S
cdcl_W-M-level-inv S
no-strange-atm S
distinct-cdcl_W-state S
cdcl_W-conflicting S
shows $\exists M''. \text{state } S' = (M'', N, U, 0, \text{Some } \{ \# \})$
using *assms(10,9,8,7,6,5,4,3,2,1)*
proof (*induction M arbitrary: E D S*)
case *Nil*
then show *?case*
using *rtrancpl-cdcl_W-stgy-conflicting-is-false* **unfolding** *full-def cdcl_W-conflicting-def* **by** *auto*
next
case (*Cons L M*) **note** $IH = \text{this}(1)$ **and** $\text{full} = \text{this}(8)$ **and** $E = \text{this}(10)$ **and** $\text{inv} = \text{this}(2-7)$ **and**
 $S = \text{this}(9)$ **and** $\text{nm} = \text{this}(11)$
obtain $K \ p$ **where** $K: L = \text{Propagated } K \ p$
using nm **by** (*cases L*) *auto*
have *every-mark-is-a-conflict S* **using** inv **unfolding** *cdcl_W-conflicting-def* **by** *auto*
then have $MpK: M \models_{\text{as}} \text{CNot } (p - \{ \#K \# \})$ **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 rtrancpl-cdcl_W-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 cdcl_W-cp S* **by** (*auto simp: cdcl_W-cp.simps*)
have *res-skip: $\exists T. (\text{resolve } S \ T \wedge \text{no-step skip } S \wedge \text{full cdcl_W-cp } T \ T)$*
 $\vee (\text{skip } S \ T \wedge \text{no-step resolve } S \wedge \text{full cdcl_W-cp } T \ T)$
proof *cases*
assume $\neg \text{lit-of } L \notin \# \ D$
then obtain T **where** $\text{sk: skip } S \ T$ **and** $\text{res: no-step resolve } S$

```

using  $S$  that  $D' K$  unfolding  $skip.simps E$  by  $fastforce$ 
have full  $cdcl_W\text{-cp } T T$ 
  using  $sk$  by ( $auto simp add: option\text{-full}\text{-}cdcl_W\text{-cp}$ )
then show  $?thesis$ 
  using  $sk res$  by  $blast$ 
next
assume  $LD: \neg \neg \text{lit-of } L \notin \# D$ 
then have  $D: \text{Some } D = \text{Some } ((D - \{\# \neg \text{lit-of } L \# \}) + \{\# \neg \text{lit-of } L \# \})$ 
  by ( $auto simp add: multiset\text{-eq}\text{-iff}$ )

have  $\bigwedge L. \text{get-level } M L = 0$ 
  by ( $simp add: nm$ )
then have  $\text{get-maximum-level } (Propagated K (p - \{\# K \# \} + \{\# K \# \}) \# M) (D - \{\# \neg$ 
 $K \# \}) = 0$ 
  using  $LD \text{get-maximum-level-exists-lit-of-max-level}$ 
  proof -
    obtain  $L'$  where  $\text{get-level } (L \# M) L' = \text{get-maximum-level } (L \# M) D$ 
    using  $LD \text{get-maximum-level-exists-lit-of-max-level[of } D L \# M]$  by  $fastforce$ 
    then show  $?thesis$  by ( $metis (mono-tags) K' \text{bex-msetE get-level-skip-all-not-marked}$ 
       $\text{get-maximum-level-exists-lit } nm \text{not-gr0}$ )
  qed
then obtain  $T$  where  $sk: \text{resolve } S T$  and  $res: \text{no-step skip } S$ 
  using  $\text{resolve-rule[of } S K p - \{\# K \# \} M N U 0 (D - \{\# \neg K \# \})$ 
   $\text{update-conflicting } (\text{Some } (\text{remdups-mset } (D - \{\# \neg K \# \} + (p - \{\# K \# \})))) (tl\text{-trail } S)]$ 
   $S$  unfolding  $K' D E$  by  $fastforce$ 
have full  $cdcl_W\text{-cp } T T$ 
  using  $sk$  by ( $auto simp add: option\text{-full}\text{-}cdcl_W\text{-cp}$ )
then show  $?thesis$ 
  using  $sk res$  by  $blast$ 
qed
then have  $step\text{-s}: \exists T. cdcl_W\text{-stgy } S T$ 
  using ( $\text{no-step } cdcl_W\text{-cp } S$ )  $other'$  by ( $meson \text{bj resolve skip}$ )
have  $\text{get-all-marked-decomposition } (L \# M) = [([], L \# M)]$ 
  using  $nm$  unfolding  $K$  apply ( $\text{induction } M \text{ rule: marked-lit-list-induct, simp}$ )
  by ( $\text{rename-tac } L l \text{xs, case-tac hd (get-all-marked-decomposition xs), auto}$ ) +
then have  $\text{no-b: no-step backtrack } S$ 
  using  $nm S$  by  $auto$ 
have  $\text{no-d: no-step decide } S$ 
  using  $S E$  by  $auto$ 

have full  $S\text{-S: full } cdcl_W\text{-cp } S S$ 
  using  $S E$  by ( $auto simp add: option\text{-full}\text{-}cdcl_W\text{-cp}$ )
then have  $\text{no-f: no-step (full1 } cdcl_W\text{-cp) } S$ 
  unfolding  $\text{full-def full1-def rtranclp-unfold}$  by ( $meson \text{trancplD}$ )
obtain  $T$  where
   $s: cdcl_W\text{-stgy } S T$  and  $st: cdcl_W\text{-stgy}^{**} T S'$ 
  using full  $step\text{-s}$  full unfolding  $\text{full-def}$  by ( $metis \text{rtranclp-unfold trancplD}$ )
have  $\text{resolve } S T \vee \text{skip } S T$ 
  using  $s \text{no-b no-d res-skip full-S-S}$  unfolding  $cdcl_W\text{-stgy.simps } cdcl_W\text{-o.simps full-unfold}$ 
   $\text{full1-def}$ 
  by ( $auto dest!: \text{trancplD simp: } cdcl_W\text{-bj.simps}$ )
then obtain  $D'$  where  $T: \text{state } T = (M, N, U, 0, \text{Some } D')$ 
  using  $S E$  by  $auto$ 

have  $st\text{-c: } cdcl_W^{**} S T$ 

```

```

    using E T rtrancpl-cdclW-stgy-rtrancpl-cdclW s by blast
have cdclW-conflicting T
    using rtrancpl-cdclW-all-inv(6)[OF st-c inv(6,5,4,3,2,1)] .
show ?thesis
    apply (rule IH[of T])
        using rtrancpl-cdclW-all-inv(6)[OF st-c inv(6,5,4,3,2,1)] apply blast
        using rtrancpl-cdclW-all-inv(5)[OF st-c inv(6,5,4,3,2,1)] apply blast
        using rtrancpl-cdclW-all-inv(4)[OF st-c inv(6,5,4,3,2,1)] apply blast
        using rtrancpl-cdclW-all-inv(3)[OF st-c inv(6,5,4,3,2,1)] apply blast
        using rtrancpl-cdclW-all-inv(2)[OF st-c inv(6,5,4,3,2,1)] apply blast
        using rtrancpl-cdclW-all-inv(1)[OF st-c inv(6,5,4,3,2,1)] apply blast
    apply (metis full-def st full)
    using T E apply blast
    apply auto[]
    using nm by simp
qed
qed

lemma full-cdclW-stgy-final-state-conclusive-is-one-false:
  fixes S' :: 'st
  assumes full: full cdclW-stgy (init-state N) S'
  and no-d: distinct-mset-mset N
  and empty: {#} ∈ # N
  shows conflicting S' = Some {#} ∧ unsatisfiable (set-mset (init-clss S'))
proof -
  let ?S = init-state N
  have cdclW-stgy** ?S S' and no-step cdclW-stgy S' using full unfolding full-def by auto
  then have plus-or-eq: cdclW-stgy++ ?S S' ∨ S' = ?S unfolding rtrancpl-unfold by auto
  have ∃ S''. conflict ?S S'' using empty not-conflict-not-any-negated-init-clss by force

  then have cdclW-stgy: ∃ S'. cdclW-stgy ?S S'
    using cdclW-cp.conflict'[of ?S] conflict-is-full1-cdclW-cp cdclW-stgy.intros(1) by metis
  have S' ≠ ?S using (no-step cdclW-stgy S') cdclW-stgy by blast

  then obtain St:: 'st where St: cdclW-stgy ?S St and cdclW-stgy** St S'
    using plus-or-eq by (metis (no-types) (cdclW-stgy** ?S S') converse-rtrancplE)
  have st: cdclW** ?S St
    by (simp add: rtrancpl-unfold (cdclW-stgy ?S St) cdclW-stgy-trancpl-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 rtrancpl-cdclW-cp-backtrack-lvl unfolding full1-def
    by (fastforce dest!: trancpl-into-rtrancpl)
  have cls-St: init-clss St = N
    using fullSt cdclW-stgy-no-more-init-clss[OF St] by auto
  have conflicting St ≠ None
  proof (rule ccontr)
    assume ¬ ?thesis
    then have ∃ T. conflict St T
      using empty cls-St[] conflict-rule[of St trail St N learned-clss St backtrack-lvl St {#}]
      by (auto simp: clauses-def)

```



```

    then show False using fullSt unfolding full1-def by blast
qed

have 1:  $\forall m \in \text{set } (\text{trail } St). \neg \text{is-marked } m$ 
  using fullSt unfolding full1-def by (auto dest!: trancplp-into-rtrancplp
    rtrancplp-cdclW-cp-dropWhile-trail)
have 2: full cdclW-stgy St S'
  using  $\langle \text{cdcl}_W\text{-stgy}^{**} St S' \rangle \langle \text{no-step cdcl}_W\text{-stgy } S' \rangle$  bt unfolding full-def by auto
have 3: all-decomposition-implies-m
  (init-clss St)
  (get-all-marked-decomposition
    (trail St))
  using rtrancplp-cdclW-all-inv(1)[OF st] no-d bt by simp
have 4: cdclW-learned-clause St
  using rtrancplp-cdclW-all-inv(2)[OF st] no-d bt by simp
have 5: cdclW-M-level-inv St
  using rtrancplp-cdclW-all-inv(3)[OF st] no-d bt by simp
have 6: no-strange-atm St
  using rtrancplp-cdclW-all-inv(4)[OF st] no-d bt by simp
have 7: distinct-cdclW-state St
  using rtrancplp-cdclW-all-inv(5)[OF st] no-d bt by simp
have 8: cdclW-conflicting St
  using rtrancplp-cdclW-all-inv(6)[OF st] no-d bt by simp
have init-clss S' = init-clss St and conflicting S' = Some {#}
  using  $\langle \text{conflicting } St \neq \text{None} \rangle$  full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St]
  2 3 4 5 6 7 8 St apply (metis  $\langle \text{cdcl}_W\text{-stgy}^{**} St S' \rangle$  rtrancplp-cdclW-stgy-no-more-init-clss)
  using  $\langle \text{conflicting } St \neq \text{None} \rangle$  full-cdclW-init-clss-with-false-normal-form[OF 1, of - - St - -
    S] 2 3 4 5 6 7 8 by (metis bt option.exhaust prod.inject)

moreover have init-clss S' = N
  using  $\langle \text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S' \rangle$  rtrancplp-cdclW-stgy-no-more-init-clss by fastforce
moreover have unsatisfiable (set-mset N)
  by (meson empty mem-set-mset-iff satisfiable-def true-clss-empty true-clss-def)
ultimately show ?thesis by auto
qed

```

lemma *full-cdcl_W-stgy-final-state-conclusive:*

```

  fixes S' :: 'st
  assumes full: full cdclW-stgy (init-state N) S' and no-d: distinct-mset-mset N
  shows  $(\text{conflicting } S' = \text{Some } \{ \# \} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S')))$ 
     $\vee (\text{conflicting } S' = \text{None} \wedge \text{trail } S' \models_{\text{asm}} \text{init-clss } S')$ 
  using assms full-cdclW-stgy-final-state-conclusive-is-one-false
full-cdclW-stgy-final-state-conclusive-non-false by blast

```

lemma *full-cdcl_W-stgy-final-state-conclusive-from-init-state:*

```

  fixes S' :: 'st
  assumes full: full cdclW-stgy (init-state N) S'
  and no-d: distinct-mset-mset N
  shows  $(\text{conflicting } S' = \text{Some } \{ \# \} \wedge \text{unsatisfiable } (\text{set-mset } N))$ 
     $\vee (\text{conflicting } S' = \text{None} \wedge \text{trail } S' \models_{\text{asm}} N \wedge \text{satisfiable } (\text{set-mset } N))$ 

```

proof –

```

  have N: init-clss S' = N
    using full unfolding full-def by (auto dest: rtrancplp-cdclW-stgy-no-more-init-clss)
  consider

```

```

  (confl) conflicting  $S' = \text{Some } \{\#\}$  and unsatisfiable (set-mset (init-clss  $S'$ ))
| (sat) conflicting  $S' = \text{None}$  and trail  $S' \models_{asm} \text{init-clss } S'$ 
using full-cdclW-stgy-final-state-conclusive[OF assms] by auto
then show ?thesis
proof cases
  case confl
    then show ?thesis by (auto simp: N)
  next
    case sat
    have cdclW-M-level-inv (init-state N) by auto
    then have cdclW-M-level-inv  $S'$ 
      using full rtrancp-cdclW-stgy-consistent-inv unfolding full-def by blast
    then have consistent-interp (lits-of (trail  $S'$ )) unfolding cdclW-M-level-inv-def by blast
    moreover have lits-of (trail  $S'$ )  $\models_s$  set-mset (init-clss  $S'$ )
      using sat(2) by (auto simp add: true-annots-def true-annot-def true-clss-def)
    ultimately have satisfiable (set-mset (init-clss  $S'$ )) by simp
    then show ?thesis using sat unfolding N by blast
  qed
qed
end
end
theory CDCL-W-Termination
imports CDCL-W
begin

context cdclW
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 *simple-clss*.

The invariant contains all the structural invariants that holds,

definition cdcl_W-all-struct-inv **where**

```

cdclW-all-struct-inv  $S =$ 
  (no-strange-atm  $S \wedge$  cdclW-M-level-inv  $S$ 
 $\wedge (\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s)$ 
 $\wedge \text{distinct-cdcl}_W\text{-state } S \wedge \text{cdcl}_W\text{-conflicting } S$ 
 $\wedge \text{all-decomposition-implies-m (init-clss } S) (\text{get-all-marked-decomposition (trail } S))$ 
 $\wedge \text{cdcl}_W\text{-learned-clause } S)$ 

```

lemma cdcl_W-all-struct-inv-inv:

```

assumes cdclW  $S S'$  and cdclW-all-struct-inv  $S$ 
shows cdclW-all-struct-inv  $S'$ 
unfolding cdclW-all-struct-inv-def
proof (intro HOL.conjI)
  show no-strange-atm  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by auto
  show cdclW-M-level-inv  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast
  show distinct-cdclW-state  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast
  show cdclW-conflicting  $S'$ 
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast
  show all-decomposition-implies-m (init-clss  $S'$ ) (get-all-marked-decomposition (trail  $S'$ ))

```

```

    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast
  show cdclW-learned-clause S'
    using cdclW-all-inv[OF assms(1)] assms(2) unfolding cdclW-all-struct-inv-def by fast

  show  $\forall s \in \# \text{learned-clss } S'. \neg \text{tautology } s$ 
    using assms(1)[THEN learned-clss-are-not-tautologies] assms(2)
    unfolding cdclW-all-struct-inv-def by fast
qed

```

```

lemma rtranclp-cdclW-all-struct-inv-inv:
  assumes cdclW** S S' and cdclW-all-struct-inv S
  shows cdclW-all-struct-inv S'
  using assms by induction (auto intro: cdclW-all-struct-inv-inv)

```

```

lemma cdclW-stgy-cdclW-all-struct-inv:
  cdclW-stgy S T  $\implies$  cdclW-all-struct-inv S  $\implies$  cdclW-all-struct-inv T
  by (meson cdclW-stgy-rtranclp-cdclW rtranclp-cdclW-all-struct-inv-inv rtranclp-unfold)

```

```

lemma rtranclp-cdclW-stgy-cdclW-all-struct-inv:
  cdclW-stgy** S T  $\implies$  cdclW-all-struct-inv S  $\implies$  cdclW-all-struct-inv T
  by (induction rule: rtranclp-induct) (auto intro: cdclW-stgy-cdclW-all-struct-inv)

```

17.8 No Relearning of a clause

```

lemma cdclW-o-new-clause-learned-is-backtrack-step:
  assumes learned: D  $\in \#$  learned-clss T and
  new: D  $\notin \#$  learned-clss S and
  cdclW: cdclW-o S T and
  lev: cdclW-M-level-inv S
  shows backtrack S T  $\wedge$  conflicting S = Some D
  using cdclW lev learned new
proof (induction rule: cdclW-o-induct-lev2)
  case (backtrack K i M1 M2 L C T) note decomp = this(1) and undef = this(6) and T = this(7)
  and
    D-T = this(9) and D-S = this(10)
  then have D = C + {#L#}
    using not-gr0 lev by (auto simp: cdclW-M-level-inv-decomp)
  then show ?case
    using T backtrack.hyps(1-5) backtrack.intros by auto
qed auto

```

```

lemma cdclW-cp-new-clause-learned-has-backtrack-step:
  assumes learned: D  $\in \#$  learned-clss T and
  new: D  $\notin \#$  learned-clss S and
  cdclW: cdclW-stgy S T and
  lev: cdclW-M-level-inv S
  shows  $\exists S'. \text{backtrack } S S' \wedge \text{cdcl}_W\text{-stgy** } S' T \wedge \text{conflicting } S = \text{Some } D$ 
  using cdclW learned new
proof (induction rule: cdclW-stgy.induct)
  case (conflict' S')
  then show ?case
    unfolding full1-def by (metis (mono-tags, lifting) rtranclp-cdclW-cp-learned-clause-inv
      tranclp-into-rtranclp)
next
  case (other' S' S'')
  then have D  $\in \#$  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 ∈ # learned-clss T and*
new: D ∉ # learned-clss S and
*cdcl_W: cdcl_W-stgy** S T and*
lev: cdcl_W-M-level-inv S
shows $\exists S' S''. \text{cdcl}_W\text{-stgy}^{**} S S' \wedge \text{backtrack } S' S'' \wedge \text{conflicting } S' = \text{Some } D \wedge$
 $\text{cdcl}_W\text{-stgy}^{**} S'' T$
using *cdcl_W learned new*
proof (*induction rule: rtranclp-induct*)
case *base*
then show ?*case* **by** *blast*
next
case (*step T U*) **note** *st = this(1) and o = this(2) and IH = this(3) and*
D-U = this(4) and D-S = this(5)
show ?*case*
proof (*cases D ∈ # learned-clss T*)
case *True*
then obtain *S' S'' where*
*st': cdcl_W-stgy** S S' and*
bt: backtrack S' S'' and
confl: conflicting S' = Some D and
*st'': cdcl_W-stgy** S'' T*
using *IH D-S by metis*
then show ?*thesis* **using** *o* **by** (*meson rtranclp.simps*)
next
case *False*
have *cdcl_W-M-level-inv T*
using *lev rtranclp-cdcl_W-stgy-consistent-inv st* **by** *blast*
then obtain *S' where*
bt: backtrack T S' and
*st': cdcl_W-stgy** S' U and*
confl: conflicting T = Some D
using *cdcl_W-cp-new-clause-learned-has-backtrack-step*[*OF D-U False o*]
by *metis*
then have *cdcl_W-stgy** S T and*
backtrack T S' and
conflicting T = Some D and
*cdcl_W-stgy** S' U*
using *o st* **by** *auto*
then show ?*thesis* **by** *blast*
qed
qed

lemma *propagate-no-more-Marked-lit*:
assumes *propagate S S'*
shows *Marked K i ∈ set (trail S) ⟷ Marked K i ∈ set (trail S')*
using *assms* **by** *auto*

lemma *conflict-no-more-Marked-lit*:

assumes *conflict S S'*

shows $\text{Marked } K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$

using *assms* **by** *auto*

lemma *cdcl_W-cp-no-more-Marked-lit*:

assumes *cdcl_W-cp S S'*

shows $\text{Marked } K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$

using *assms* **apply** (*induct rule: cdcl_W-cp.induct*)

using *conflict-no-more-Marked-lit propagate-no-more-Marked-lit* **by** *auto*

lemma *rtrancpl-cdcl_W-cp-no-more-Marked-lit*:

assumes *cdcl_W-cp** S S'*

shows $\text{Marked } K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$

using *assms* **apply** (*induct rule: 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 $\text{Marked } K \ i \in \text{set } (\text{trail } S') \longrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S)$

using *assms*

proof (*induct rule: cdcl_W-o-induct-lev2*)

case *backtrack* **note** *decomp = this(1)* **and** *undef = this(6)* **and** *T = 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 = this(1)* **and** *no-dup = this(2)* **and** *S' = this(3)* **and** *S = this(4)*

have *cdcl_W-M-level-inv S'*

using *full1-cdcl_W-cp-consistent-inv no-dup st* **by** *blast*

then have $\text{Marked } L \ i \in \text{set } (\text{trail } S') \text{ and } \text{Marked } L \ i \notin \text{set } (\text{trail } S)$

using *no-dup unfolding S S' cdcl_W-M-level-inv-def* **by** (*auto simp add: rev-image-eqI*)

then have *False*

using *st rtrancpl-cdcl_W-cp-no-more-Marked-lit[of S S']*

unfolding *full1-def rtrancpl-unfold* **by** *blast*

then show *?case* **by** *fast*

next

case (*other' T U*) **note** *o = this(1)* **and** *ns = this(2)* **and** *st = this(3)* **and** *no-dup = this(4)* **and** *S' = this(5)* **and** *S = this(6)*

have *cdcl_W-M-level-inv U*

by (*metis (full-types) lev cdcl_W.simps cdcl_W-consistent-inv full-def o other'.hyps(3) rtrancpl-cdcl_W-cp-consistent-inv*)

then have $\text{Marked } L \ i \in \text{set } (\text{trail } U) \text{ and } \text{Marked } L \ i \notin \text{set } (\text{trail } S)$

using *no-dup unfolding S S' cdcl_W-M-level-inv-def* **by** (*auto simp add: rev-image-eqI*)

then have $\text{Marked } L \ i \in \text{set } (\text{trail } T)$
using $st \text{ rtrancpl-cdcl}_W\text{-cp-no-more-Marked-lit}$ **unfolding** full-def **by** blast
then show $?case$
using $\text{cdcl}_W\text{-o-no-more-Marked-lit}[OF \ o] \langle \text{Marked } L \ i \notin \text{set } (\text{trail } S) \rangle \text{ ns lev}$ **by** meson
qed

lemma $\text{cdcl}_W\text{-o-is-decide}$:

assumes $\text{cdcl}_W\text{-o } S' \ T$ **and** $\text{cdcl}_W\text{-M-level-inv } S'$
 $\text{trail } T = \text{drop } (\text{length } M_0) \ M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**
 $\neg (\exists M'. \text{trail } S' = M' @ \text{Marked } L \ i \ \# \ H @ M)$
shows $\text{decide } S' \ T$
using assms
proof ($\text{induction rule:cdcl}_W\text{-o-induct-lev2}$)
case ($\text{backtrack } K \ i \ M1 \ M2 \ L \ D$)
then obtain c **where** $\text{trail } S' = c @ M2 @ \text{Marked } K \ (\text{Suc } i) \ \# \ M1$
by auto
then show $?case$
using $\text{backtrack by } (\text{cases drop } (\text{length } M_0) \ M') \ (\text{auto simp: cdcl}_W\text{-M-level-inv-def})$
next
case decide
show $?case$ **using** $\text{decide-rule}[of \ S'] \ \text{decide}(1-4)$ **by** auto
qed auto

lemma $\text{rtrancpl-cdcl}_W\text{-new-marked-at-beginning-is-decide}$:

assumes $\text{cdcl}_W\text{-stgy}^{**} \ R \ U$ **and**
 $\text{trail } U = M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**
 $\text{trail } R = M$ **and**
 $\text{cdcl}_W\text{-M-level-inv } R$
shows
 $\exists S \ T \ T'. \text{cdcl}_W\text{-stgy}^{**} \ R \ S \wedge \text{decide } S \ T \wedge \text{cdcl}_W\text{-stgy}^{**} \ T \ U \wedge \text{cdcl}_W\text{-stgy}^{**} \ S \ U \wedge$
 $\text{no-step cdcl}_W\text{-cp } S \wedge \text{trail } T = \text{Marked } L \ i \ \# \ H @ M \wedge \text{trail } S = H @ M \wedge \text{cdcl}_W\text{-stgy } S \ T' \wedge$
 $\text{cdcl}_W\text{-stgy}^{**} \ T' \ U$
using assms
proof ($\text{induct arbitrary: } M \ H \ M' \ i \text{ rule: rtrancpl-induct}$)
case base
then show $?case$ **by** auto
next
case ($\text{step } T \ U$) **note** $st = \text{this}(1)$ **and** $IH = \text{this}(3)$ **and** $s = \text{this}(2)$ **and**
 $U = \text{this}(4)$ **and** $S = \text{this}(5)$ **and** $\text{lev} = \text{this}(6)$
show $?case$
proof ($\text{cases } \exists M'. \text{trail } T = M' @ \text{Marked } L \ i \ \# \ H @ M$)
case False
with s **show** $?thesis$ **using** $U \ s \ st \ S$
proof induction
case ($\text{conflict}' \ W$) **note** $cp = \text{this}(1)$ **and** $nd = \text{this}(2)$ **and** $W = \text{this}(3)$
then obtain M_0 **where** $\text{trail } W = M_0 @ \text{trail } T$ **and** $\text{nmarked: } \forall l \in \text{set } M_0. \neg \text{is-marked } l$
using $\text{rtrancpl-cdcl}_W\text{-cp-dropWhile-trail}$ **unfolding** $\text{full1-def rtrancpl-unfold}$ **by** meson
then have $MV: M' @ \text{Marked } L \ i \ \# \ H @ M = M_0 @ \text{trail } T$ **unfolding** W **by** simp
then have $V: \text{trail } T = \text{drop } (\text{length } M_0) \ (M' @ \text{Marked } L \ i \ \# \ H @ M)$
by auto
have $\text{takeWhile } (\text{Not } o \text{ is-marked}) \ M' = M_0 @ \text{takeWhile } (\text{Not } o \text{ is-marked}) \ (\text{trail } T)$
using $\text{arg-cong}[OF \ MV, \text{ of takeWhile } (\text{Not } o \text{ is-marked})]$ nmarked
by ($\text{simp add: takeWhile-tail}$)
from $\text{arg-cong}[OF \ \text{this, of length}]$ **have** $\text{length } M_0 \leq \text{length } M'$
unfolding length-append **by** ($\text{metis (no-types, lifting) Nat.le-trans le-add1}$)

```

    length-takeWhile-le)
  then have False using nd V by auto
  then show ?case by fast
next
case (other' T' U) note o = this(1) and ns = this(2) and cp = this(3) and nd = this(4)
  and U = this(5) and st = this(6)
obtain M0 where trail U = M0 @ trail T' and nmarked:  $\forall l \in \text{set } M_0. \neg \text{is-marked } l$ 
  using rtrancpl-cdclW-cp-dropWhile-trail cp unfolding full-def by meson
then have MV: M' @ Marked L i # H @ M = M0 @ trail T' unfolding U by simp
then have V: trail T' = drop (length M0) (M' @ Marked L i # H @ M)
  by auto
have takeWhile (Not o is-marked) M' = M0 @ takeWhile (Not o is-marked) (trail T')
  using arg-cong[OF MV, of takeWhile (Not o is-marked)] nmarked
  by (simp add: takeWhile-tail)
from arg-cong[OF this, of length] have length M0 ≤ length M'
  unfolding length-append by (metis (no-types, lifting) Nat.le-trans le-add1
    length-takeWhile-le)
then have tr-T': trail T' = drop (length M0) M' @ Marked L i # H @ M using V by auto
then have LT': Marked L i ∈ set (trail T') by auto
moreover
  have cdclW-M-level-inv T
    using lev rtrancpl-cdclW-stgy-consistent-inv step.hyps(1) by blast
  then have decide T T' using o nd tr-T' cdclW-o-is-decide by metis
ultimately have decide T T' using cdclW-o-no-more-Marked-lit[OF o] by blast
then have 1: cdclW-stgy** R T and 2: decide T T' and 3: cdclW-stgy** T' U
  using st other'.prems(4)
  by (metis cdclW-stgy.conflict' cp full-unfold r-into-rtrancpl rtrancpl.rtrancpl-refl)+
have [simp]: drop (length M0) M' = []
  using ⟨decide T T'⟩ ⟨Marked L i ∈ set (trail T')⟩ nd tr-T'
  by (auto simp add: Cons-eq-append-conv)
have T': drop (length M0) M' @ Marked L i # H @ M = Marked L i # trail T
  using ⟨decide T T'⟩ ⟨Marked L i ∈ set (trail T')⟩ nd tr-T'
  by auto
have trail T' = Marked L i # trail T
  using ⟨decide T T'⟩ ⟨Marked L i ∈ set (trail T')⟩ tr-T'
  by auto
then have 5: trail T' = Marked L i # H @ M
  using append.simps(1) list.sel(3) local.other'(5) tl-append2 by (simp add: tr-T')
have 6: trail T = H @ M
  by (metis (no-types) ⟨trail T' = Marked L i # trail T⟩
    ⟨trail T' = drop (length M0) M' @ Marked L i # H @ M⟩ append-Nil list.sel(3) nd
    tl-append2)
have 7: cdclW-stgy** T U using other'.prems(4) st by auto
have 8: cdclW-stgy T U cdclW-stgy** U U
  using cdclW-stgy.other'[OF other'(1-3)] by simp-all
show ?case apply (rule exI[of - T], rule exI[of - T'], rule exI[of - U])
  using ns 1 2 3 5 6 7 8 by fast
qed
next
case True
then obtain M' where T: trail T = M' @ Marked L i # H @ M by metis
from IH[OF this S lev] obtain S' S'' S''' where
  1: cdclW-stgy** R S' and
  2: decide S' S'' and
  3: cdclW-stgy** S'' T and

```

4: *no-step cdcl_W-cp S' and*
 6: *trail S'' = Marked L i # H @ M and*
 7: *trail S' = H @ M and*
 8: *cdcl_W-stgy** S' T and*
 9: *cdcl_W-stgy S' S''' and*
 10: *cdcl_W-stgy** S''' T*
 by *blast*
 have *cdcl_W-stgy** S'' U using s ⟨cdcl_W-stgy** S'' T⟩ by auto*
 moreover have *cdcl_W-stgy** S' U using 8 s by auto*
 moreover have *cdcl_W-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 *rtrancpl-cdcl_W-new-marked-at-beginning-is-decide'*:

assumes *cdcl_W-stgy** R U and*
trail U = M' @ Marked L i # H @ M and
trail R = M and
cdcl_W-M-level-inv R
 shows $\exists y y'. \text{cdcl}_W\text{-stgy}^{**} R y \wedge \text{cdcl}_W\text{-stgy } y y' \wedge \neg (\exists c. \text{trail } y = c @ \text{Marked } L i \# H @ M)$
 $\wedge (\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L i \# H @ M))^{**} y' U$
 proof –
 fix *T'*
 obtain *S' T T'* where
*st: cdcl_W-stgy** R S' and*
decide S' T and
*TU: cdcl_W-stgy** T U and*
no-step cdcl_W-cp S' and
trT: trail T = Marked L i # H @ M and
trS': trail S' = H @ M and
*S'U: cdcl_W-stgy** S' U and*
S'T': cdcl_W-stgy S' T' and
*T'U: cdcl_W-stgy** T' U*
 using *rtrancpl-cdcl_W-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 *rtrancpl-trans[OF st] rtrancpl-exists-last-with-prop[of cdcl_W-stgy S' T' -*
 $\lambda a -. \neg (\exists c. \text{trail } a = c @ \text{Marked } L i \# H @ M), \text{OF } S'T' T'U n]$
 by *meson*
 qed

lemma *beginning-not-marked-invert*:

assumes *A: M @ A = M' @ Marked K i # H and*
nm: ∀ m ∈ set M. ¬ is-marked m
 shows $\exists M. A = M @ \text{Marked } K i \# H$
 proof –
 have *A = drop (length M) (M' @ Marked K i # H)*
 using *arg-cong[OF A, of drop (length M)] by auto*
 moreover have *drop (length M) (M' @ Marked K i # H) = drop (length M) M' @ Marked K i # H*
 using *nm by (metis (no-types, lifting) A drop-Cons' drop-append marked-lit.disc(1) not-gr0*
nth-append nth-append-length nth-mem zero-less-diff)
 finally show *?thesis by fast*
 qed

lemma *cdcl_W-stgy-trail-has-new-marked-is-decide-step*:

assumes *cdcl_W-stgy* *S T*

$\neg (\exists c. \text{trail } S = c @ \text{Marked } L \ i \ \# \ H @ M) \text{ and}$

$(\lambda a \ b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L \ i \ \# \ H @ M))^* \ T \ U \text{ and}$

$\exists M'. \text{trail } U = M' @ \text{Marked } L \ i \ \# \ H @ M \text{ and}$

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.prem*s **by** (*meson decide-state-eq-compatible ns state-eq-ref state-eq-sym*)

next

case (*backtrack* *K j M1 M2 L' D T*) **note** *decomp* = *this*(1) **and** *cp* = *this*(3)

and *undef* = *this*(6) **and** *T* = *this*(7) **and** *trT* = *this*(12) **and** *ns* = *this*(4)

obtain *MS3* **where** *MS3*: *trail* *S* = *MS3* @ *M2* @ *Marked* *K (Suc j) # M1*

using *get-all-marked-decomposition-exists-prepend*[*OF decomp*] **by** *metis*

have *tl* (*M'* @ *Marked* *L i # H @ M*) = *tl* *M'* @ *Marked* *L i # H @ M*

using *lev trT T lev undef decomp* **by** (*cases M'*) (*auto simp: cdcl_W-M-level-inv-decomp*)

then have *M''*: *M1* = *tl* *M'* @ *Marked* *L i # H @ M*

using *arg-cong*[*OF trT*[*simplified*], *of tl*] *T decomp undef lev*

by (*simp add: cdcl_W-M-level-inv-decomp*)

have *False* **using** *nd MS3 T undef decomp unfolding M''* **by** *auto*

then show ?*case* **by** *fast*

qed *auto*

qed

qed

lemma *rtrancpl-cdcl_W-stgy-with-trail-end-has-trail-end:*

assumes $(\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L i \# H @ M))^{**} T U$ **and**
 $\exists M'. \text{trail } U = M' @ \text{Marked } L i \# H @ M$
shows $\exists M'. \text{trail } T = M' @ \text{Marked } L i \# H @ M$
using *assms* **by** (*induction rule: rtrancpl-induct*) *auto*

lemma *cdcl_W-o-cannot-learn:*

assumes
cdcl_W-o *y z* **and**
lev: *cdcl_W-M-level-inv* *y* **and**
trM: *trail y = c @ Marked Kh i # H* **and**
DL: $D + \{\#L\# \} \notin \# \text{learned-clss } y$ **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: $\forall T. \text{conflicting } y = \text{Some } T \longrightarrow \text{trail } y \models_{\text{as}} \text{CNot } T$ **and**
z: *trail z = c' @ 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: cdcl_W-o-induct-lev2*)

case (*backtrack K j M1 M2 L' D' T*) **note** *decomp = this(1)* **and** *confl = this(3)* **and** *levD = this(5)*
and *undef = this(6)* **and** *T = this(7)*

obtain *M3* **where** *M3*: *trail y = M3 @ M2 @ Marked K (Suc j) # M1*

using *decomp get-all-marked-decomposition-exists-prepend* **by** *metis*

have *M*: *trail y = c @ Marked Kh i # H* **using** *trM* **by** *simp*

have *H*: *get-all-levels-of-marked (trail y) = rev [1..<1 + backtrack-lvl y]*

using *lev unfolding cdcl_W-M-level-inv-def* **by** *auto*

have $c' @ \text{Marked } Kh i \# H = \text{Propagated } L' (D' + \{\#L\# \}) \# \text{trail (reduce-trail-to } M1 y)$

using *backtrack.premis(6) decomp undef T lev* **by** (*force simp: cdcl_W-M-level-inv-def*)

then obtain *d* **where** *d*: $M1 = d @ \text{Marked } Kh i \# H$

by (*metis (no-types) decomp in-get-all-marked-decomposition-trail-update-trail list.inject list.sel(3) marked-lit.distinct(1) self-append-conv2 tl-append2*)

have $i \in \text{set (get-all-levels-of-marked (M3 @ M2 @ Marked K (Suc j) \# d @ Marked Kh i \# H))}$
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 neq0-conv undef decomp lev* **by** (*fastforce simp: cdcl_W-M-level-inv-def*)

have *L-cKh*: $\text{atm-of } L \in \text{atm-of 'lits-of } (c @ [\text{Marked } Kh i])$

using *LH learned M DLD'[symmetric] confl* **by** (*fastforce simp add: image-iff*)

have *get-all-levels-of-marked (M3 @ M2 @ Marked K (j + 1) # M1)*

$= \text{rev } [1..<1 + \text{backtrack-lvl } y]$

using *lev unfolding cdcl_W-M-level-inv-def M3* **by** *auto*

from *arg-cong[OF this, of $\lambda a. (\text{Suc } j) \in \text{set } a]$* **have** *backtrack-lvl y $\geq j$* **by** *auto*

have DD' [*simp*]: $D = D'$

proof (*rule ccontr*)

assume $D \neq D'$

then have $L' \in \# D$ **using** DLD' **by** (*metis add.left-neutral count-single count-union diff-union-cancelR neq0-conv union-single-eq-member*)

then have *get-level (trail y) L' \leq get-maximum-level (trail y) D*

using *get-maximum-level-ge-get-level* **by** *blast*

```

moreover {
  have get-maximum-level (trail y) D = get-maximum-level H D
    using DH unfolding M by (simp add: get-maximum-level-skip-beginning)
  moreover
    have get-all-levels-of-marked (trail y) = rev [1..1 + backtrack-lvl y]
      using lev unfolding cdclW-M-level-inv-def by auto
    then have get-all-levels-of-marked H = rev [1..i]
      unfolding M by (auto dest: append-cons-eq-upt-length-i
        simp add: rev-swap[symmetric])
    then have get-maximum-possible-level H < i
      using get-maximum-possible-level-max-get-all-levels-of-marked[of H] ⟨i > 0⟩ by auto
  ultimately have get-maximum-level (trail y) D < i
    by (metis (full-types) dual-order.strict-trans nat-neq-iff not-le
      get-maximum-possible-level-ge-get-maximum-level) }
moreover
  have L ∈# D'
    by (metis DLD' ⟨D ≠ D'⟩ add.left-neutral count-single count-union diff-union-cancelR
      neq0-conv union-single-eq-member)
  then have get-maximum-level (trail y) D' ≥ get-level (trail y) L
    using get-maximum-level-ge-get-level by blast
  moreover {
    have get-all-levels-of-marked (c @ [Marked Kh i]) = rev [i..backtrack-lvl y + 1]
      using append-cons-eq-upt-length-i-end[of rev (get-all-levels-of-marked H) i
        rev (get-all-levels-of-marked c) Suc 0 Suc (backtrack-lvl y)] H
      unfolding M apply (auto simp add: rev-swap[symmetric])
      by (metis (no-types, hide-lams) Nil-is-append-conv Suc-le-eq less-Suc-eq list.sel(1)
        rev.simps(2) rev-rev-ident upt-Suc upt-rec)
    have get-level (trail y) L = get-level (c @ [Marked Kh i]) L
      using L-cKh LH unfolding M by simp
    have get-level (c @ [Marked Kh i]) L ≥ i
      using L-cKh
      ⟨get-all-levels-of-marked (c @ [Marked Kh i]) = rev [i..backtrack-lvl y + 1]⟩
      backtrack.hyps(2) calculation(1,2) by auto
    then have get-level (trail y) L ≥ i
      using M ⟨get-level (trail y) L = get-level (c @ [Marked Kh i]) L⟩ by auto }
  moreover have get-maximum-level (trail y) D' < get-level (trail y) L
    using ⟨j ≤ backtrack-lvl y⟩ backtrack.hyps(2,5) calculation(1-4) by linarith
  ultimately show False using backtrack.hyps(4) by linarith
qed
then have LL': L = L' using DLD' by auto
have nd: no-dup (trail y) using lev unfolding cdclW-M-level-inv-def by auto

{ assume D: D' = {#}
  then have j: j = 0 using levD by auto
  have ∀ m ∈ set M1. ¬is-marked m
    using H unfolding M3 j
    by (auto simp add: rev-swap[symmetric] get-all-levels-of-marked-no-marked
      dest!: append-cons-eq-upt-length-i)
  then have False using d by auto
}
moreover {
  assume D[simp]: D' ≠ {#}
  have i ≤ j
    using H unfolding M3 d by (auto simp add: rev-swap[symmetric]
      dest: upt-decomp-lt)

```

```

have  $j > 0$  apply (rule ccontr)
  using  $H \langle i > 0 \rangle$  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 (trail y)  $L'' =$  get-maximum-level (trail y)  $D'$ 
  using get-maximum-level-exists-lit-of-max-level[OF  $D$ , of trail y] by auto
have  $L''M$ : atm-of  $L'' \in$  atm-of 'lits-of (trail y)
  using get-rev-level-ge-0-atm-of-in[of 0 rev (trail y)  $L''$ ]  $\langle j > 0 \rangle$  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 (trail y)  $L'' =$  get-level  $H\ L''$ 
    using  $L''H$  unfolding  $M$  by simp
    ultimately have False
    using levD  $\langle j > 0 \rangle$  get-rev-level-in-levels-of-marked[of rev  $H\ 0\ L''$ ]  $\langle i \leq j \rangle$ 
    unfolding  $L''D'$ [symmetric] nd by 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
 $D + \{\#L\} \notin$  learned-clss  $y$  and
 $DH$ : atms-of  $D \subseteq$  atm-of 'lits-of  $H$  and
 $LH$ : atm-of  $L \notin$  atm-of 'lits-of  $H$  and
 $\forall T.$  conflicting  $y =$  Some  $T \longrightarrow$  trail  $y \models_{as} CNot\ T$  and
trail  $z = c' @$  Marked Kh  $i \# H$ 
shows  $D + \{\#L\} \notin$  learned-clss  $z$ 
using assms

```

proof induction

case conflict'

then show ?case

unfolding full1-def **using** tranclp- $cdcl_W$ -cp-learned-clause-inv **by** auto

next

case (other' $T\ U$) **note** $o =$ this(1) **and** $cp =$ this(3) **and** $lev =$ this(4) **and** $trY =$ this(5) **and**
 $notin =$ this(6) **and** $DH =$ this(7) **and** $LH =$ this(8) **and** $confl =$ this(9) **and** $trU =$ this(10)

obtain c' **where** c' : trail $T = c' @$ Marked Kh $i \# H$

using cp beginning-not-marked-invert[of - trail $T\ c'$ Kh $i\ H$]

$rtranclp$ - $cdcl_W$ -cp-dropWhile-trail[of $T\ U$] **unfolding** trU full-def **by** fastforce

show ?case

using $cdcl_W$ -o-cannot-learn[OF o lev trY $notin$ DH LH $confl$ c']
 $rtranclp$ - $cdcl_W$ - cp -learned-clause-inv cp unfolding $full$ -def by auto
qed

lemma $rtranclp$ - $cdcl_W$ -stgy-with-trail-end-has-not-been-learned:

assumes $(\lambda a b. cdcl_W$ -stgy a $b \wedge (\exists c. trail$ $a = c @$ $Marked$ K $i \#$ $H @ []))^{**} S$ z and
 $cdcl_W$ -all-struct-inv S and
 $trail$ $S = c @$ $Marked$ K $i \#$ H and
 $D + \{\#L\# \} \notin \#$ $learned$ -clss S and
 DH : $atms$ -of $D \subseteq atm$ -of ' $lits$ -of H and
 LH : atm -of $L \notin atm$ -of ' $lits$ -of H and
 $\exists c'. trail$ $z = c' @$ $Marked$ K $i \#$ H
shows $D + \{\#L\# \} \notin \#$ $learned$ -clss z
using $assms(1-4,7)$

proof (induction rule: $rtranclp$ -induct)

case $base$

then show ?case by auto[1]

next

case (step T U) note $st = this(1)$ and $s = this(2)$ and $IH = this(3)[OF$ $this(4-6)]$
and $lev = this(4)$ and $trS = this(5)$ and DL - $S = this(6)$ and $trU = this(7)$

obtain c where c : $trail$ $T = c @$ $Marked$ K $i \#$ H using s by auto

obtain c' where c' : $trail$ $U = c' @$ $Marked$ K $i \#$ H using trU by blast

have $cdcl_W^{**} S$ 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) $mono$ - $rtranclp$)

then have $cdcl_W$ -stgy $^{**} S$ T

using st by blast

then show ?thesis

using $rtranclp$ - $cdcl_W$ -stgy- $rtranclp$ - $cdcl_W$ by blast

qed

then have lev' : $cdcl_W$ -all-struct-inv T

using $rtranclp$ - $cdcl_W$ -all-struct-inv-inv[of S T] lev by auto

then have $confl'$: $\forall Ta. conflicting$ $T = Some$ $Ta \longrightarrow trail$ $T \models_{as}$ $CNot$ Ta

unfolding $cdcl_W$ -all-struct-inv-def $cdcl_W$ -conflicting-def by blast

show ?case

apply (rule $cdcl_W$ -stgy-with-trail-end-has-not-been-learned[OF - - c - DH LH $confl'$ c'])

using s lev' IH c unfolding $cdcl_W$ -all-struct-inv-def by blast+

qed

lemma $cdcl_W$ -stgy-new-learned-clause:

assumes $cdcl_W$ -stgy S T and

lev : $cdcl_W$ - M -level-inv S and

$E \notin \#$ $learned$ -clss S and

$E \in \#$ $learned$ -clss T

shows $\exists S'. backtrack$ S $S' \wedge conflicting$ $S = Some$ $E \wedge full$ $cdcl_W$ - cp S' T

using $assms$

proof induction

case $confl'$

then show ?case unfolding $full1$ -def by (auto dest: $tranclp$ - $cdcl_W$ - cp -learned-clause-inv)

next

case (other' T U) note $o = this(1)$ and $cp = this(3)$ and not -yet = $this(5)$ and $learned = this(6)$

have $E \in \#$ $learned$ -clss T

using $learned$ cp $rtranclp$ - $cdcl_W$ - cp -learned-clause-inv unfolding $full$ -def by auto

then have *backtrack S T and conflicting S = Some E*
using *cdcl_W-o-new-clause-learned-is-backtrack-step[OF - not-yet o] lev* **by** *blast+*
then show *?case* **using** *cp* **by** *blast*
qed

lemma *cdcl_W-stgy-no-relearned-clause:*

assumes

invR: cdcl_W-all-struct-inv R and
*st': cdcl_W-stgy** R S and*
bt: backtrack S T and
confl: conflicting S = Some E and
already-learned: E ∈# clauses S and
R: trail R = []

shows *False*

proof –

have *M-lev: cdcl_W-M-level-inv R*

using *invR unfolding cdcl_W-all-struct-inv-def* **by** *auto*

have *cdcl_W-M-level-inv S*

using *M-lev assms(2) rtranclp-cdcl_W-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 (trail S) D) (update-conflicting None S)))

and

decomp: (Marked K (Suc (get-maximum-level (trail S) D)) # M1, M2-loc) ∈
set (get-all-marked-decomposition (trail S)) and

k: get-level (trail S) L = backtrack-lvl S and

level: get-level (trail S) L = get-maximum-level (trail S) (D + {#L#}) and

confl-S: conflicting S = Some (D + {#L#}) and

i: i = get-maximum-level (trail S) D 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: cdcl_W-all-struct-inv S*

using *invR rtranclp-cdcl_W-all-struct-inv-inv rtranclp-cdcl_W-stgy-rtranclp-cdcl_W st'* **by** *blast*

then have *confl: cdcl_W-conflicting S unfolding cdcl_W-all-struct-inv-def* **by** *blast*

then have *trail S ⊨_{as} CNot (D + {#L#}) unfolding cdcl_W-conflicting-def confl-S* **by** *auto*

then have *MD: trail S ⊨_{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-lvls-M: get-all-levels-of-marked (trail S) = rev [1..<Suc (backtrack-lvl 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 ⊆ atm-of ' lits-of M1*

using *backtrack-atms-of-D-in-M1[OF lev' undef - decomp - - - T] confl-S conf T decomp k level*
lev' i undef unfolding 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:*

$\forall x \in \text{atms-of } D. x \notin \text{atm-of ' lits-of } (M2 @ [\text{Marked } K (\text{get-maximum-level } (\text{trail } S) D + 1)])$
apply *(rule vars-of-D distinct-atms-of-incl-not-in-other[of*

$M2 @ \text{Marked } K \text{ (get-maximum-level (trail } S) \text{ } D + 1) \# [] \text{ } M1 \text{ } D])$
using $\langle \text{no-dup (trail } S) \rangle M \text{ vars-of-} D$ **by** *simp-all*
have $M1\text{-}D: M1 \models_{as} CNot \text{ } D$
using $\text{vars-in-} M1 \text{ true-annots-remove-if-notin-vars[of } M2 @ \text{Marked } K (i + 1) \# [] \text{ } M1 \text{ } CNot \text{ } D]$
 $\langle \text{trail } S \models_{as} CNot \text{ } D \rangle M$ **by** *simp*

have $\text{get-lvls-}M: \text{get-all-levels-of-marked (trail } S) = \text{rev } [1..<Suc \text{ (backtrack-lvl } S)]$
using *lev'* **unfolding** $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ **by** *auto*
then have $\text{backtrack-lvl } S > 0$ **unfolding** M **by** $(\text{auto split: split-if-asm simp add: upt.simps}(2))$

obtain $M1' \text{ } K' \text{ } Ls$ **where**
 $M': \text{trail } S = Ls @ \text{Marked } K' \text{ (backtrack-lvl } S) \# M1'$ **and**
 $Ls: \forall l \in \text{set } Ls. \neg \text{is-marked } l$ **and**
 $\text{set } M1 \subseteq \text{set } M1'$
proof –
let $?Ls = \text{takeWhile (Not } o \text{ is-marked) (trail } S)$
have $MLs: \text{trail } S = ?Ls @ \text{dropWhile (Not } o \text{ is-marked) (trail } S)$
by *auto*
have $\text{dropWhile (Not } o \text{ is-marked) (trail } S) \neq []$ **unfolding** M **by** *auto*
moreover
from $\text{hd-dropWhile[OF this]}$ **have** $\text{is-marked(hd (dropWhile (Not } o \text{ is-marked) (trail } S))}$
by *simp*
ultimately
obtain $K' \text{ } K'k$ **where**
 $K'k: \text{dropWhile (Not } o \text{ is-marked) (trail } S)$
 $= \text{Marked } K' \text{ } K'k \# \text{tl (dropWhile (Not } o \text{ is-marked) (trail } S))$
by $(\text{cases dropWhile (Not } o \text{ is-marked) (trail } S);$
 $\text{cases hd (dropWhile (Not } o \text{ is-marked) (trail } S))})$
simp-all
moreover have $\forall l \in \text{set } ?Ls. \neg \text{is-marked } l$ **using** set-takeWhileD **by** *force*
moreover
have $\text{get-all-levels-of-marked (trail } S)$
 $= K'k \# \text{get-all-levels-of-marked(tl (dropWhile (Not } o \text{ is-marked) (trail } S))}$
apply $(\text{subst } MLs, \text{subst } K'k)$
using $\text{calculation}(2)$ **by** $(\text{auto simp add: get-all-levels-of-marked-no-marked})$
then have $K'k = \text{backtrack-lvl } S$
using $\text{calculation}(2)$ **by** $(\text{auto split: split-if-asm simp add: get-lvls-}M \text{ upt.simps}(2))$
moreover have $\text{set } M1 \subseteq \text{set (tl (dropWhile (Not } o \text{ is-marked) (trail } S))}$
unfolding M **by** $(\text{induction } M2) \text{ auto}$
ultimately show $?thesis$ **using** $\text{that } MLs$ **by** *metis*
qed

have $\text{get-lvls-}M: \text{get-all-levels-of-marked (trail } S) = \text{rev } [1..<Suc \text{ (backtrack-lvl } S)]$
using *lev'* **unfolding** $\text{cdcl}_W\text{-}M\text{-level-inv-def}$ **by** *auto*
then have $\text{backtrack-lvl } S > 0$ **unfolding** M **by** $(\text{auto split: split-if-asm simp add: upt.simps}(2) \text{ } i)$

have $M1'\text{-}D: M1' \models_{as} CNot \text{ } D$ **using** $M1\text{-}D \langle \text{set } M1 \subseteq \text{set } M1' \rangle$ **by** $(\text{auto intro: true-annots-mono})$
have $\neg L \in \text{lits-of (trail } S)$ **using** $\text{conf confl-}S$ **unfolding** $\text{cdcl}_W\text{-conflicting-def}$ **by** *auto*
have $\text{lvls-}M1': \text{get-all-levels-of-marked } M1' = \text{rev } [1..<\text{backtrack-lvl } S]$
using $\text{get-lvls-}M \text{ } Ls$ **by** $(\text{auto simp add: get-all-levels-of-marked-no-marked } M'$
 $\text{split: split-if-asm simp add: upt.simps}(2))$
have $L\text{-notin: atm-of } L \in \text{atm-of 'lits-of } Ls \vee \text{atm-of } L = \text{atm-of } K'$
proof (rule ccontr)
assume $\neg ?thesis$
then have $\text{atm-of } L \notin \text{atm-of 'lits-of (Marked } K' \text{ (backtrack-lvl } S) \# \text{rev } Ls)$ **by** *simp*

```

    then have get-level (trail S) L = get-level M1' L
      unfolding M' by auto
    then show False using get-level-in-levels-of-marked[of M1' L] ⟨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 = None using ⟨decide Y Y'⟩ by auto
have cdclW** R Y by (simp add: RY rtrancpl-cdclW-stgy-rtrancpl-cdclW)
then have init-clss Y = init-clss R using rtrancpl-cdclW-init-clss[of R Y] M-lev by auto
{ assume DL: D + {#L#} ∈ # 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 !Y-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#}  $\notin$  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)  $\langle \text{set } M1 \subseteq \text{set } M1 \rangle$  image-mono order-trans
        vars-of-D lits-of-def)
    using L-notin  $\langle \text{no-dup } (\text{trail } S) \rangle$  unfolding M' by (auto simp add: image-iff lits-of-def)
  then have False
  using already-learned DL confl st' M-lev unfolding M'
  by (simp add:  $\langle \text{init-clss } Y = \text{init-clss } R \rangle$  clauses-def confl-S
      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: rtrancp-cdclW-cp-no-more-clauses)
  next
    case (other' S') note o = this(1) and full = this(3)
    have [simp]: clauses T = clauses S'
      using full unfolding full-def by (auto dest: rtrancp-cdclW-cp-no-more-clauses)
    show ?thesis
      using o IH
    proof (cases rule: cdclW-o-rule-cases)
      case backtrack
      moreover
      have cdclW-all-struct-inv S
      using invR rtrancp-cdclW-stgy-cdclW-all-struct-inv st by blast
      then have cdclW-M-level-inv S
      unfolding cdclW-all-struct-inv-def by auto
    ultimately obtain E where
      conflicting S = Some E and
      cls-S': clauses S' = {#E#} + clauses S
      using  $\langle \text{cdcl}_W\text{-M-level-inv } S \rangle$ 
      by (induction rule: backtrack-induction-lev2) (auto simp: cdclW-M-level-inv-decomp)
  end
end

```

```

    then have  $E \notin \# \text{ clauses } S$ 
    using  $\text{cdcl}_W\text{-stgy-no-relearned-clause } R \text{ invR local.backtrack st by blast}$ 
    then show  $?thesis$  using  $IH$  by (simp add:  $\text{distinct-mset-add-single cls-S'}$ )
  qed auto
qed
qed

```

```

lemma  $\text{cdcl}_W\text{-stgy-distinct-mset-clauses}$ :
  assumes
     $st: \text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$  and
     $\text{no-duplicate-clause: distinct-mset } N$  and
     $\text{no-duplicate-in-clause: distinct-mset-mset } N$ 
  shows  $\text{distinct-mset (clauses } S)$ 
  using  $\text{rtrancp-cdcl}_W\text{-stgy-distinct-mset-clauses}[OF - st] \text{ assms}$ 
  by (auto simp:  $\text{cdcl}_W\text{-all-struct-inv-def distinct-cdcl}_W\text{-state-def}$ )

```

17.9 Decrease of a measure

```

fun  $\text{cdcl}_W\text{-measure}$  where
 $\text{cdcl}_W\text{-measure } S =$ 
   $[(\exists :: \text{nat}) \wedge (\text{card (atms-of-msu (init-clss } S))) - \text{card (set-mset (learned-clss } S)),$ 
     $\text{if conflicting } S = \text{None then } 1 \text{ else } 0,$ 
     $\text{if conflicting } S = \text{None then } \text{card (atms-of-msu (init-clss } S)) - \text{length (trail } S)$ 
     $\text{else length (trail } S)$ 
  ]

```

```

lemma  $\text{length-model-le-vars-all-inv}$ :
  assumes  $\text{cdcl}_W\text{-all-struct-inv } S$ 
  shows  $\text{length (trail } S) \leq \text{card (atms-of-msu (init-clss } S))$ 
  using  $\text{assms length-model-le-vars}[of S] \text{ unfolding } \text{cdcl}_W\text{-all-struct-inv-def}$ 
  by (auto simp:  $\text{cdcl}_W\text{-M-level-inv-decomp}$ )
end

```

```

context  $\text{cdcl}_W$ 
begin

```

```

lemma  $\text{learned-clss-less-upper-bound}$ :
  fixes  $S :: 'st$ 
  assumes
     $\text{distinct-cdcl}_W\text{-state } S$  and
     $\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$ 
  shows  $\text{card(set-mset (learned-clss } S)) \leq \exists \wedge \text{card (atms-of-msu (learned-clss } S))$ 
proof -
  have  $\text{set-mset (learned-clss } S) \subseteq \text{simple-clss (atms-of-msu (learned-clss } S))$ 
  apply (rule  $\text{simplified-in-simple-clss}$ )
  using  $\text{assms unfolding distinct-cdcl}_W\text{-state-def by auto}$ 
  then have  $\text{card(set-mset (learned-clss } S))$ 
     $\leq \text{card (simple-clss (atms-of-msu (learned-clss } S))$ 
  by (simp add:  $\text{simple-clss-finite card-mono}$ )
  then show  $?thesis$ 
  by (meson  $\text{atms-of-ms-finite simple-clss-card finite-set-mset order-trans}$ )
qed

```

```

lemma  $\text{lexn3}[intro!, \text{simp}]$ :
 $a < a' \vee (a = a' \wedge b < b') \vee (a = a' \wedge b = b' \wedge c < c')$ 
 $\implies ([a :: \text{nat}, b, c], [a', b', c']) \in \text{lexn } \{(x, y). x < y\} \exists$ 

```

```

apply auto
unfolding lexn-conv apply fastforce
unfolding lexn-conv apply auto
apply (metis append.simps(1) append.simps(2)) +
done

lemma cdclW-measure-decreasing:
fixes S :: 'st
assumes
  cdclW S S' and
  no-restart:
     $\neg(\text{learned-clss } S \subseteq\# \text{learned-clss } S' \wedge [] = \text{trail } S' \wedge \text{conflicting } S' = \text{None})$ 
  and
  learned-clss S  $\subseteq\#$  learned-clss S' and
  no-relearn:  $\bigwedge S'. \text{backtrack } S S' \implies \forall T. \text{conflicting } S = \text{Some } T \longrightarrow T \notin\# \text{learned-clss } S$ 
  and
  alien: no-strange-atm S and
  M-level: cdclW-M-level-inv S and
  no-taut:  $\forall s \in\# \text{learned-clss } S. \neg \text{tautology } s$  and
  no-dup: distinct-cdclW-state S and
  confl: cdclW-conflicting S
shows (cdclW-measure S', cdclW-measure S)  $\in \text{lexn } \{(a, b). a < b\} \text{ } 3$ 
using assms(1) M-level assms(2,3)
proof (induct rule: cdclW-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 cdclW-M-level-inv-decomp(2) marked-lit.sel(2) propagate'
    r-into-rtranclp rtranclp-cdclW-cp-consistent-inv trail-cons-trail undef)

let ?N = init-clss S
have no-strange-atm (cons-trail (Propagated L (C + {#L#})) S)
  using alien cdclW.propagate cdclW-no-strange-atm-inv propa M-level by blast
then have atm-of ' lits-of (Propagated L (C + {#L#})) # trail S
   $\subseteq$  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 cdclW:cdclW S (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))
    using cdclW.simps by blast
moreover
  have lev: cdclW-M-level-inv (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))

```

```

    using cdclW M-level cdclW-consistent-inv[OF cdclW] by auto
  then have no-dup: no-dup (Marked L (backtrack-lvl S + 1) # trail S)
    using undef unfolding cdclW-M-level-inv-def by auto
  have no-strange-atm (cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S))
    using M-level alien calculation(4) cdclW-no-strange-atm-inv by blast
  then have length (Marked L ((backtrack-lvl S) + 1) # (trail S))
    ≤ card (atms-of-msu (init-clss S))
    using no-dup clauses-def undef
    length-model-le-vars[of cons-trail (Marked L (backtrack-lvl S + 1)) (incr-lvl S)]
    by fastforce
  ultimately show ?case using conf by auto
next
case (skip L C' M D) note tr = this(1) and conf = this(2) and T = this(5)
show ?case using conf T unfolding clauses-def by (simp add: tr)
next
case conflict
then show ?case by simp
next
case resolve
then show ?case using finite unfolding clauses-def by simp
next
case (backtrack K i M1 M2 L D T) note decomp = this(1) and conf = this(3) and undef = this(6)
and
  T = this(7) and lev = this(8)
let ?S' = T
have bt: backtrack S ?S'
  using backtrack.hyps backtrack.intros[of S - - - D L K i] by auto
have D + {#L#} ∉ learned-clss S
  using no-relearn conf bt by auto
then have card-T:
  card (set-mset ({#D + {#L#}#} + learned-clss S)) = Suc (card (set-mset (learned-clss S)))
  by (simp add:)
have distinct-cdclW-state ?S'
  using bt M-level distinct-cdclW-state-inv no-dup other by blast
moreover have ∀ s ∈ #learned-clss ?S'. ¬ tautology s
  using learned-clss-are-not-tautologies[OF cdclW.other[OF cdclW-o.bj[OF
    cdclW-bj.backtrack[OF bt]]]] M-level no-taut confl by auto
ultimately have card (set-mset (learned-clss T)) ≤ 3 ^ card (atms-of-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))
  ≤ 3 ^ 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) ⊆ 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))
  ≤ card (atms-of-msu (init-clss S))
  by (meson atms-of-ms-finite card-mono finite-set-mset)
then have (3::nat) ^ card (atms-of-msu ({#D + {#L#}#} + learned-clss S))
  ≤ 3 ^ card (atms-of-msu (init-clss S)) by simp
ultimately have (3::nat) ^ card (atms-of-msu (init-clss S))
  ≥ card (set-mset ({#D + {#L#}#} + learned-clss S))
  using le-trans by blast
then show ?case using decomp undef diff-less-mono2 card-T T lev
  by (auto simp: cdclW-M-level-inv-decomp)

```

```

next
  case restart
  then show ?case using alien by (auto simp: state-eq-def simp del: state-simp)
next
  case (forget C T)
  then have  $C \in \# \text{ learned-clss } S$  and  $C \notin \# \text{ learned-clss } T$ 
    by auto
  then show ?case using forget(9) by (simp add: mset-leD)
qed

```

```

lemma propagate-measure-decreasing:
  fixes S :: 'st
  assumes propagate S S' and cdclW-all-struct-inv S
  shows (cdclW-measure S', cdclW-measure S) ∈ 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 *trancpl-cdcl_W-cp-measure-decreasing*:

fixes $S :: 'st$

assumes $cdcl_W\text{-cp}^{++} S S'$ **and** $cdcl_W\text{-all-struct-inv } S$

shows $(cdcl_W\text{-measure } S', cdcl_W\text{-measure } S) \in le_{rn} \{(a, b). a < b\} \text{ } 3$

using *assms*

proof *induction*

case *base*

then show *?case* **using** $cdcl_W\text{-cp-measure-decreasing}$ **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 $(cdcl_W\text{-measure } T, cdcl_W\text{-measure } S) \in le_{rn} \{a. case \ a \ of \ (a, b) \Rightarrow a < b\} \text{ } 3$ **by** *blast*

moreover have $(cdcl_W\text{-measure } U, cdcl_W\text{-measure } T) \in le_{rn} \{a. case \ a \ of \ (a, b) \Rightarrow a < b\} \text{ } 3$

using $cdcl_W\text{-cp-measure-decreasing}[OF \ step]$ $rtrancpl\text{-}cdcl_W\text{-all-struct-inv-inv } inv$

$trancpl\text{-}cdcl_W\text{-cp-trancpl-cdcl}_W[OF \ st]$

unfolding *trans-def* $rtrancpl\text{-unfold}$

by *blast*

ultimately show *?case* **using** $le_{rn}\text{-transI}[OF \ trans\text{-}le]$ **unfolding** *trans-def* **by** *blast*

qed

lemma *cdcl_W-stgy-step-decreasing*:

fixes $R \ S \ T :: 'st$

assumes $cdcl_W\text{-stgy } S \ T$ **and**

$cdcl_W\text{-stgy}^{**} R \ S$

$trail \ R = []$ **and**

$cdcl_W\text{-all-struct-inv } R$

shows $(cdcl_W\text{-measure } T, cdcl_W\text{-measure } S) \in le_{rn} \{(a, b). a < b\} \text{ } 3$

proof *–*

have $cdcl_W\text{-all-struct-inv } S$

using *assms*

by $(metis \ rtrancpl\text{-unfold} \ rtrancpl\text{-}cdcl_W\text{-all-struct-inv-inv} \ trancpl\text{-}cdcl_W\text{-stgy-trancpl-cdcl}_W)$

with *assms* **show** *?thesis*

proof *induction*

case $(conflict' \ V)$ **note** $cp = this(1)$ **and** $inv = this(5)$

show *?case*

using $trancpl\text{-}cdcl_W\text{-cp-measure-decreasing}[OF \ HOL.conjunct1[OF \ cp[unfolded \ full1\text{-}def]] \ inv]$

.

next

case $(other' \ T \ U)$ **note** $st = this(1)$ **and** $H = this(4,5,6,7)$ **and** $cp = this(3)$

have $cdcl_W\text{-all-struct-inv } T$

using $cdcl_W\text{-all-struct-inv-inv} \ other \ other'.hyps(1) \ other'.prems(4)$ **by** *blast*

from $trancpl\text{-}cdcl_W\text{-cp-measure-decreasing}[OF \ \text{this}]$

have *le-or-eq*: $(cdcl_W\text{-measure } U, cdcl_W\text{-measure } T) \in le_{rn} \{a. case \ a \ of \ (a, b) \Rightarrow a < b\} \text{ } 3 \vee$

$cdcl_W\text{-measure } U = cdcl_W\text{-measure } T$

using cp **unfolding** *full-def* $rtrancpl\text{-unfold}$ **by** *blast*

moreover

have $cdcl_W\text{-M-level-inv } S$

using $cdcl_W\text{-all-struct-inv-def} \ other'.prems(4)$ **by** *blast*

with st **have** $(cdcl_W\text{-measure } T, cdcl_W\text{-measure } S) \in le_{rn} \{a. case \ a \ of \ (a, b) \Rightarrow a < b\} \text{ } 3$

proof $(induction \ rule:cdcl_W\text{-o-induct-lev2})$

case $(decide \ T)$

then show *?case* **using** $decide\text{-measure-decreasing } H$ **by** *blast*

next

```

      case (backtrack K i M1 M2 L D T) note decomp = this(1) and undef = this(6) and T =
this(7)
    have bt: backtrack S T
    apply (rule backtrack-rule)
    using backtrack.hyps by auto
  then have no-relearn:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow T \notin \# \text{learned-clss } S$ 
    using cdclW-stgy-no-relearned-clause[of R S T] H
    unfolding cdclW-all-struct-inv-def clauses-def by auto
  have inv: cdclW-all-struct-inv S
    using ⟨cdclW-all-struct-inv S⟩ by blast
  show ?case
    apply (rule cdclW-measure-decreasing)
      using bt cdclW-bj.backtrack cdclW-o.bj other apply simp
      using bt T undef decomp inv unfolding cdclW-all-struct-inv-def
        cdclW-M-level-inv-def apply auto[]
      using bt T undef decomp inv unfolding cdclW-all-struct-inv-def
        cdclW-M-level-inv-def apply auto[]
      using bt no-relearn apply auto[]
      using inv unfolding cdclW-all-struct-inv-def apply simp
      using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def apply simp
      using inv unfolding cdclW-all-struct-inv-def apply simp
      using inv unfolding cdclW-all-struct-inv-def apply simp
      using inv unfolding cdclW-all-struct-inv-def by simp
    next
      case skip
      then show ?case by force
    next
      case resolve
      then show ?case by force
  qed
ultimately show ?case
  by (metis leW-transI transD trans-le)
qed
qed

```

lemma *trancp-cdcl_W-stgy-decreasing*:

```

fixes R S T :: 'st
assumes cdclW-stgy++ R S
trail R = [] and
cdclW-all-struct-inv R
shows (cdclW-measure S, cdclW-measure R) ∈ leW {(a, b). a < b} 3
using assms
apply induction
  using cdclW-stgy-step-decreasing[of R - R] apply blast
  using cdclW-stgy-step-decreasing[of - - R] trancp-into-rtrancp[of cdclW-stgy R]
  leW-transI[OF trans-le, of 3] unfolding trans-def by blast

```

lemma *trancp-cdcl_W-stgy-S0-decreasing*:

```

fixes R S T :: 'st
assumes pl: cdclW-stgy++ (init-state N) S and
no-dup: distinct-mset-mset N
shows (cdclW-measure S, cdclW-measure (init-state N)) ∈ leW {(a, b). a < b} 3
proof -
  have cdclW-all-struct-inv (init-state N)
    using no-dup unfolding cdclW-all-struct-inv-def by auto

```

then show *?thesis* using *pl* *tranclp-cdcl_W-stgy-decreasing init-state-trail* by *blast*
qed

lemma *wf-tranclp-cdcl_W-stgy*:
 $wf \{(S::'st, \text{init-state } N) \mid S \ N. \text{distinct-mset-mset } N \wedge \text{cdcl}_W\text{-stgy}^{++} (\text{init-state } N) \ S\}$
apply (*rule* *wf-wf-if-measure'-notation2*[*of le_W* $\{(a, b). a < b\}$ *3 - - cdcl_W-measure*])
apply (*simp add*: *wf wf-le_W*)
using *tranclp-cdcl_W-stgy-S0-decreasing* by *blast*
end

end
theory *DPLL-CDCL-W-Implementation*
imports *Partial-Annotated-Clausal-Logic*
begin

18 Simple Implementation of the DPLL and CDCL

18.1 Common Rules

18.1.1 Propagation

The following theorem holds:

lemma *lits-of-unfold*[*iff*]:
 $(\forall c \in \text{set } C. -c \in \text{lits-of } Ms) \longleftrightarrow Ms \models_{as} CNot (\text{mset } C)$
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* =
 $(\text{case } List.filter (\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of } M) \ l \text{ of}$
 $\quad a \# [] \Rightarrow \text{if } M \models_{as} CNot (\text{mset } l - \{\#a\}) \text{ then } Some \ a \text{ else } None$
 $\quad | - \Rightarrow None)$

definition *is-unit-clause-code* :: *'a literal list* \Rightarrow (*'a, 'b, 'c*) *marked-lit list*
 \Rightarrow *'a literal option* **where**
is-unit-clause-code *l M* =
 $(\text{case } List.filter (\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of } M) \ l \text{ of}$
 $\quad a \# [] \Rightarrow \text{if } (\forall c \in \text{set } (\text{remove1 } a \ l). -c \in \text{lits-of } M) \text{ then } Some \ a \text{ else } None$
 $\quad | - \Rightarrow None)$

lemma *is-unit-clause-is-unit-clause-code*[*code*]:
is-unit-clause *l M* = *is-unit-clause-code* *l M*
proof –
have *1*: $\bigwedge a. (\forall c \in \text{set } (\text{remove1 } a \ l). -c \in \text{lits-of } M) \longleftrightarrow M \models_{as} CNot (\text{mset } l - \{\#a\})$
using *lits-of-unfold*[*of remove1 - l, of - M*] by *simp*
thus *?thesis*
unfolding *is-unit-clause-code-def is-unit-clause-def 1* by *blast*
qed

lemma *is-unit-clause-some-undef*:
assumes *is-unit-clause* *l M* = *Some a*
shows *undefined-lit M a*
proof –
have $(\text{case } [a \leftarrow l . \text{atm-of } a \notin \text{atm-of ' lits-of } M] \text{ of } [] \Rightarrow None$


```

| [a] ⇒ if M ⊨as CNot (mset l - {#a#}) then Some a else None
| a # ab # xa ⇒ Map.empty xa) = Some a
using assms unfolding is-unit-clause-def .
hence a ∈ set [a ← l . atm-of a ∉ atm-of ' lits-of M]
apply (cases [a ← l . atm-of a ∉ atm-of ' lits-of M])
  apply simp
  apply (rename-tac aa list; case-tac list) by (auto split: split-if-asm)
hence atm-of a ∉ 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 ⇒ M ⊨_{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 (cases [a ← l . atm-of a ∉ atm-of ' lits-of M], simp)
apply simp
apply (rename-tac aa list; 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 (cases [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)
 $\Rightarrow c \in \text{set } l \wedge M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\}) \wedge \text{undefined-lit } M a \wedge a \in \text{set } c$
apply (induction l)
apply simp

by (auto split: option.splits dest: is-unit-clause-some-in is-unit-clause-some-CNot
is-unit-clause-some-undef)

lemma propagate-is-unit-clause-not-None:

assumes *dist*: distinct *c* **and**

M: $M \models_{as} CNot (mset\ c - \{\#a\# \})$ **and**

undef: undefined-lit *M* *a* **and**

ac: $a \in set\ c$

shows is-unit-clause *c* *M* $\neq None$

proof –

have $[a \leftarrow c . atm-of\ a \notin atm-of\ 'lits-of\ M] = [a]$

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*]:
 $find-first-unused-var\ l\ M = None \iff (\forall a \in set\ l. atm-of\ 'set\ a \subseteq atm-of\ 'M)$
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*:
 $find-first-unused-var\ l\ M = Some\ a \implies (\exists m \in set\ l. a \in set\ m \wedge a \notin M \wedge \neg a \notin M)$
by (*induct l*) (*auto split: option.splits dest: find-some*)

lemma *find-first-unused-var-undefined*:
 $find-first-unused-var\ l\ (lits-of\ Ms) = Some\ a \implies undefined-lit\ Ms\ a$
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* :: $int\ dpll_W\text{-marked-lits} \times int\ literal\ list\ list$

$\Rightarrow int\ dpll_W\text{-marked-lits} \times int\ literal\ list\ list$ **where**

DPLL-step = $(\lambda(Ms, N).$

(case find-first-unit-clause N Ms of

Some (L, -) \Rightarrow (Propagated L () # Ms, N)

| - \Rightarrow

if $\exists C \in set\ N. (\forall c \in set\ C. \neg c \in lits-of\ Ms)$

then

(case backtrack-split Ms of

(-, L # M) \Rightarrow (Propagated (\neg (lit-of L)) () # M, N)

| (-, -) \Rightarrow (Ms, N)

)

else

(case find-first-unused-var N (lits-of Ms) of

Some a \Rightarrow (Marked a () # Ms, N)

| None \Rightarrow (Ms, N))))

Example of propagation:

value *DPLL-step* ([*Marked (Neg 1) ()*], [*Pos (1::int), Neg 2*]))

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

abbreviation $toS \equiv \lambda(Ms::(int, unit, unit) \text{ marked-lit list})$
 $(N:: int \text{ literal list list}). (Ms, mset (\text{map } mset N))$
abbreviation $toS' \equiv \lambda(Ms::(int, unit, unit) \text{ marked-lit list},$
 $N:: int \text{ literal list list}). (Ms, mset (\text{map } mset N))$

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-dpll_W-step*:

assumes *step*: $(Ms', N') = \text{DPLL-step } (Ms, N)$
and *neq*: $(Ms, N) \neq (Ms', N')$
shows $dpll_W (toS Ms N) (toS Ms' N')$

proof –

let $?S = (Ms, mset (\text{map } mset N))$
{ fix $L E$
assume *unit*: $\text{find-first-unit-clause } N Ms = \text{Some } (L, E)$
hence $Ms'N: (Ms', N') = (\text{Propagated } L () \# Ms, N)$
using *step* **unfolding** *DPLL-step-def* **by** *auto*
obtain C **where**
 $C: C \in \text{set } N$ **and**
 $Ms: Ms \models_{as} C \text{Not } (mset C - \{\#L\# \})$ **and**
 $\text{undef}: \text{undefined-lit } Ms L$ **and**
 $L \in \text{set } C$ **using** *find-first-unit-clause-some[OF unit]* **by** *metis*
have $dpll_W (Ms, mset (\text{map } mset N))$
 $(\text{Propagated } L () \# \text{fst } (Ms, mset (\text{map } mset N)), \text{snd } (Ms, mset (\text{map } mset N)))$
apply $(\text{rule } dpll_W.\text{propagate})$
using $Ms \text{ undef } C \langle L \in \text{set } C \rangle$ **unfolding** *mem-set-multiset-eq* **by** $(\text{auto simp add: } C)$
hence $?thesis$ **using** $Ms'N$ **by** *auto*
}

moreover

{ assume *unit*: $\text{find-first-unit-clause } N Ms = \text{None}$
assume *exC*: $\exists C \in \text{set } N. Ms \models_{as} C \text{Not } (mset C)$
then obtain C **where** $C: C \in \text{set } N$ **and** $Ms: Ms \models_{as} C \text{Not } (mset C)$ **by** *auto*
then obtain $L M M'$ **where** $bt: \text{backtrack-split } Ms = (M', L \# M)$
using *step* *exC* **neq** **unfolding** *DPLL-step-def prod.case unit*
by $(\text{cases } \text{backtrack-split } Ms, \text{rename-tac } b, \text{case-tac } b) \text{ auto}$
hence *is-marked* L **using** *backtrack-split-snd-hd-marked[of Ms]* **by** *auto*
have $1: dpll_W (Ms, mset (\text{map } mset N))$
 $(\text{Propagated } (- \text{lit-of } L) () \# M, \text{snd } (Ms, mset (\text{map } mset N)))$
apply $(\text{rule } dpll_W.\text{backtrack}[OF - \langle \text{is-marked } L \rangle, \text{of }])$
using $C Ms bt$ **by** *auto*
moreover have $(Ms', N') = (\text{Propagated } (- (\text{lit-of } L)) () \# M, N)$
using *step* *exC* **unfolding** *DPLL-step-def bt prod.case unit* **by** *auto*
ultimately have $?thesis$ **by** *auto*
}

moreover

{ assume *unit*: $\text{find-first-unit-clause } N Ms = \text{None}$
assume *exC*: $\neg (\exists C \in \text{set } N. Ms \models_{as} C \text{Not } (mset C))$
obtain L **where** *unused*: $\text{find-first-unused-var } N (\text{lits-of } Ms) = \text{Some } L$
using *step* *exC* **neq** **unfolding** *DPLL-step-def prod.case unit*
by $(\text{cases } \text{find-first-unused-var } N (\text{lits-of } Ms)) \text{ auto}$
have $dpll_W (Ms, mset (\text{map } mset N))$
 $(\text{Marked } L () \# \text{fst } (Ms, mset (\text{map } mset N)), \text{snd } (Ms, mset (\text{map } mset N)))$
apply $(\text{rule } dpll_W.\text{decided}[of ?S L])$
using *find-first-unused-var-Some[OF unused]*
}

```

    by (auto simp add: Marked-Propagated-in-iff-in-lits-of atms-of-ms-def)
  moreover have  $(Ms', N') = (\text{Marked } L \ () \# Ms, N)$ 
    using step exC unfolding DPLL-step-def unused prod.case unit by auto
  ultimately have ?thesis by auto
}
ultimately show ?thesis by (cases find-first-unit-clause N Ms) auto
qed

lemma DPLL-step-stuck-final-state:
  assumes step:  $(Ms, N) = \text{DPLL-step } (Ms, N)$ 
  shows conclusive-dpllW-state (toS Ms N)
proof -
  have unit: find-first-unit-clause N Ms = None
    using step unfolding DPLL-step-def by (auto split: option.splits)

  { assume n:  $\exists C \in \text{set } N. Ms \models_{\text{as}} \text{CNot } (\text{mset } C)$ 
    hence Ms:  $(Ms, N) = (\text{case backtrack-split } Ms \text{ of } (x, []) \Rightarrow (Ms, N) \mid (x, L \# M) \Rightarrow (\text{Propagated } (- \text{lit-of } L) \ () \# M, N))$ 
      using step unfolding DPLL-step-def by (simp add: unit)

    have 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  $(- \text{lit-of } aa) \ () \# list$  using Ms by auto
      have is-marked aa using backtrack-split-snd-hd-marked[of Ms] bt bt' by auto
      moreover have fst (backtrack-split Ms) @ aa # list = Ms
        using backtrack-split-list-eq[of Ms] bt' by auto
      ultimately have False unfolding Ms by auto
      thus snd (backtrack-split Ms) = [] by blast
    qed

    hence ?thesis
      using n backtrack-snd-empty-not-marked[of Ms] unfolding conclusive-dpllW-state-def
      by (cases backtrack-split Ms) auto
  }
moreover {
  assume n:  $\neg (\exists C \in \text{set } N. Ms \models_{\text{as}} \text{CNot } (\text{mset } C))$ 
  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)  $\models_{\text{asm}} \text{snd } (toS Ms N)$  unfolding true-annots-def CNot-def Ball-def
  proof clarify
    fix x
    assume x:  $x \in \text{set-mset } (\text{clauses } (toS Ms N))$ 
    hence  $\neg Ms \models_{\text{as}} \text{CNot } x$  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  $\text{fst } (toS \ Ms \ N) \models_a x$ 
    using  $\text{total-not-CNot}[of \ lits-of \ Ms \ x]$  by (simp add: true-annot-def true-annots-true-cls)
  qed
  hence ?thesis unfolding  $\text{conclusive-dpll}_W\text{-state-def}$  by blast
}
ultimately show ?thesis by blast
qed

```

18.2.2 Adding invariants

Invariant tested in the function $\text{function } DPLL\text{-ci} :: \text{int } dpll_W\text{-marked-lits} \Rightarrow \text{int literal list list}$

$\Rightarrow \text{int } dpll_W\text{-marked-lits} \times \text{int literal list list}$ **where**

```

DPLL-ci Ms N =
  (if  $\neg dpll_W\text{-all-inv } (Ms, \text{mset } (\text{map } \text{mset } N))$ 
   then  $(Ms, N)$ 
   else
    let  $(Ms', N') = DPLL\text{-step } (Ms, N)$  in
    if  $(Ms', N') = (Ms, N)$  then  $(Ms, N)$  else  $DPLL\text{-ci } Ms' N$ )
  by fast+

```

termination

```

proof (relation  $\{(S', S). (toS' \ S', toS' \ S) \in \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}\}$ )
  show wf  $\{(S', S). (toS' \ S', toS' \ S) \in \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}\}$ 
  using wf-if-measure-f[ $OF \ dpll_W\text{-wf, of } toS'$ ] by auto

```

next

```

fix Ms :: int dpll_W-marked-lits and N x xa y
assume  $\neg \neg dpll_W\text{-all-inv } (toS \ Ms \ N)$ 
and step:  $x = DPLL\text{-step } (Ms, N)$ 
and x:  $(xa, y) = x$ 
and  $(xa, y) \neq (Ms, N)$ 
thus  $((xa, N), Ms, N) \in \{(S', S). (toS' \ S', toS' \ S) \in \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W \ S \ S'\}\}$ 
  using  $DPLL\text{-step-is-a-dpll}_W\text{-step } dpll_W\text{-same-clauses split-conv}$  by fastforce
qed

```

No invariant tested $\text{function } (domintros) \ DPLL\text{-part} :: \text{int } dpll_W\text{-marked-lits} \Rightarrow \text{int literal list list}$

$\Rightarrow \text{int } dpll_W\text{-marked-lits} \times \text{int literal list list}$ **where**

```

DPLL-part Ms N =
  (let  $(Ms', N') = DPLL\text{-step } (Ms, N)$  in
  if  $(Ms', N') = (Ms, N)$  then  $(Ms, N)$  else  $DPLL\text{-part } Ms' N$ )
  by fast+

```

lemma $\text{snd-DPLL-step}[simp]$:

```

  snd  $(DPLL\text{-step } (Ms, N)) = N$ 
  unfolding  $DPLL\text{-step-def}$  by (auto split: split-if option.splits prod.splits list.splits)

```

lemma $dpll_W\text{-all-inv-implicS-2-eq3-and-dom}$:

```

  assumes  $dpll_W\text{-all-inv } (Ms, \text{mset } (\text{map } \text{mset } N))$ 
  shows  $DPLL\text{-ci } Ms \ N = DPLL\text{-part } Ms \ N \wedge DPLL\text{-part-dom } (Ms, N)$ 
  using assms

```

proof (induct rule: $DPLL\text{-ci.induct}$)

case $(1 \ Ms \ N)$

have $\text{snd } (DPLL\text{-step } (Ms, N)) = N$ **by** auto

then obtain Ms' **where** $Ms': DPLL\text{-step } (Ms, N) = (Ms', N)$ **by** (cases $DPLL\text{-step } (Ms, N)$) auto

have $\text{inv}': dpll_W\text{-all-inv } (toS \ Ms' \ N)$ **by** (metis (mono-tags) 1.prem DPLL-step-is-a-dpll_W-step $Ms' \ dpll_W\text{-all-inv old.prod.inject}$)

{ assume $(Ms', N) \neq (Ms, N)$
hence $DPLL\text{-}ci\ Ms'\ N = DPLL\text{-}part\ Ms'\ N \wedge DPLL\text{-}part\text{-}dom\ (Ms', N)$ **using** $1(1)[of - Ms'\ N]$
 Ms'
 $1(2)\ inv'$ **by** *auto*
hence $DPLL\text{-}part\text{-}dom\ (Ms, N)$ **using** $DPLL\text{-}part.\text{domintros}\ Ms'$ **by** *fastforce*
moreover have $DPLL\text{-}ci\ Ms\ N = DPLL\text{-}part\ Ms\ N$ **using** $1.\text{prems}\ DPLL\text{-}part.\text{psimps}\ Ms'$
 $\langle DPLL\text{-}ci\ Ms'\ N = DPLL\text{-}part\ Ms'\ N \wedge DPLL\text{-}part\text{-}dom\ (Ms', N) \rangle \langle DPLL\text{-}part\text{-}dom\ (Ms, N) \rangle$ **by**
auto
ultimately have $?case$ **by** *blast*
}
moreover {
assume $(Ms', N) = (Ms, N)$
hence $?case$ **using** $DPLL\text{-}part.\text{domintros}\ DPLL\text{-}part.\text{psimps}\ Ms'$ **by** *fastforce*
}
ultimately show $?case$ **by** *blast*
qed

lemma $DPLL\text{-}ci\text{-}dpll_W\text{-}rtranclp$:

assumes $DPLL\text{-}ci\ Ms\ N = (Ms', N')$
shows $dpll_W^{**}\ (toS\ Ms\ N)\ (toS\ Ms'\ N)$
using *assms*

proof (*induct* $Ms\ N$ *arbitrary*: $Ms'\ N'$ *rule*: $DPLL\text{-}ci.\text{induct}$)

case $(1\ Ms\ N\ Ms'\ N')$ **note** $IH = \text{this}(1)$ **and** $\text{step} = \text{this}(2)$

obtain $S_1\ S_2$ **where** $S: (S_1, S_2) = DPLL\text{-}step\ (Ms, N)$ **by** (*cases* $DPLL\text{-}step\ (Ms, N)$) *auto*

{ assume $\neg dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
hence $(Ms, N) = (Ms', N)$ **using** step **by** *auto*
hence $?case$ **by** *auto*
}

moreover

{ assume $dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
and $(S_1, S_2) = (Ms, N)$
hence $?case$ **using** $S\ \text{step}$ **by** *auto*
}

moreover

{ assume $dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
and $(S_1, S_2) \neq (Ms, N)$
moreover obtain $S_1'\ S_2'$ **where** $DPLL\text{-}ci\ S_1\ N = (S_1', S_2')$ **by** (*cases* $DPLL\text{-}ci\ S_1\ N$) *auto*
moreover have $DPLL\text{-}ci\ Ms\ N = DPLL\text{-}ci\ S_1\ N$ **using** $DPLL\text{-}ci.\text{simps}[of\ Ms\ N]$ *calculation*
proof –

have (*case* (S_1, S_2) *of* $(ms, lss) \Rightarrow$
if $(ms, lss) = (Ms, N)$ *then* (Ms, N) *else* $DPLL\text{-}ci\ ms\ N = DPLL\text{-}ci\ Ms\ N$
using $S\ DPLL\text{-}ci.\text{simps}[of\ Ms\ N]$ *calculation* **by** *presburger*
hence (*if* $(S_1, S_2) = (Ms, N)$ *then* (Ms, N) *else* $DPLL\text{-}ci\ S_1\ N = DPLL\text{-}ci\ Ms\ N$
by *fastforce*
thus $?thesis$
using $\text{calculation}(2)$ **by** *presburger*

qed

ultimately have $dpll_W^{**}\ (toS\ S_1'\ N)\ (toS\ Ms'\ N)$ **using** $IH[of\ (S_1, S_2)\ S_1\ S_2]$ $S\ \text{step}$ **by** *simp*

moreover have $dpll_W\ (toS\ Ms\ N)\ (toS\ S_1\ N)$

by (*metis* $DPLL\text{-}step\text{-}is\text{-}a\text{-}dpll_W\text{-}step\ S\ \langle (S_1, S_2) \neq (Ms, N) \rangle\ \text{prod.sel}(2)\ \text{snd}\text{-}DPLL\text{-}step$)

ultimately have $?case$ **by** (*metis* (*mono-tags*, *hide-lams*) $IH\ S\ \langle (S_1, S_2) \neq (Ms, N) \rangle$)

$\langle DPLL\text{-}ci\ Ms\ N = DPLL\text{-}ci\ S_1\ N \rangle \langle dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N) \rangle\ \text{converse}\text{-}rtranclp\text{-}into\text{-}rtranclp$
local.step

```

}
ultimately show ?case by blast
qed

```

lemma *dpll_W-all-inv-dpll_W-tranclp-irrefl*:

```

assumes dpllW-all-inv (Ms, N)
and dpllW++ (Ms, N) (Ms, N)
shows False

```

proof –

```

have 1: wf {(S', S). dpllW-all-inv S ∧ dpllW++ S S'} using dpllW-wf-tranclp by auto
have ((Ms, N), (Ms, N)) ∈ {(S', S). dpllW-all-inv S ∧ dpllW++ S S'} using assms by auto
thus False using wf-not-refl[OF 1] by blast

```

qed

lemma *DPLL-ci-final-state*:

```

assumes step: DPLL-ci Ms N = (Ms, N)
and inv: dpllW-all-inv (toS Ms N)
shows conclusive-dpllW-state (toS Ms N)

```

proof –

```

have st: dpllW** (toS Ms N) (toS Ms N) using DPLL-ci-dpllW-rtranclp[OF step] .

```

```

have DPLL-step (Ms, N) = (Ms, N)

```

proof (rule ccontr)

```

obtain Ms' N' where Ms'N: (Ms', N') = DPLL-step (Ms, N)

```

```

by (cases DPLL-step (Ms, N)) auto

```

```

assume ¬ ?thesis

```

```

hence DPLL-ci Ms' N = (Ms, N) using step inv st Ms'N[symmetric] by fastforce

```

```

hence dpllW++ (toS Ms N) (toS Ms N)

```

```

by (metis DPLL-ci-dpllW-rtranclp DPLL-step-is-a-dpllW-step Ms'N ⟨DPLL-step (Ms, N) ≠ (Ms,

```

N)⟩

```

prod.sel(2) rtranclp-into-tranclp2 snd-DPLL-step)

```

```

thus False using dpllW-all-inv-dpllW-tranclp-irrefl inv by auto

```

qed

```

thus ?thesis using DPLL-step-stuck-final-state[of Ms N] by simp

```

qed

lemma *DPLL-step-obtains*:

```

obtains Ms' where (Ms', N) = DPLL-step (Ms, N)

```

```

unfolding DPLL-step-def by (metis (no-types, lifting) DPLL-step-def prod.collapse snd-DPLL-step)

```

lemma *DPLL-ci-obtains*:

```

obtains Ms' where (Ms', N) = DPLL-ci Ms N

```

proof (induct rule: DPLL-ci.induct)

```

case (1 Ms N) note IH = this(1) and that = this(2)

```

```

obtain S where SN: (S, N) = DPLL-step (Ms, N) using DPLL-step-obtains by metis

```

```

{ assume ¬ dpllW-all-inv (toS Ms N)

```

```

hence ?case using that by auto

```

```

}

```

moreover {

```

assume n: (S, N) ≠ (Ms, N)

```

```

and inv: dpllW-all-inv (toS Ms N)

```

```

have ∃ ms. DPLL-step (Ms, N) = (ms, N)

```

```

by (metis ⟨∧thesis⟩. (∧S. (S, N) = DPLL-step (Ms, N) ⇒ thesis) ⇒ thesis)

```

```

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  $step: DPLL-ci\ Ms\ N = (Ms', N')$ 
shows  $DPLL-ci\ Ms'\ N' = (Ms', N')$ 
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  $S: (S_1, N) = DPLL-step\ (Ms, N)$  using  $DPLL-step-obtains$  by  $auto$ 
  { assume  $\neg dpll_W-all-inv\ (toS\ Ms\ N)$ 
    hence  $?case$  using  $step$  by  $auto$ 
  }
  moreover {
    assume  $dpll_W-all-inv\ (toS\ Ms\ N)$ 
    and  $(S_1, N) = (Ms, N)$ 
    hence  $?case$  using  $S\ step$  by  $auto$ 
  }
  moreover
  { assume  $inv: dpll_W-all-inv\ (toS\ Ms\ N)$ 
    assume  $n: (S_1, N) \neq (Ms, N)$ 
    obtain  $S_1'$  where  $SS: (S_1', N) = DPLL-ci\ S_1\ N$  using  $DPLL-ci-obtains$  by  $blast$ 
    moreover have  $DPLL-ci\ Ms\ N = DPLL-ci\ S_1\ N$ 
    proof –
      have ( $case\ (S_1, N)\ of\ (ms, lss) \Rightarrow if\ (ms, lss) = (Ms, N)\ then\ (Ms, N)\ else\ DPLL-ci\ ms\ N$ )
         $= DPLL-ci\ Ms\ N$ 
      using  $S\ DPLL-ci.simps[of\ Ms\ N]$   $calculation\ inv$  by  $presburger$ 
      hence ( $if\ (S_1, N) = (Ms, N)\ then\ (Ms, N)\ else\ DPLL-ci\ S_1\ N = DPLL-ci\ Ms\ N$ )
        by  $fastforce$ 
      thus  $?thesis$ 
      using  $calculation\ n$  by  $presburger$ 
    qed
    moreover
    have  $DPLL-ci\ S_1'\ N = (S_1', N)$  using  $step\ IH[OF\ -\ -\ S\ n\ SS[symmetric]]\ inv$  by  $blast$ 
    ultimately have  $?case$  using  $step$  by  $fastforce$ 
  }
  ultimately show  $?case$  by  $blast$ 
qed

```

lemma $DPLL-part-dpll_W-all-inv-final$:

```

fixes  $M\ Ms':: (int, unit, unit)\ marked-lit\ list$  and
   $N:: int\ literal\ list\ list$ 
assumes  $inv: dpll_W-all-inv\ (Ms, mset\ (map\ mset\ N))$ 
and  $MsN: DPLL-part\ Ms\ N = (Ms', N)$ 
shows  $conclusive-dpll_W-state\ (toS\ Ms'\ N) \wedge dpll_W^{**}\ (toS\ Ms\ N)\ (toS\ Ms'\ N)$ 
proof –
  have  $2: DPLL-ci\ Ms\ N = DPLL-part\ Ms\ N$  using  $inv\ dpll_W-all-inv-implieS-2-eq3-and-dom$  by  $blast$ 
  hence  $star: dpll_W^{**}\ (toS\ Ms\ N)\ (toS\ Ms'\ N)$  unfolding  $MsN$  using  $DPLL-ci-dpll_W-rtranclp$  by
   $blast$ 

```

hence *inv'*: *dpll_W-all-inv* (*toS Ms' N*) **using** *inv rtrancp-dpll_W-all-inv* **by** *blast*
 show *?thesis* **using** *star DPLL-ci-final-state[OF DPLL-ci-no-more-step inv'] 2 unfolding MsN* **by**
blast
qed

Embedding the invariant into the type

Defining the type **typedef** *dpll_W-state* =

{(*M*::(*int*, *unit*, *unit*) *marked-lit list*, *N*::*int literal list list*).
dpll_W-all-inv (*toS M N*)}

morphisms *rough-state-of state-of*

proof

show (\square, \square) $\in \{(M, N). \text{dpll}_W\text{-all-inv } (toS\ M\ N)\}$ **by** (*auto simp add: dpll_W-all-inv-def*)

qed

lemma

DPLL-part-dom (\square, N)

using *assms dpll_W-all-inv-implieS-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-rtrancp case-prodE case-prodI2 rough-state-of mem-Collect-eq old.prod.case prod.sel(2) rtrancp-dpll_W-all-inv snd-DPLL-step*)

lemma *rough-state-of-DPLL-step'-DPLL-step*[*simp*]:

rough-state-of (*DPLL-step'* *S*) = *DPLL-step* (*rough-state-of S*)

using *DPLL-step-dpll_W-conc-inv DPLL-step'-def state-of-inverse* **by** *auto*

function *DPLL-tot*:: *dpll_W-state* \Rightarrow *dpll_W-state* **where**

DPLL-tot S =

(*let S' = DPLL-step' S* *in*

if S' = S *then S* *else DPLL-tot S'*)

by *fast+*

termination

proof (*relation* $\{(T', T).$

(*rough-state-of T'*, *rough-state-of T*)

$\in \{(S', S). (toS'\ S', toS'\ S)$

$\in \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W\ S\ S'\}\}$)

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 dpllW-wf, of toS'], of rough-state-of] .
next
fix S x
assume x: x = DPLL-step' S
and x ≠ S
have dpllW-all-inv (case rough-state-of S of (Ms, N) ⇒ (Ms, mset (map mset N)))
  by (metis (no-types, lifting) case-prodE mem-Collect-eq old.prod.case rough-state-of)
moreover have dpllW (case rough-state-of S of (Ms, N) ⇒ (Ms, mset (map mset N)))
  (case rough-state-of (DPLL-step' S) of (Ms, N) ⇒ (Ms, mset (map mset N)))
proof -
  obtain Ms N where Ms: (Ms, N) = rough-state-of S by (cases rough-state-of S) auto
  have dpllW-all-inv (toS' (Ms, N)) using calculation unfolding Ms by blast
  moreover obtain Ms' N' where Ms': (Ms', N') = rough-state-of (DPLL-step' S)
    by (cases rough-state-of (DPLL-step' S)) auto
  ultimately have dpllW-all-inv (toS' (Ms', N')) unfolding Ms'
    by (metis (no-types, lifting) case-prod-unfold mem-Collect-eq rough-state-of)

  have dpllW (toS Ms N) (toS Ms' N')
    apply (rule DPLL-step-is-a-dpllW-step[of Ms' N' Ms N])
    unfolding Ms Ms' using ⟨x ≠ S⟩ rough-state-of-inject x by fastforce+
    thus ?thesis unfolding Ms[symmetric] Ms'[symmetric] by auto
qed
ultimately show (x, S) ∈ {(T', T). (rough-state-of T', rough-state-of T)}
  ∈ {(S', S). (toS' S', toS' S) ∈ {(S', S). dpllW-all-inv S ∧ dpllW S S'}}}
  by (auto simp add: x)
qed

lemma [code]:
DPLL-tot S =
  (let S' = DPLL-step' S in
  if S' = S then S else DPLL-tot S') by auto

lemma DPLL-tot-DPLL-step-DPLL-tot[simp]: DPLL-tot (DPLL-step' S) = DPLL-tot S
  apply (cases DPLL-step' S = S)
  apply simp
  unfolding DPLL-tot.simps[of S] by (simp del: DPLL-tot.simps)

lemma DOPLL-step'-DPLL-tot[simp]:
DPLL-step' (DPLL-tot S) = DPLL-tot S
  by (rule DPLL-tot.induct[of λS. DPLL-step' (DPLL-tot S) = DPLL-tot S])
  (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.prem 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-W-Implementation.dpll_W-state* **definition** *Con* :: (*int*, *unit*, *unit*) marked-lit
list × *int* literal *list list*

⇒ *dpll_W-state* **where**

Con xs = *state-of* (*if dpll_W-all-inv* (*toS* (*fst xs*) (*snd xs*)) *then xs* *else* ([], []))

lemma [*code abstype*]:

Con (*rough-state-of S*) = *S*

using *rough-state-of[of S]* **unfolding** *Con-def* **by** *auto*

declare *rough-state-of-DPLL-step'-DPLL-step*[*code abstract*]

lemma *Con-DPLL-step-rough-state-of-state-of[simp]*:

$Con (DPLL\text{-}step (rough\text{-}state\text{-}of\ s)) = state\text{-}of (DPLL\text{-}step (rough\text{-}state\text{-}of\ s))$
unfolding $Con\text{-}def$ **by** $(metis (mono\text{-}tags, lifting) DPLL\text{-}step\text{-}dpll_W\text{-}conc\text{-}inv\ mem\text{-}Collect\text{-}eq\ prod.\text{case}\text{-}eq\text{-}if)$

A slightly different version of $DPLL\text{-}tot$ where the returned boolean indicates the result.

definition $DPLL\text{-}tot\text{-}rep$ **where**

$DPLL\text{-}tot\text{-}rep\ S =$
 $(let\ (M, N) = (rough\text{-}state\text{-}of\ (DPLL\text{-}tot\ S))\ in\ (\forall A \in set\ N. (\exists a \in set\ A. a \in lits\text{-}of\ (M)), M))$

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 $Clausal\text{-}Logic$;
- 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 $CDCL\text{-}W\text{-}Implementation$

imports $DPLL\text{-}CDCL\text{-}W\text{-}Implementation\ CDCL\text{-}W\text{-}Termination$

begin

notation $image\text{-}mset$ (**infixr** $'\#$ 90)

type-synonym $'a\ cdcl_W\text{-}mark = 'a\ clause$

type-synonym $cdcl_W\text{-}marked\text{-}level = nat$

type-synonym $'v\ cdcl_W\text{-}marked\text{-}lit = ('v, cdcl_W\text{-}marked\text{-}level, 'v\ cdcl_W\text{-}mark)\ marked\text{-}lit$

type-synonym $'v\ cdcl_W\text{-}marked\text{-}lits = ('v, cdcl_W\text{-}marked\text{-}level, 'v\ cdcl_W\text{-}mark)\ marked\text{-}lits$

type-synonym $'v\ cdcl_W\text{-}state =$

$'v\ cdcl_W\text{-}marked\text{-}lits \times 'v\ clauses \times 'v\ clauses \times nat \times 'v\ clause\ option$

abbreviation $trail :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a$ **where**

$trail \equiv (\lambda(M, -). M)$

abbreviation $cons\text{-}trail :: 'a \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e$

where

$cons\text{-}trail \equiv (\lambda L\ (M, S). (L\#M, S))$

abbreviation $tl\text{-}trail :: 'a\ list \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a\ list \times 'b \times 'c \times 'd \times 'e$ **where**

$tl\text{-}trail \equiv (\lambda(M, S). (tl\ M, S))$

abbreviation $clss :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b$ **where**

$clss \equiv \lambda(M, N, -). N$

abbreviation $learned\text{-}clss :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c$ **where**

$learned\text{-}clss \equiv \lambda(M, N, U, -). U$

abbreviation $backtrack\text{-}lvl :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'd$ **where**

$backtrack\text{-}lvl \equiv \lambda(M, N, U, k, -). k$

abbreviation $update\text{-}backtrack\text{-}lvl :: 'd \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$

where

update-backtrack-lvl $\equiv \lambda k \ (M, N, U, -, S). \ (M, N, U, k, S)$

abbreviation *conflicting* $:: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e$ **where**
conflicting $\equiv \lambda(M, N, U, k, D). \ D$

abbreviation *update-conflicting* $:: 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
update-conflicting $\equiv \lambda S \ (M, N, U, k, -). \ (M, N, U, k, S)$

abbreviation *S0-cdcl_W* $N \equiv (([], N, \{\#\}, 0, None):: 'v \text{ cdcl}_W\text{-state})$

abbreviation *add-learned-cls* **where**
add-learned-cls $\equiv \lambda C \ (M, N, U, S). \ (M, N, \{\#C\# \} + U, S)$

abbreviation *remove-cls* **where**
remove-cls $\equiv \lambda C \ (M, N, U, S). \ (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$

lemma *trail-conv*: *trail* $(M, N, U, k, D) = M$ **and**
clauses-conv: *clss* $(M, N, U, k, D) = N$ **and**
learned-clss-conv: *learned-clss* $(M, N, U, k, D) = U$ **and**
conflicting-conv: *conflicting* $(M, N, U, k, D) = D$ **and**
backtrack-lvl-conv: *backtrack-lvl* $(M, N, U, k, D) = k$
by *auto*

lemma *state-conv*:
 $S = (\text{trail } S, \text{clss } S, \text{learned-clss } S, \text{backtrack-lvl } S, \text{conflicting } S)$
by (*cases* S) *auto*

interpretation *state_W* *trail* *clss* *learned-clss* *backtrack-lvl* *conflicting*
 $\lambda L \ (M, S). \ (L \# M, S)$
 $\lambda(M, S). \ (tl \ M, S)$
 $\lambda C \ (M, N, S). \ (M, \{\#C\# \} + N, S)$
 $\lambda C \ (M, N, U, S). \ (M, N, \{\#C\# \} + U, S)$
 $\lambda C \ (M, N, U, S). \ (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$
 $\lambda(k::nat) \ (M, N, U, -, D). \ (M, N, U, k, D)$
 $\lambda D \ (M, N, U, k, -). \ (M, N, U, k, D)$
 $\lambda N. \ ([], N, \{\#\}, 0, None)$
 $\lambda(-, N, U, -). \ ([], N, U, 0, None)$
by *unfold-locales auto*

interpretation *cdcl_W* *trail* *clss* *learned-clss* *backtrack-lvl* *conflicting*
 $\lambda L \ (M, S). \ (L \# M, S)$
 $\lambda(M, S). \ (tl \ M, S)$
 $\lambda C \ (M, N, S). \ (M, \{\#C\# \} + N, S)$
 $\lambda C \ (M, N, U, S). \ (M, N, \{\#C\# \} + U, S)$
 $\lambda C \ (M, N, U, S). \ (M, \text{remove-mset } C \ N, \text{remove-mset } C \ U, S)$
 $\lambda(k::nat) \ (M, N, U, -, D). \ (M, N, U, k, D)$
 $\lambda D \ (M, N, U, k, -). \ (M, N, U, k, D)$
 $\lambda N. \ ([], N, \{\#\}, 0, None)$
 $\lambda(-, N, U, -). \ ([], N, U, 0, None)$
by *unfold-locales auto*

declare *clauses-def*[*simp*]

lemma *cdcl_W-state-eq-equality*[*iff*]: *state-eq S T* \longleftrightarrow *S = T*
unfolding *state-eq-def* **by** (*cases S, cases T*) *auto*
declare *state-simp*[*simp del*]

18.3 CDCL Implementation

18.3.1 Definition of the rules

Types **lemma** *true-clss-remdups*[*simp*]:
 $I \models_s (mset \circ remdups) \text{ ' } N \longleftrightarrow I \models_s mset \text{ ' } N$
by (*simp add: true-clss-def*)

lemma *satisfiable-mset-remdups*[*simp*]:
 $satisfiable ((mset \circ remdups) \text{ ' } N) \longleftrightarrow satisfiable (mset \text{ ' } N)$
unfolding *satisfiable-carac*[*symmetric*] **by** *simp*

value *backtrack-split* [*Marked (Pos (Suc 0)) ()*]
value $\exists C \in set \ [[Pos (Suc 0), Neg (Suc 0)]]. (\forall c \in set C. -c \in lits-of \ [Marked (Pos (Suc 0)) ()])$

type-synonym *cdcl_W-state-inv-st* = (*nat, nat, nat literal list*) *marked-lit list* \times
nat literal list list \times *nat literal list list* \times *nat* \times *nat literal list option*

We need some functions to convert between our abstract state *nat cdcl_W-state* and the concrete state *cdcl_W-state-inv-st*.

fun *convert* :: (*'a, 'b, 'c list*) *marked-lit* \Rightarrow (*'a, 'b, 'c multiset*) *marked-lit* **where**
convert (Propagated L C) = Propagated L (mset C) |
convert (Marked K i) = Marked K i

abbreviation *convertC* :: *'a list option* \Rightarrow *'a multiset option* **where**
convertC \equiv *map-option mset*

lemma *convert-Propagated*[*elim!*]:
 $convert \ z = Propagated \ L \ C \Longrightarrow (\exists C'. z = Propagated \ L \ C' \wedge C = mset \ C')$
by (*cases z*) *auto*

lemma *get-rev-level-map-convert*:
 $get-rev-level \ (map \ convert \ M) \ n \ x = get-rev-level \ M \ n \ x$
by (*induction M arbitrary: n rule: marked-lit-list-induct*) *auto*

lemma *get-level-map-convert*[*simp*]:
 $get-level \ (map \ convert \ M) = get-level \ M$
using *get-rev-level-map-convert*[*of rev M*] **by** (*simp add: rev-map*)

lemma *get-maximum-level-map-convert*[*simp*]:
 $get-maximum-level \ (map \ convert \ M) \ D = get-maximum-level \ M \ D$
by (*induction D*)
(auto simp add: get-maximum-level-plus)

lemma *get-all-levels-of-marked-map-convert*[*simp*]:
 $get-all-levels-of-marked \ (map \ convert \ M) = (get-all-levels-of-marked \ M)$
by (*induction M rule: marked-lit-list-induct*) *auto*

Conversion function

fun *toS* :: *cdcl_W-state-inv-st* \Rightarrow *nat cdcl_W-state* **where**
toS (M, N, U, k, C) = (map convert M, mset (map mset N), mset (map mset U), k, convertC C)

Definition an abstract type

```

typedef cdclW-state-inv = {S::cdclW-state-inv-st. cdclW-all-struct-inv (toS S)}
morphisms rough-state-of state-of
proof
  show ([],[], [], 0, None) ∈ {S. cdclW-all-struct-inv (toS S)}
  by (auto simp add: cdclW-all-struct-inv-def)
qed

instantiation cdclW-state-inv :: equal
begin
definition equal-cdclW-state-inv :: cdclW-state-inv ⇒ cdclW-state-inv ⇒ bool where
  equal-cdclW-state-inv S S' = (rough-state-of S = rough-state-of S')
instance
  by standard (simp add: rough-state-of-inject equal-cdclW-state-inv-def)
end

```

```

lemma lits-of-map-convert[simp]: lits-of (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 ⟷ undefined-lit M L
by (auto simp add: Marked-Propagated-in-iff-in-lits-of)

```

```

lemma true-annot-map-convert[simp]: map convert M ⊨a N ⟷ M ⊨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 ⊨as N ⟷ M ⊨as N
unfolding true-annots-def by auto

```

lemmas *propagateE*

```

lemma find-first-unit-clause-some-is-propagate:
  assumes H: find-first-unit-clause (N @ U) M = Some (L, C)
  shows propagate (toS (M, N, U, k, None)) (toS (Propagated L C # M, N, U, k, None))
  using assms
  by (auto dest!: find-first-unit-clause-some simp add: propagate.simps
    intro!: exI[of - mset C - {#L#}])

```

18.3.2 The Transitions

Propagate **definition** *do-propagate-step* **where**

```

do-propagate-step S =
  (case S of
    (M, N, U, k, None) ⇒
      (case find-first-unit-clause (N @ U) M of
        Some (L, C) ⇒ (Propagated L C # M, N, U, k, None)
        | None ⇒ (M, N, U, k, None))
    | S ⇒ S)

```

```

lemma do-propagate-step:
  do-propagate-step S ≠ S ⟹ propagate (toS S) (toS (do-propagate-step S))
apply (cases S, cases conflicting S)
using find-first-unit-clause-some-is-propagate[of class S learned-class S trail S - -
  backtrack-lvl S]
by (auto simp add: do-propagate-step-def split: option.splits)

```



```

lemma do-propagate-step-option[simp]:
  conflicting  $S \neq \text{None} \implies \text{do-propagate-step } S = S$ 
  unfolding do-propagate-step-def by (cases  $S$ , cases conflicting  $S$ ) auto

lemma do-propagate-step-no-step:
  assumes dist:  $\forall c \in \text{set } (\text{clss } S @ \text{learned-clss } S). \text{distinct } c$  and
  prop-step:  $\text{do-propagate-step } S = S$ 
  shows no-step propagate (toS  $S$ )
proof (standard, standard)
  fix  $T$ 
  assume propagate (toS  $S$ )  $T$ 
  then obtain  $M\ N\ U\ k\ C\ L$  where
    toSS:  $\text{toS } S = (M, N, U, k, \text{None})$  and
     $T$ :  $T = (\text{Propagated } L\ (C + \{\#L\# \}) \# M, N, U, k, \text{None})$  and
    MC:  $M \models_{\text{as}} C \text{Not } C$  and
    undef: undefined-lit  $M\ L$  and
    CL:  $C + \{\#L\# \} \in \# N + U$ 
  apply – by (cases toS  $S$ ) auto
  let  $?M = \text{trail } S$ 
  let  $?N = \text{clss } S$ 
  let  $?U = \text{learned-clss } S$ 
  let  $?k = \text{backtrack-lvl } S$ 
  let  $?D = \text{None}$ 
  have  $S: S = (?M, ?N, ?U, ?k, ?D)$ 
    using toSS by (cases  $S$ , cases conflicting  $S$ ) simp-all
  have  $S: \text{toS } S = \text{toS } (?M, ?N, ?U, ?k, ?D)$ 
    unfolding  $S[\text{symmetric}]$  by simp

  have
     $M: M = \text{map convert } ?M$  and
     $N: N = \text{mset } (\text{map mset } ?N)$  and
     $U: U = \text{mset } (\text{map mset } ?U)$ 
    using toSS[unfolded  $S$ ] by auto

  obtain  $D$  where
    DCL:  $\text{mset } D = C + \{\#L\# \}$  and
     $D: D \in \text{set } (?N @ ?U)$ 
    using CL unfolding  $N\ U$  by auto
  obtain  $C'\ L'$  where
    setD:  $\text{set } D = \text{set } (L' \# C')$  and
     $C'$ :  $\text{mset } C' = C$  and
     $L: L = L'$ 
    using DCL by (metis ex-mset mset.simps(2) mset-eq-setD)
  have find-first-unit-clause ( $?N @ ?U$ )  $?M \neq \text{None}$ 
    apply (rule dist find-first-unit-clause-none[of  $D\ ?N @ ?U\ ?M\ L$ , OF -  $D$  ])
    using  $D$  assms(1) apply auto[1]
    using  $MC\ \text{setD}\ DCL\ M\ MC$  unfolding  $C'[\text{symmetric}]$  apply auto[1]
    using  $M\ \text{undef}$  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  $M\ [] = \text{None}$  |
  find-conflict  $M\ (N \# Ns) = (\text{if } (\forall c \in \text{set } N. -c \in \text{lits-of } M) \text{ then } \text{Some } N \text{ else } \text{find-conflict } M\ Ns)$ 

```

lemma *find-conflict-Some*:

find-conflict M $Ns = \text{Some } N \implies N \in \text{set } Ns \wedge M \models_{as} CNot \ (mset \ N)$
by (*induction* Ns *rule*: *find-conflict.induct*)
 (*auto split: split-if-asm*)

lemma *find-conflict-None*:

find-conflict M $Ns = \text{None} \longleftrightarrow (\forall N \in \text{set } Ns. \neg M \models_{as} CNot \ (mset \ N))$
by (*induction* Ns) *auto*

lemma *find-conflict-None-no-conf*:

find-conflict M $(N @ U) = \text{None} \longleftrightarrow \text{no-step conflict} \ (toS \ (M, N, U, k, \text{None}))$
by (*auto simp add: find-conflict-None conflict.simps*)

definition *do-conflict-step* **where**

do-conflict-step $S =$
 (*case* S *of*
 $(M, N, U, k, \text{None}) \Rightarrow$
 (*case* *find-conflict* M $(N @ U)$ *of*
 $\text{Some } a \Rightarrow (M, N, U, k, \text{Some } a)$
 $|\ \text{None} \Rightarrow (M, N, U, k, \text{None})$)
 $| \ S \Rightarrow S$)

lemma *do-conflict-step*:

do-conflict-step $S \neq S \implies \text{conflict} \ (toS \ S) \ (toS \ (do-conflict-step \ S))$
apply (*cases* S , *cases conflicting* S)
unfolding *conflict.simps do-conflict-step-def*
by (*auto dest!: find-conflict-Some split: option.splits*)

lemma *do-conflict-step-no-step*:

do-conflict-step $S = S \implies \text{no-step conflict} \ (toS \ S)$
apply (*cases* S , *cases conflicting* S)
unfolding *do-conflict-step-def*
using *find-conflict-None-no-conf*[*of trail* S *class* S *learned-class* S
backtrack-lvl S]
by (*auto split: option.splits*)

lemma *do-conflict-step-option[simp]*:

conflicting $S \neq \text{None} \implies do-conflict-step \ S = S$
unfolding *do-conflict-step-def* **by** (*cases* S , *cases conflicting* S) *auto*

lemma *do-conflict-step-conflicting[dest]*:

do-conflict-step $S \neq S \implies \text{conflicting} \ (do-conflict-step \ S) \neq \text{None}$
unfolding *do-conflict-step-def* **by** (*cases* S , *cases conflicting* S) (*auto split: option.splits*)

definition *do-cp-step* **where**

do-cp-step $S =$
 (*do-propagate-step* o *do-conflict-step*) S

lemma *cp-step-is-cdcl_W-cp*:

assumes H : *do-cp-step* $S \neq S$
shows *cdcl_W-cp* $(toS \ S) \ (toS \ (do-cp-step \ S))$

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*:
 $do\text{-}cp\text{-}step\ S = S \implies do\text{-}conflict\text{-}step\ S = S \wedge do\text{-}propagate\text{-}step\ S = S$
by (cases S, cases conflicting S)
(auto simp add: do-conflict-step-def do-propagate-step-def do-cp-step-def split: option.splits)

lemma *no-cdcl_W-cp-iff-no-propagate-no-conflict*:
 $no\text{-}step\ cdcl_W\text{-}cp\ S \longleftrightarrow no\text{-}step\ propagate\ S \wedge no\text{-}step\ conflict\ S$
by (auto simp: cdcl_W-cp.simps)

lemma *do-cp-step-eq-no-step*:
assumes H: $do\text{-}cp\text{-}step\ S = S$ **and** $\forall c \in set\ (clss\ S @ learned\text{-}clss\ S).$ *distinct c*
shows $no\text{-}step\ cdcl_W\text{-}cp\ (toS\ S)$
unfolding *no-cdcl_W-cp-iff-no-propagate-no-conflict*
using *assms* **apply** (cases S, cases conflicting S)
using *do-propagate-step-no-step*[of S]
by (auto dest!: *do-cp-step-eq-no-prop-no-conf*[simplified] *do-conflict-step-no-step*
split: option.splits)

lemma *cdcl_W-cp-cdcl_W-st*: $cdcl_W\text{-}cp\ S\ S' \implies cdcl_W^{**}\ S\ S'$
by (simp add: cdcl_W-cp-tranclp-cdcl_W tranclp-into-rtranclp)

lemma *cdcl_W-cp-wf-all-inv*:
 $wf\ \{(S', S::'v::linorder\ cdcl_W\text{-}state). cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \wedge cdcl_W\text{-}cp\ S\ S'\}$
(is wf ?R)
proof (rule wf-bounded-measure[of - $\lambda S. card\ (atms\text{-}of\text{-}msu\ (clss\ S)) + 1$
 $\lambda S. length\ (trail\ S) + (if\ conflicting\ S = None\ then\ 0\ else\ 1)$], goal-cases)
case (1 S S')
then have $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ **and** $cdcl_W\text{-}cp\ S\ S'$ **by** auto
moreover then have $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S'$
using *rtranclp-cdcl_W-all-struct-inv-inv cdcl_W-cp-cdcl_W-st* **by** blast
ultimately show ?case

by (auto simp: cdcl_W-cp.simps elim!: conflictE propagateE
 dest: length-model-le-vars-all-inv)
 qed

lemma cdcl_W-all-struct-inv-rough-state[simp]: cdcl_W-all-struct-inv (toS (rough-state-of S))
 using rough-state-of by auto

lemma [simp]: cdcl_W-all-struct-inv (toS S) \implies rough-state-of (state-of S) = S
 by (simp add: state-of-inverse)

lemma rough-state-of-state-of-do-cp-step[simp]:
 rough-state-of (state-of (do-cp-step (rough-state-of S))) = do-cp-step (rough-state-of S)

proof –

have cdcl_W-all-struct-inv (toS (do-cp-step (rough-state-of S)))
 apply (cases do-cp-step (rough-state-of S) = (rough-state-of S))
 apply simp
 using cp-step-is-cdcl_W-cp[of rough-state-of S] cdcl_W-all-struct-inv-rough-state[of S]
 cdcl_W-cp-cdcl_W-st rtranclp-cdcl_W-all-struct-inv-inv by blast
 then show ?thesis by auto

qed

Skip fun do-skip-step :: cdcl_W-state-inv-st \Rightarrow cdcl_W-state-inv-st **where**

do-skip-step (Propagated L C # Ls, N, U, k, Some D) =
 (if $-L \notin \text{set } D \wedge D \neq []$
 then (Ls, N, U, k, Some D)
 else (Propagated L C # Ls, N, U, k, Some D)) |
 do-skip-step S = S

lemma do-skip-step:
 do-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-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-if-asm)

lemma do-skip-step-trail-is-None[iff]:
 do-skip-step S = (a, b, c, d, None) \longleftrightarrow S = (a, b, c, d, None)
 by (cases S rule: do-skip-step.cases) auto

Resolve fun maximum-level-code:: 'a literal list \Rightarrow ('a, nat, 'a literal list) marked-lit list \Rightarrow nat
where
 maximum-level-code [] = 0 |
 maximum-level-code (L # Ls) M = max (get-level M L) (maximum-level-code Ls M)

lemma maximum-level-code-eq-get-maximum-level[code, simp]:
 maximum-level-code D M = get-maximum-level M (mset D)
 by (induction D) (auto simp add: get-maximum-level-plus)

fun do-resolve-step :: cdcl_W-state-inv-st \Rightarrow cdcl_W-state-inv-st **where**
 do-resolve-step (Propagated L C # Ls, N, U, k, Some D) =
 (if $-L \in \text{set } D \wedge \text{maximum-level-code (remove1 } (-L) D) (\text{Propagated L C \# Ls}) = k$
 then (Ls, N, U, k, Some (remdups (remove1 L C @ remove1 (-L) D)))

else (Propagated L C # Ls, N, U, k, Some D)) |
do-resolve-step S = S

lemma do-resolve-step:

cdcl_W-all-struct-inv (toS S) \implies do-resolve-step S \neq S
 \implies resolve (toS S) (toS (do-resolve-step S))

proof (induction S rule: do-resolve-step.induct)

case (1 L C M N U k D)

then have

– L \in set D and

M: maximum-level-code (remove1 (–L) D) (Propagated L C # M) = k

by (cases mset D – {#– L#} = {#},

auto dest!: get-maximum-level-exists-lit-of-max-level[of - Propagated L C # M]

split: split-if-asm)+

have every-mark-is-a-conflict (toS (Propagated L C # M, N, U, k, Some D))

using 1(1) unfolding cdcl_W-all-struct-inv-def cdcl_W-conflicting-def by fast

then have 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 (– L \in set D) by (metis add.commute in-multiset-in-set insert-DiffM)

have D'L: D' + {#– L#} – {#–L#} = D' by (auto simp add: multiset-eq-iff)

have CL: mset C – {#L#} + {#L#} = mset C using (L \in set C) by (auto simp add: multiset-eq-iff)

have get-maximum-level (Propagated L (C' + {#L#}) # map convert M) D' = k

using M[simplified] unfolding maximum-level-code-eq-get-maximum-level C[symmetric] CL

by (metis D D'L convert.simps(1) get-maximum-level-map-convert list.simps(9))

then have

resolve

(map convert (Propagated L C # M), mset '# mset N, mset '# mset U, k, Some (mset D))

(map convert M, mset '# mset N, mset '# mset U, k,

Some (((mset D – {#–L#}) # \cup (mset C – {#L#}))))

unfolding resolve.simps

by (simp add: C D)

moreover have

(map convert (Propagated L C # M), mset '# mset N, mset '# mset U, k, Some (mset D))

= toS (Propagated L C # M, N, U, k, Some D)

by (auto simp: mset-map)

moreover

have distinct-mset (mset C) and distinct-mset (mset D)

using (cdcl_W-all-struct-inv (toS (Propagated L C # M, N, U, k, Some D)))

unfolding cdcl_W-all-struct-inv-def distinct-cdcl_W-state-def

by auto

then have (mset C – {#L#}) # \cup (mset D – {#– L#}) =

remdups-mset (mset C – {#L#} + (mset D – {#– L#}))

by (auto simp: distinct-mset-remdups-union-mset)

then have (map convert M, mset '# mset N, mset '# mset U, k,

Some ((mset D – {#– L#}) # \cup (mset C – {#L#})))

= toS (do-resolve-step (Propagated L C # M, N, U, k, Some D))

using (– L \in set D) M by (auto simp: ac-simps mset-map)

ultimately show ?case

by simp

qed auto

lemma do-resolve-step-no:

```

do-resolve-step S = S  $\implies$  no-step resolve (toS S)
apply (cases S; cases hd (trail S); cases conflicting S)
by (auto
  elim!: resolveE split: split-if-asm
  dest!: union-single-eq-member
  simp del: in-multiset-in-set get-maximum-level-map-convert
  simp: 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)  $\implies$  rough-state-of (state-of (do-resolve-step S)) = do-resolve-step S
apply (rule state-of-inverse)
apply (cases do-resolve-step S = S)
apply simp
by (blast dest: other resolve bj do-resolve-step cdclW-all-struct-inv-inv)

```

```

lemma do-resolve-step-trail-is-None[iff]:
  do-resolve-step S = (a, b, c, d, None)  $\longleftrightarrow$  S = (a, b, c, d, None)
by (cases S rule: do-resolve-step.cases) auto

```

Backjumping **fun** find-level-decomp **where**

```

find-level-decomp M [] D k = None |
find-level-decomp M (L # Ls) D k =
  (case (get-level M L, maximum-level-code (D @ Ls) M) of
    (i, j)  $\Rightarrow$  if i = k  $\wedge$  j < i then Some (L, j) else find-level-decomp M Ls (L # D) k
  )

```

lemma find-level-decomp-some:

```

assumes find-level-decomp M Ls D k = Some (L, j)
shows L  $\in$  set Ls  $\wedge$  get-maximum-level M (mset (remove1 L (Ls @ D))) = j  $\wedge$  get-level M L = k
using assms

```

proof (induction Ls arbitrary: D)

```

case Nil
then show ?case by simp

```

next

```

case (Cons L' Ls) note IH = this(1) and H = this(2)

```

```

def find  $\equiv$  (if get-level M L'  $\neq$  k  $\vee$   $\neg$  get-maximum-level M (mset D + mset Ls) < get-level M L'
  then find-level-decomp M Ls (L' # D) k
  else Some (L', get-maximum-level M (mset D + mset Ls)))

```

```

have a1:  $\bigwedge D$ . find-level-decomp M Ls D k = Some (L, j)  $\implies$ 

```

```

  L  $\in$  set Ls  $\wedge$  get-maximum-level M (mset Ls + mset D - {#L#}) = j  $\wedge$  get-level M L = k

```

```

using IH by simp

```

```

have a2: find = Some (L, j)

```

```

using H unfolding find-def by (auto split: split-if-asm)

```

```

{ assume Some (L', get-maximum-level M (mset D + mset Ls))  $\neq$  find

```

```

then have f3: L  $\in$  set Ls and get-maximum-level M (mset Ls + mset (L' # D) - {#L#}) = j

```

```

using a1 IH a2 unfolding find-def by meson+

```

```

moreover then have mset Ls + mset D - {#L#} + {#L'#} = {#L'#} + mset D + (mset Ls
- {#L#})

```

```

by (auto simp: ac-simps multiset-eq-iff Suc-leI)

```

```

ultimately have f4: get-maximum-level M (mset Ls + mset D - {#L#} + {#L'#}) = j

```

```

by (metis (no-types) diff-union-single-conv mem-set-multiset-eq mset.simps(2) union-commute)

```

```

} note f4 = this

```

```

have {#L'#} + (mset Ls + mset D) = mset Ls + (mset D + {#L'#})

```

```

    by (auto simp: ac-simps)
  then have
    ( $L = L' \longrightarrow \text{get-maximum-level } M \text{ (mset } Ls + \text{mset } D) = j \wedge \text{get-level } M \text{ } L' = k$ ) and
    ( $L \neq L' \longrightarrow L \in \text{set } Ls \wedge \text{get-maximum-level } M \text{ (mset } Ls + \text{mset } D - \{\#L\#\} + \{\#L'\#\}) = j \wedge$ 
       $\text{get-level } M \text{ } L = k$ )
    using f4 a2 a1[of  $L' \# D$ ] unfolding find-def by (metis (no-types) add-diff-cancel-left'
      mset.simps(2) option.inject prod.inject union-commute)+
  then show ?case by simp
qed

```

lemma *find-level-decomp-none*:

```

  assumes find-level-decomp  $M \text{ } Ls \text{ } E \text{ } k = \text{None}$  and  $\text{mset } (L \# D) = \text{mset } (Ls @ E)$ 
  shows  $\neg(L \in \text{set } Ls \wedge \text{get-maximum-level } M \text{ (mset } D) < k \wedge k = \text{get-level } M \text{ } L)$ 
  using assms
proof (induction  $Ls$  arbitrary:  $E \text{ } L \text{ } D$ )
  case Nil
  then show ?case by simp
next
  case (Cons  $L' \text{ } Ls$ ) note IH = this(1) and find-none = this(2) and LD = this(3)
  have  $\text{mset } D + \{\#L'\#\} = \text{mset } E + (\text{mset } Ls + \{\#L'\#\}) \implies \text{mset } D = \text{mset } E + \text{mset } Ls$ 
    by (metis add-right-imp-eq union-assoc)
  then show ?case
    using find-none IH[of  $L' \# E \text{ } L \text{ } D$ ] LD by (auto simp add: ac-simps split: split-if-asm)
qed

```

fun *bt-cut* **where**

```

bt-cut  $i \text{ (Propagated - - } \# Ls) = \text{bt-cut } i \text{ } Ls \mid$ 
bt-cut  $i \text{ (Marked } K \text{ } k \# Ls) = (\text{if } k = \text{Suc } i \text{ then Some (Marked } K \text{ } k \# Ls) \text{ else bt-cut } i \text{ } Ls) \mid$ 
bt-cut  $i \text{ } [] = \text{None}$ 

```

lemma *bt-cut-some-decomp*:

```

   $\text{bt-cut } i \text{ } M = \text{Some } M' \implies \exists K \text{ } M2 \text{ } M1. M = M2 @ M' \wedge M' = \text{Marked } K \text{ (} i+1 \text{)} \# M1$ 
  by (induction  $i \text{ } M$  rule: bt-cut.induct) (auto split: split-if-asm)

```

lemma *bt-cut-not-none*: $M = M2 @ \text{Marked } K \text{ (Suc } i) \# M' \implies \text{bt-cut } i \text{ } M \neq \text{None}$

```

  by (induction  $M2$  arbitrary:  $M$  rule: marked-lit-list-induct) auto

```

lemma *get-all-marked-decomposition-ex*:

```

 $\exists N. (\text{Marked } K \text{ (Suc } i) \# M', N) \in \text{set } (\text{get-all-marked-decomposition } (M2 @ \text{Marked } K \text{ (Suc } i) \# M'))$ 
  apply (induction  $M2$  rule: marked-lit-list-induct)
  apply auto[2]
  by (rename-tac  $L \text{ } m \text{ } xs$ , case-tac get-all-marked-decomposition ( $xs @ \text{Marked } K \text{ (Suc } i) \# M'$ ))
  auto

```

lemma *bt-cut-in-get-all-marked-decomposition*:

```

   $\text{bt-cut } i \text{ } M = \text{Some } M' \implies \exists M2. (M', M2) \in \text{set } (\text{get-all-marked-decomposition } M)$ 
  by (auto dest!: bt-cut-some-decomp simp add: get-all-marked-decomposition-ex)

```

fun *do-backtrack-step* **where**

```

do-backtrack-step ( $M, N, U, k, \text{Some } D$ ) =
  (case find-level-decomp  $M \text{ } D [] \text{ } k$  of
    None  $\Rightarrow (M, N, U, k, \text{Some } D)$ 
  | Some ( $L, j$ )  $\Rightarrow$ 
    (case bt-cut  $j \text{ } M$  of

```

```

    Some (Marked - - # Ls) ⇒ (Propagated L D # Ls, N, D # U, j, None)
  | - ⇒ (M, N, U, k, Some D))
) |
do-backtrack-step S = S

```

lemma *get-all-marked-decomposition-map-convert*:
 (get-all-marked-decomposition (map convert M)) =
 map (λ(a, b). (map convert a, map convert b)) (get-all-marked-decomposition M)
apply (induction M rule: marked-lit-list-induct)
apply simp
by (rename-tac L l xs, case-tac get-all-marked-decomposition xs; auto)+

lemma *do-backtrack-step*:

assumes

db: do-backtrack-step S ≠ 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 = Some C **using** S confl **by** auto

obtain L j **where** fd: find-level-decomp M C [] k = Some (L, j)

using db **unfolding** S E **by** (cases C) (auto split: split-if-asm option.splits)

have L ∈ set C **and** get-maximum-level M (mset (remove1 L C)) = j **and**

levL: get-level M L = k

using find-level-decomp-some[OF fd] **by** auto

obtain C' **where** C: mset C = mset C' + {#L#}

using ⟨L ∈ set C⟩ **by** (metis add.commute ex-mset in-multiset-in-set insert-DiffM)

obtain M₂ **where** M₂: bt-cut j M = Some M₂

using db fd **unfolding** S E **by** (auto split: option.splits)

obtain M1 K **where** M1: M₂ = Marked K (Suc j) # M1

using bt-cut-some-decomp[OF M₂] **by** (cases M₂) auto

obtain c **where** c: M = c @ Marked K (Suc j) # M1

using bt-cut-in-get-all-marked-decomposition[OF M₂]

unfolding M1 **by** fastforce

have get-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-cong[OF this, of λa. Suc j ∈ set a] **have** j ≤ k **unfolding** c **by** auto

have max-l-j: maximum-level-code C' M = j

using db fd M₂ C **unfolding** S E **by** (auto

split: option.splits list.splits marked-lit.splits

dest!: find-level-decomp-some)[1]

have get-maximum-level M (mset C) ≥ k

using ⟨L ∈ set C⟩ get-maximum-level-ge-get-level levL **by** blast

moreover **have** get-maximum-level M (mset C) ≤ k

using get-maximum-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 dest: sym[of get-level - -])

ultimately **have** get-maximum-level M (mset C) = k **by** auto

obtain M2 **where** M2: (M₂, M2) ∈ set (get-all-marked-decomposition M)

using bt-cut-in-get-all-marked-decomposition[OF M₂] **by** metis


```

have H: (reduce-trail-to (map convert M1)
  (add-learned-cls (mset C' + {#L#})
    (map convert M, mset (map mset N), mset (map mset U), j, None))) =
  (map convert M1, mset (map mset N), {#mset C' + {#L#}#} + mset (map mset U), j, None)
  apply (subst state-conv[of reduce-trail-to -])
using M2 unfolding M1 by auto
have
  backtrack
    (map convert M, mset '# mset N, mset '# mset U, k, Some (mset C))
    (Propagated L (mset C) # map convert M1, mset '# mset N, mset '# mset U + {#mset C#},
      j,
        None)
  apply (rule backtrack-rule)
    unfolding C apply simp
    using Set.imageI[of (M2, M2) set (get-all-marked-decomposition M)
      (λ(a, b). (map convert a, map convert b))] M2
    apply (auto simp: get-all-marked-decomposition-map-convert M1)[1]
    using max-l-j levL ⟨j ≤ k⟩ apply (simp add: get-maximum-level-plus)
    using C ⟨get-maximum-level M (mset C) = k⟩ levL apply auto[1]
    using max-l-j apply simp
  apply (cases reduce-trail-to (map convert M1)
    (add-learned-cls (mset C' + {#L#})
      (map convert M, mset (map mset N), mset (map mset U), j, None)))
    using M2 M1 H by (auto simp: ac-simps mset-map)
  then show ?case
    using M2 fd unfolding S E M1 by (auto simp: mset-map)
  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*: *cdcl_W-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 M L = get-maximum-level M (D + {#L#})* **and**

k: *k = get-maximum-level M (D + {#L#})* **and**

j: *j = get-maximum-level M D* **and**

CE: *convertC E = Some (D + {#L#})* **and**

decomp: *(z # M1, b) ∈ set (get-all-marked-decomposition M)* **and**

z: *Marked K (Suc j) = convert z* **using** *bt* **unfolding** *S*

by (auto split: option.splits elim!: *backtrackE*

simp: *get-all-marked-decomposition-map-convert*)

have *z*: *z = Marked K (Suc j)* **using** *z* **by** (cases *z*) *auto*

obtain *c* **where** *c*: *M = c @ b @ Marked K (Suc j) # M1*

using *decomp* **unfolding** *z* **by** *blast*

have *get-all-levels-of-marked (map convert M) = rev [1..<Suc k]*

using *inv* **unfolding** *cdcl_W-all-struct-inv-def cdcl_W-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 = Some C and
C: mset C = mset (L # D')
using CE apply (cases E)
  apply simp
  by (metis ex-mset mset.simps(2) option.inject option.simps(9))
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 M L' ≤ get-maximum-level M D
      using get-maximum-level-ge-get-level by blast
    then show False using ⟨k > j⟩ j find-level-decomp-some[OF fd-some] by auto
  qed
then have j': j' = j using find-level-decomp-some[OF fd-some] j C D'D by auto

have btc-none: bt-cut j M ≠ None
  apply (rule bt-cut-not-none[of M - @ -])
  using c by simp
show ?case using db unfolding S E
  by (auto split: option.splits list.splits marked-lit.splits
    simp add: fd-some L' j' btc-none
    dest: bt-cut-some-decomp)
qed

lemma rough-state-of-state-of-backtrack[simp]:
  assumes inv: cdclW-all-struct-inv (toS S)
  shows rough-state-of (state-of (do-backtrack-step S)) = do-backtrack-step S
proof (rule state-of-inverse)
  have f2: backtrack (toS S) (toS (do-backtrack-step S)) ∨ do-backtrack-step S = S
    using do-backtrack-step inv by blast
  have ∧p. ¬ cdclW-o (toS S) p ∨ cdclW-all-struct-inv p
    using inv cdclW-all-struct-inv-inv other by blast
  then have do-backtrack-step S = S ∨ cdclW-all-struct-inv (toS (do-backtrack-step S))
    using f2 by blast
  then show do-backtrack-step S ∈ {S. cdclW-all-struct-inv (toS S)}
    using inv by fastforce
qed

Decide fun do-decide-step where
do-decide-step (M, N, U, k, None) =
  (case find-first-unused-var N (lits-of M) of
    None ⇒ (M, N, U, k, None)
  | Some L ⇒ (Marked L (Suc k) # M, N, U, k+1, None)) |
do-decide-step S = S

lemma do-decide-step:

```

```

do-decide-step  $S \neq S \implies$  decide (toS  $S$ ) (toS (do-decide-step  $S$ ))
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  $a :: (\text{nat}, \text{nat}, \text{nat literal list}) \text{ marked-lit list and}$ 
     $b :: \text{nat literal list list and } c :: \text{nat literal list list and}$ 
     $d :: \text{nat and } e :: \text{nat literal list option}$ 
  {
    fix  $a :: (\text{nat}, \text{nat}, \text{nat literal list}) \text{ marked-lit list and}$ 
       $b :: \text{nat literal list list and } c :: \text{nat literal list list and}$ 
       $d :: \text{nat and } x2 :: \text{nat literal and } m :: \text{nat literal list}$ 
    assume  $a1: m \in \text{set } b$ 
    assume  $x2 \in \text{set } m$ 
    then have  $f2: \text{atm-of } x2 \in \text{atms-of } (\text{mset } m)$ 
      by simp
    have  $\bigwedge f. (f m :: \text{nat literal multiset}) \in f \text{ ' set } b$ 
      using  $a1$  by blast
    then have  $\bigwedge f. (\text{atms-of } (f m) :: \text{nat set}) \subseteq \text{atms-of-ms } (f \text{ ' set } b)$ 
      using atms-of-atms-of-ms-mono by blast
    then have  $\bigwedge n f. (n :: \text{nat}) \in \text{atms-of-ms } (f \text{ ' set } b) \vee n \notin \text{atms-of } (f m)$ 
      by (meson contra-subsetD)
    then have  $\text{atm-of } x2 \in \text{atms-of-ms } (\text{mset ' set } b)$ 
      using  $f2$  by blast
  } note  $H = \text{this}$ 
  {
    fix  $m :: \text{nat literal list and } x2$ 
    have  $m \in \text{set } b \implies x2 \in \text{set } m \implies x2 \notin \text{lits-of } a \implies \neg x2 \notin \text{lits-of } a \implies$ 
       $\exists aa \in \text{set } b. \neg \text{atm-of ' set } aa \subseteq \text{atm-of ' lits-of } a$ 
      by (meson atm-of-in-atm-of-set-in-uminus contra-subsetD rev-image-eqI)
  } note  $H' = \text{this}$ 

  assume do-decide-step  $S \neq S$  and
     $S = (a, b, c, d, e)$  and
    conflicting  $S = \text{None}$ 
  then show decide (toS  $S$ ) (toS (do-decide-step  $S$ ))
    using  $H H'$  by (auto split: option.splits simp: decide.simps Marked-Propagated-in-iff-in-lits-of
      dest!: find-first-unused-var-Some)
qed

lemma do-decide-step-no:
  do-decide-step  $S = S \implies$  no-step decide (toS  $S$ )
by (cases  $S$ , cases conflicting  $S$ )
  (fastforce simp: atms-of-ms-mset-unfold atm-of-eq-atm-of Marked-Propagated-in-iff-in-lits-of
    split: option.splits)+

lemma rough-state-of-state-of-do-decide-step[simp]:
   $\text{cdcl}_W\text{-all-struct-inv } (\text{toS } S) \implies \text{rough-state-of } (\text{state-of } (\text{do-decide-step } S)) = \text{do-decide-step } S$ 
proof (subst state-of-inverse, goal-cases)
  case 1
  then show ?case
    by (cases do-decide-step  $S = S$ )
    (auto dest: do-decide-step decide other intro: cdclW-all-struct-inv-inv)

```

qed simp

lemma *rough-state-of-state-of-do-skip-step*[simp]:
 $cdcl_W\text{-all-struct-inv } (toS\ S) \implies \text{rough-state-of } (state\text{-of } (do\text{-skip-step } S)) = do\text{-skip-step } S$
apply (*subst state-of-inverse, cases do-skip-step S = S*)
apply simp
by (*blast dest: other skip bj do-skip-step cdcl_W-all-struct-inv-inv*)+

18.3.3 Code generation

Type definition There are two invariants: one while applying conflict and propagate and one for the other rules

declare *rough-state-of-inverse*[simp add]

definition *Con* **where**

Con xs = state-of (if cdcl_W-all-struct-inv (toS (fst xs, snd xs)) then xs
else ([], [], [], 0, None))

lemma [*code abstype*]:

Con (rough-state-of S) = S

using *rough-state-of*[*of S*] **unfolding** *Con-def* **by** simp

definition *do-cp-step'* **where**

do-cp-step' S = state-of (do-cp-step (rough-state-of S))

typedef *cdcl_W-state-inv-from-init-state* = $\{S::cdcl_W\text{-state-inv-st. } cdcl_W\text{-all-struct-inv } (toS\ S)$
 $\wedge cdcl_W\text{-stgy}^{**} (S0\text{-}cdcl_W (clss (toS\ S))) (toS\ S)\}$

morphisms *rough-state-from-init-state-of state-from-init-state-of*

proof

show ($[], [], [], 0, None$) $\in \{S. cdcl_W\text{-all-struct-inv } (toS\ S)$
 $\wedge cdcl_W\text{-stgy}^{**} (S0\text{-}cdcl_W (clss (toS\ S))) (toS\ S)\}$
by (*auto simp add: cdcl_W-all-struct-inv-def*)

qed

instantiation *cdcl_W-state-inv-from-init-state* :: *equal*

begin

definition *equal-cdcl_W-state-inv-from-init-state* :: *cdcl_W-state-inv-from-init-state* \Rightarrow

cdcl_W-state-inv-from-init-state \Rightarrow *bool* **where**

equal-cdcl_W-state-inv-from-init-state S S' \longleftrightarrow

(rough-state-from-init-state-of S = rough-state-from-init-state-of S')

instance

by *standard (simp add: rough-state-from-init-state-of-inject*

equal-cdcl_W-state-inv-from-init-state-def)

end

definition *ConI* **where**

ConI S = state-from-init-state-of (if cdcl_W-all-struct-inv (toS (fst S, snd S))
 $\wedge cdcl_W\text{-stgy}^{**} (S0\text{-}cdcl_W (clss (toS\ S))) (toS\ S)$ *then S else ([], [], [], 0, None))*

lemma [*code abstype*]:

ConI (rough-state-from-init-state-of S) = S

using *rough-state-from-init-state-of*[*of S*] **unfolding** *ConI-def*

by (*simp add: rough-state-from-init-state-of-inverse*)

definition *id-of-I-to::* *cdcl_W-state-inv-from-init-state* \Rightarrow *cdcl_W-state-inv* **where**

id-of-I-to S = state-of (rough-state-from-init-state-of S)

lemma *[code abstract]*:
rough-state-of (id-of-I-to S) = rough-state-from-init-state-of S
unfolding *id-of-I-to-def* **using** *rough-state-from-init-state-of* **by** *auto*

Conflict and Propagate **function** *do-full1-cp-step* :: *cdcl_W-state-inv* \Rightarrow *cdcl_W-state-inv* **where**
do-full1-cp-step S =

(let S' = do-cp-step' S in
if S = S' then S else do-full1-cp-step S')

by *auto*

termination

proof *(relation {(T', T). (rough-state-of T', rough-state-of T) \in {(S', S). (toS S', toS S) \in {(S', S). cdcl_W-all-struct-inv S \wedge cdcl_W-cp S S'}}}, goal-cases)*

case *1*

show *?case*

using *wf-if-measure-f[OF wf-if-measure-f[OF cdcl_W-cp-wf-all-inv, of toS], of rough-state-of]* .

next

case *(2 S' S)*

then show *?case*

unfolding *do-cp-step'-def*

apply *simp*

by *(metis cp-step-is-cdcl_W-cp rough-state-of-inverse)*

qed

lemma *do-full1-cp-step-fix-point-of-do-full1-cp-step*:

do-cp-step(rough-state-of (do-full1-cp-step S)) = (rough-state-of (do-full1-cp-step S))

by *(rule do-full1-cp-step.induct[of $\lambda S. do-cp-step(rough-state-of (do-full1-cp-step S))$]
 $= (rough-state-of (do-full1-cp-step S))$)*

(metis (full-types) do-full1-cp-step.elims rough-state-of-state-of-do-cp-step do-cp-step'-def)

lemma *in-clauses-rough-state-of-is-distinct*:

c \in set (clss (rough-state-of S) @ learned-clss (rough-state-of S)) \implies distinct c

apply *(cases rough-state-of S)*

using *rough-state-of[of S] by (auto simp add: distinct-mset-set-distinct cdcl_W-all-struct-inv-def
distinct-cdcl_W-state-def)*

lemma *do-full1-cp-step-full*:

full cdcl_W-cp (toS (rough-state-of S))

(toS (rough-state-of (do-full1-cp-step S)))

unfolding *full-def*

proof *(rule conjI, induction S rule: do-full1-cp-step.induct)*

case *(1 S)*

then have *f1*:

*cdcl_W-cp** (toS (do-cp-step (rough-state-of S))) (
toS (rough-state-of (do-full1-cp-step (state-of (do-cp-step (rough-state-of S))))))
 \vee state-of (do-cp-step (rough-state-of S)) = S*

using *do-cp-step'-def rough-state-of-state-of-do-cp-step by fastforce*

have *f2*: $\bigwedge c. (if c = state-of (do-cp-step (rough-state-of c))$
 $then c else do-full1-cp-step (state-of (do-cp-step (rough-state-of c))))$
 $= do-full1-cp-step c$

by *(metis (full-types) do-cp-step'-def do-full1-cp-step.simps)*

have *f3*: $\neg cdcl_W-cp (toS (rough-state-of S)) (toS (do-cp-step (rough-state-of S)))$

$\vee state-of (do-cp-step (rough-state-of S)) = S$

$\vee cdcl_W-cp^{++} (toS (rough-state-of S))$

$(toS (rough-state-of (do-full1-cp-step (state-of (do-cp-step (rough-state-of S))))))$

```

  using f1 by (meson rtrancp-into-trancp2)
{ assume do-full1-cp-step S ≠ S
  then have do-cp-step (rough-state-of S) = rough-state-of S
    → cdclW-cp** (toS (rough-state-of S)) (toS (rough-state-of (do-full1-cp-step S)))
    ∨ do-cp-step (rough-state-of S) ≠ rough-state-of S
    ∧ state-of (do-cp-step (rough-state-of S)) ≠ S
  using f2 f1 by (metis (no-types))
  then have do-cp-step (rough-state-of S) ≠ rough-state-of S
    ∧ state-of (do-cp-step (rough-state-of S)) ≠ S
    ∨ cdclW-cp** (toS (rough-state-of S)) (toS (rough-state-of (do-full1-cp-step S)))
  by (metis rough-state-of-state-of-do-cp-step)
  then have cdclW-cp** (toS (rough-state-of S)) (toS (rough-state-of (do-full1-cp-step S)))
    using f3 f2 by (metis (no-types) cp-step-is-cdclW-cp trancp-into-rtrancp) }
then show ?case
  by fastforce
next
show no-step cdclW-cp (toS (rough-state-of (do-full1-cp-step S)))
  apply (rule do-cp-step-eq-no-step[OF do-full1-cp-step-fix-point-of-do-full1-cp-step[of S]])
  using in-clauses-rough-state-of-is-distinct unfolding do-cp-step'-def by blast
qed

```

lemma [code abstract]:
 $\text{rough-state-of (do-cp-step' } S) = \text{do-cp-step (rough-state-of } S)$
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 ≠ S
    then T
    else
      (let U = do-resolve-step T in
        if U ≠ T
        then U else
          (let V = do-backtrack-step U in
            if V ≠ U then V else do-decide-step V)))

```

lemma do-other-step:
assumes inv: cdcl_W-all-struct-inv (toS S) **and**
 st: do-other-step S ≠ 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]:
 $\text{rough-state-of (state-of (do-other-step (rough-state-of } S))) = \text{do-other-step (rough-state-of } S)$

proof (*cases do-other-step (rough-state-of S) = rough-state-of S*)
 case *True*
 then show ?thesis **by** *simp*
next
 case *False*
 have *cdcl_W-o (toS (rough-state-of S)) (toS (do-other-step (rough-state-of S)))*
 by (*metis False cdcl_W-all-struct-inv-rough-state do-other-step[of rough-state-of S]*)
 then have *cdcl_W-all-struct-inv (toS (do-other-step (rough-state-of S)))*
 using *cdcl_W-all-struct-inv-inv cdcl_W-all-struct-inv-rough-state other* **by** *blast*
 then show ?thesis
 by (*simp add: CollectI state-of-inverse*)
qed

definition *do-other-step'* **where**

do-other-step' S =
state-of (do-other-step (rough-state-of S))

lemma *rough-state-of-do-other-step'[code abstract]:*

rough-state-of (do-other-step' S) = do-other-step (rough-state-of S)

apply (*cases do-other-step (rough-state-of S) = rough-state-of S*)

unfolding *do-other-step'-def* **apply** *simp*

using *do-other-step[of rough-state-of S]* **by** (*auto intro: cdcl_W-all-struct-inv-inv*
cdcl_W-all-struct-inv-rough-state other state-of-inverse)

definition *do-cdcl_W-stgy-step* **where**

do-cdcl_W-stgy-step S =
(let T = do-full1-cp-step S in
if T ≠ S
then T
else
(let U = (do-other-step' T) in
(do-full1-cp-step U)))

definition *do-cdcl_W-stgy-step'* **where**

do-cdcl_W-stgy-step' S = state-from-init-state-of (rough-state-of (do-cdcl_W-stgy-step (id-of-I-to S)))

lemma *toS-do-full1-cp-step-not-eq: do-full1-cp-step S ≠ S ⇒*

toS (rough-state-of S) ≠ toS (rough-state-of (do-full1-cp-step S))

proof –

assume *a1: do-full1-cp-step S ≠ S*

then have *S ≠ do-cp-step' S*

by *fastforce*

then show ?thesis

by (*metis (no-types) cp-step-is-cdcl_W-cp do-cp-step'-def do-cp-step-eq-no-step*
do-full1-cp-step-fix-point-of-do-full1-cp-step in-clauses-rough-state-of-is-distinct
rough-state-of-inverse)

qed

do-full1-cp-step should not be unfolded anymore:

declare *do-full1-cp-step.simps[simp del]*

Correction of the transformation **lemma** *do-cdcl_W-stgy-step:*

assumes *do-cdcl_W-stgy-step S ≠ S*

shows *cdcl_W-stgy (toS (rough-state-of S)) (toS (rough-state-of (do-cdcl_W-stgy-step S)))*

proof (*cases do-full1-cp-step S = S*)

```

case False
then show ?thesis
  using assms do-full1-cp-step-full[of S] unfolding full-unfold do-cdclW-stgy-step-def
  by (auto intro!: cdclW-stgy.intros dest: toS-do-full1-cp-step-not-eq)
next
case True
have cdclW-o (toS (rough-state-of S)) (toS (rough-state-of (do-other-step' S)))
  by (smt True assms cdclW-all-struct-inv-rough-state do-cdclW-stgy-step-def do-other-step
    rough-state-of-do-other-step' rough-state-of-inverse)
moreover
  have
    np: no-step propagate (toS (rough-state-of S)) and
    nc: no-step conflict (toS (rough-state-of S))
    apply (metis True do-cp-step-eq-no-prop-no-conf
      do-full1-cp-step-fix-point-of-do-full1-cp-step do-propagate-step-no-step
      in-clauses-rough-state-of-is-distinct)
    by (metis True do-conflict-step-no-step do-cp-step-eq-no-prop-no-conf
      do-full1-cp-step-fix-point-of-do-full1-cp-step)
  then have no-step cdclW-cp (toS (rough-state-of S))
    by (simp add: cdclW-cp.simps)
  moreover have full cdclW-cp (toS (rough-state-of (do-other-step' S)))
    (toS (rough-state-of (do-full1-cp-step (do-other-step' S))))
  using do-full1-cp-step-full by auto
ultimately show ?thesis
  using assms True unfolding do-cdclW-stgy-step-def
  by (auto intro!: cdclW-stgy.other' dest: toS-do-full1-cp-step-not-eq)
qed

```

```

lemma length-trail-toS[simp]:
  length (trail (toS S)) = length (trail S)
  by (cases S) auto

```

```

lemma conflicting-noTrue-iff-toS[simp]:
  conflicting (toS S) ≠ None ⟷ conflicting S ≠ None
  by (cases S) auto

```

```

lemma trail-toS-neq-imp-trail-neq:
  trail (toS S) ≠ trail (toS S') ⟹ trail S ≠ trail S'
  by (cases S, cases S') auto

```

```

lemma do-skip-step-trail-changed-or-conflict:
  assumes d: do-other-step S ≠ S
  and inv: cdclW-all-struct-inv (toS S)
  shows trail S ≠ trail (do-other-step S)

```

proof –

```

have M: ∧M K M1 c. M = c @ K # M1 ⟹ Suc (length M1) ≤ length M
  by auto
have cdclW-M-level-inv (toS S)
  using inv unfolding cdclW-all-struct-inv-def by auto
have cdclW-o (toS S) (toS (do-other-step S)) using do-other-step[OF inv d] .
then show ?thesis
  using ⟨cdclW-M-level-inv (toS S)⟩
  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#}))
  (reduce-trail-to M1
    (add-learned-cls (D + {#L#})
      (update-backtrack-lvl (get-maximum-level (trail (toS S)) D)
        (update-conflicting None (toS S))))))
=
(Propagated L (D + {#L#})# M1, mset (map mset (cls S)),
  {#D + {#L#}#} + mset (map mset (learned-cls S)),
  get-maximum-level (trail (toS S)) D, None)
apply (subst state-conv[of cons-trail - -])
using decomp undef by (cases S) auto
then show ?case
  apply (cases do-other-step S)
  apply (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 simp: Let-def)
  using d apply (cases S rule: do-decide-step.cases; auto split: option.splits)[]
done
qed
qed

```

lemma *do-full1-cp-step-induct*:

$(\bigwedge S. (S \neq \text{do-cp-step}' S \implies P (\text{do-cp-step}' S)) \implies P S) \implies P a0$
 using *do-full1-cp-step.induct* by metis

lemma *do-cp-step-neq-trail-increase*:

$\exists c. \text{trail} (\text{do-cp-step } S) = c @ \text{trail } S \wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$
 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 (rename-tac S, 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*) \neq *None* \implies *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*) \neq *None* \implies *do-full1-cp-step* *S* = *S*
unfolding *do-cp-step'-def* *do-cp-step-def*
apply (*induction rule*: *do-full1-cp-step-induct*)
by (*rename-tac* *S*, *case-tac* *S* \neq *do-cp-step'* *S*)
(auto simp add: do-full1-cp-step.simps do-cp-step-conflicting)

lemma *do-decide-step-not-conflicting-one-more-decide*:

assumes
conflicting *S* = *None* **and**
do-decide-step *S* \neq *S*
shows *Suc* (*length* (*filter is-marked* (*trail* *S*)))
= *length* (*filter is-marked* (*trail* (*do-decide-step* *S*)))
using *assms* **unfolding** *do-other-step'-def*
by (*cases* *S*) (*auto simp: Let-def split: split-if-asm option.splits*
dest! find-first-unused-var-Some-not-all-incl)

lemma *do-decide-step-not-conflicting-one-more-decide-bt*:

assumes *conflicting* *S* \neq *None* **and**
do-decide-step *S* \neq *S*
shows *length* (*filter is-marked* (*trail* *S*)) < *length* (*filter is-marked* (*trail* (*do-decide-step* *S*)))
using *assms* **unfolding** *do-other-step'-def* **by** (*cases* *S*, *cases conflicting* *S*)
(auto simp add: Let-def split: split-if-asm option.splits)

lemma *do-other-step-not-conflicting-one-more-decide-bt*:

assumes
conflicting (*rough-state-of* *S*) \neq *None* **and**
conflicting (*rough-state-of* (*do-other-step'* *S*)) = *None* **and**
do-other-step' *S* \neq *S*
shows *length* (*filter is-marked* (*trail* (*rough-state-of* *S*)))
> *length* (*filter is-marked* (*trail* (*rough-state-of* (*do-other-step'* *S*))))

proof (*cases* *S*, *goal-cases*)

case (*1 y*) **note** *S* = *this*(*1*) **and** *inv* = *this*(*2*)

obtain *M N U k E* **where** *y*: *y* = (*M*, *N*, *U*, *k*, *Some* *E*)

using *assms*(*1*) *S* *inv* **by** (*cases* *y*, *cases conflicting* *y*) *auto*

have *M*: *rough-state-of* (*state-of* (*M*, *N*, *U*, *k*, *Some* *E*)) = (*M*, *N*, *U*, *k*, *Some* *E*)

using *inv y* **by** (*auto simp add: state-of-inverse*)

have *bt*: *do-other-step'* *S* = *state-of* (*do-backtrack-step* (*rough-state-of* *S*))

proof (*cases rough-state-of S rule: do-decide-step.cases*)

case *1*

then show *?thesis*

using *assms*(*1,2*) **by** *auto*[]

next

case (*2 v vb vd vf vh*)

have *f3*: $\bigwedge c$. (*if* *do-skip-step* (*rough-state-of* *c*) \neq *rough-state-of* *c*
then *do-skip-step* (*rough-state-of* *c*)

else *if* *do-resolve-step* (*do-skip-step* (*rough-state-of* *c*)) \neq *do-skip-step* (*rough-state-of* *c*)
then *do-resolve-step* (*do-skip-step* (*rough-state-of* *c*))

else *if* *do-backtrack-step* (*do-resolve-step* (*do-skip-step* (*rough-state-of* *c*)))
 \neq *do-resolve-step* (*do-skip-step* (*rough-state-of* *c*))

```

    then do-backtrack-step (do-resolve-step (do-skip-step (rough-state-of c)))
    else do-decide-step (do-backtrack-step (do-resolve-step
      (do-skip-step (rough-state-of c))))))
  = rough-state-of (do-other-step' c)
  by (simp add: rough-state-of-do-other-step')
  have (trail (rough-state-of (do-other-step' S)), clss (rough-state-of (do-other-step' S)),
    learned-clss (rough-state-of (do-other-step' S)),
    backtrack-lvl (rough-state-of (do-other-step' S)), None)
  = rough-state-of (do-other-step' S)
  using assms(2) by (metis (no-types) state-conv)
  then show ?thesis
    using f3 2 by (metis (no-types) do-decide-step.simps(2) do-resolve-step-trail-is-None
      do-skip-step-trail-is-None rough-state-of-inverse)
  qed
show ?case
  using assms(2) S unfolding bt y inv
  apply simp
  by (auto simp add: M bt-cut-not-none
    split: option.splits
    dest!: bt-cut-some-decomp)
qed

lemma do-other-step-not-conflicting-one-more-decide:
  assumes conflicting (rough-state-of S) = None and
  do-other-step' S ≠ S
  shows 1 + 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 where y: y = (M, N, U, k, None) using assms(1) S inv by (cases y) auto
  have M: rough-state-of (state-of (M, N, U, k, None)) = (M, N, U, k, None)
    using inv y by (auto simp add: state-of-inverse)
  have state-of (do-decide-step (M, N, U, k, None)) ≠ state-of (M, N, U, k, None)
    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) = None ⟷ conflicting S = None
  by (cases S rule: do-resolve-step.cases)
    (auto simp add: Let-def split: option.splits)

```

lemma *conflicting-do-skip-step-iff*[iff]:
 $\text{conflicting } (\text{do-skip-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
by (cases S rule: *do-skip-step.cases*)
(auto simp add: *Let-def split: option.splits*)

lemma *conflicting-do-decide-step-iff*[iff]:
 $\text{conflicting } (\text{do-decide-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
by (cases S rule: *do-decide-step.cases*)
(auto simp add: *Let-def split: option.splits*)

lemma *conflicting-do-backtrack-step-imp*[simp]:
 $\text{do-backtrack-step } S \neq S \implies \text{conflicting } (\text{do-backtrack-step } S) = \text{None}$
by (cases S rule: *do-backtrack-step.cases*)
(auto simp add: *Let-def split: list.splits option.splits marked-lit.splits*)

lemma *do-skip-step-eq-iff-trail-eq*:
 $\text{do-skip-step } S = S \longleftrightarrow \text{trail } (\text{do-skip-step } S) = \text{trail } S$
by (cases S rule: *do-skip-step.cases*) auto

lemma *do-decide-step-eq-iff-trail-eq*:
 $\text{do-decide-step } S = S \longleftrightarrow \text{trail } (\text{do-decide-step } S) = \text{trail } S$
by (cases S rule: *do-decide-step.cases*) (auto split: *option.split*)

lemma *do-backtrack-step-eq-iff-trail-eq*:
 $\text{do-backtrack-step } S = S \longleftrightarrow \text{trail } (\text{do-backtrack-step } S) = \text{trail } S$
by (cases S rule: *do-backtrack-step.cases*)
(auto split: *option.split list.splits marked-lit.splits*
dest!: *bt-cut-in-get-all-marked-decomposition*)

lemma *do-resolve-step-eq-iff-trail-eq*:
 $\text{do-resolve-step } S = S \longleftrightarrow \text{trail } (\text{do-resolve-step } S) = \text{trail } S$
by (cases S rule: *do-resolve-step.cases*) auto

lemma *do-other-step-eq-iff-trail-eq*:
 $\text{trail } (\text{do-other-step } S) = \text{trail } S \longleftrightarrow \text{do-other-step } S = S$
by (auto simp add: *Let-def do-skip-step-eq-iff-trail-eq[symmetric]*
do-decide-step-eq-iff-trail-eq[symmetric] *do-backtrack-step-eq-iff-trail-eq[symmetric]*
do-resolve-step-eq-iff-trail-eq[symmetric])

lemma *do-full1-cp-step-do-other-step'-normal-form*[dest!]:
assumes H : $\text{do-full1-cp-step } (\text{do-other-step}' S) = S$
shows $\text{do-other-step}' S = S \wedge \text{do-full1-cp-step } S = S$
proof –
let $?T = \text{do-other-step}' S$
{ **assume** *confl*: $\text{conflicting } (\text{rough-state-of } ?T) \neq \text{None}$
then have tr : $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } ?T)) = \text{trail } (\text{rough-state-of } ?T)$
using *do-full1-cp-step-conflicting* **by** auto
have $\text{trail } (\text{rough-state-of } (\text{do-full1-cp-step } (\text{do-other-step}' S))) = \text{trail } (\text{rough-state-of } S)$
using *arg-cong*[OF H , of $\lambda S. \text{trail } (\text{rough-state-of } S)$] .
then have $\text{trail } (\text{rough-state-of } (\text{do-other-step}' S)) = \text{trail } (\text{rough-state-of } S)$
by (auto simp add: *do-full1-cp-step-conflicting confl*)
then have $\text{do-other-step}' S = S$
by (simp add: *do-other-step-eq-iff-trail-eq do-other-step'-def*)

```

    del: do-other-step.simps)
  }
  moreover {
    assume eq[simp]: do-other-step' S = S
    obtain c where c: trail (rough-state-of (do-full1-cp-step S)) = c @ trail (rough-state-of S)
    using do-full1-cp-step-neq-trail-increase by auto

    moreover have trail (rough-state-of (do-full1-cp-step S)) = trail (rough-state-of S)
      using arg-cong[OF H, of  $\lambda S. \text{trail (rough-state-of S)}$ ] by simp
    finally have c = [] by blast
    then have do-full1-cp-step S = S using assms by auto
  }
  moreover {
    assume confl: conflicting (rough-state-of ?T) = None and neq: do-other-step' S  $\neq$  S
    obtain c where
      c: trail (rough-state-of (do-full1-cp-step ?T)) = c @ trail (rough-state-of ?T) and
      nm:  $\forall m \in \text{set } c. \neg \text{is-marked } m$ 
    using do-full1-cp-step-neq-trail-increase by auto
    have length (filter is-marked (trail (rough-state-of (do-full1-cp-step ?T))))
      = length (filter is-marked (trail (rough-state-of ?T))) using nm unfolding c by force
    moreover have length (filter is-marked (trail (rough-state-of S)))
       $\neq$  length (filter is-marked (trail (rough-state-of ?T)))
    using do-other-step-not-conflicting-one-more-decide[OF - neq]
      do-other-step-not-conflicting-one-more-decide-bt[of S, OF - confl neq]
      by linarith
    finally have False unfolding H by blast
  }
  ultimately show ?thesis by blast
qed

lemma do-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 *cdcl_W-cp-is-rtrancp-cdcl_W*: *cdcl_W-cp S T \implies cdcl_W^{**} S T*
apply (induction rule: *cdcl_W-cp.induct*)
using *conflict apply blast*
using *propagate by blast*

lemma *rtrancp-cdcl_W-cp-is-rtrancp-cdcl_W*: *cdcl_W-cp^{**} S T \implies cdcl_W^{**} S T*
apply (induction rule: *rtrancp-induct*)
apply *simp*
by (fastforce dest!: *cdcl_W-cp-is-rtrancp-cdcl_W*)

lemma *cdcl_W-stgy-is-rtrancp-cdcl_W*:
*cdcl_W-stgy S T \implies cdcl_W^{**} S T*
apply (induction rule: *cdcl_W-stgy.induct*)
using *cdcl_W-stgy.conflict' rtrancp-cdcl_W-stgy-rtrancp-cdcl_W apply blast*
unfolding *full-def* **by** (fastforce dest!: *other rtrancp-cdcl_W-cp-is-rtrancp-cdcl_W*)

lemma *cdcl_W-stgy-init-clss*: *cdcl_W-stgy S T \implies cdcl_W-M-level-inv S \implies clss S = clss T*
using *rtrancp-cdcl_W-init-clss cdcl_W-stgy-is-rtrancp-cdcl_W* **by** *fast*

lemma *clauses-toS-rough-state-of-do-cdcl_W-stgy-step*[simp]:
clss (toS (rough-state-of (do-cdcl_W-stgy-step (state-of (rough-state-from-init-state-of S))))))
 = *clss (toS (rough-state-from-init-state-of S)) (is - = clss (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 (clss (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).
  (cdclW-measure (toS (rough-state-from-init-state-of T)),
  cdclW-measure (toS (rough-state-from-init-state-of S)))
  ∈ lexn {(a, b). a < b} 3}, goal-cases)
case 1
show ?case by (rule wf-if-measure-f) (auto intro!: wf-lexn wf-less)
next
case (2 S T) note T = this(1) and ST = this(2)
let ?S = rough-state-from-init-state-of S
have S: cdclW-stgy** (S0-cdclW (clss (toS ?S))) (toS ?S)
  using rough-state-from-init-state-of[of S] by auto
moreover have cdclW-stgy (toS (rough-state-from-init-state-of S))
  (toS (rough-state-from-init-state-of T))
proof -
  have ∧c. rough-state-of (state-of (rough-state-from-init-state-of c)) =
    rough-state-from-init-state-of c
  using rough-state-from-init-state-of by force
  then have do-cdclW-stgy-step (state-of (rough-state-from-init-state-of S))
    ≠ state-of (rough-state-from-init-state-of S)
  using ST T by (metis (no-types) id-of-I-to-def rough-state-from-init-state-of-inject
    rough-state-from-init-state-of-do-cdclW-stgy-step')
  then show ?thesis
    using do-cdclW-stgy-step id-of-I-to-def rough-state-from-init-state-of-do-cdclW-stgy-step' T
    by fastforce
qed
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 (clss (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 (clss (toS ?S))]
    simp del: cdclW-measure.simps)
qed

thm do-all-cdclW-stgy.induct
lemma do-all-cdclW-stgy.induct:
  (∧S. (do-cdclW-stgy-step' S ≠ S ⇒ P (do-cdclW-stgy-step' S)) ⇒ P S) ⇒ 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 (rename-tac S, case-tac do-cdclW-stgy-step' S ≠ S)
proof -
  fix Sa :: cdclW-state-inv-from-init-state
  assume a1: ¬ do-cdclW-stgy-step' Sa ≠ Sa
  { fix pp
    have (if True then Sa else do-all-cdclW-stgy Sa) = do-all-cdclW-stgy Sa
      using a1 by auto
    then have ¬ cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa))) pp
      using a1 by (metis (no-types) do-cdclW-stgy-step-no id-of-I-to-def

```

```

    rough-state-from-init-state-of-do-cdclW-stgy-step' rough-state-of-inverse) }
  then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa)))
    by fastforce
next
fix Sa :: cdclW-state-inv-from-init-state
assume a1: do-cdclW-stgy-step' Sa ≠ Sa
  ⇒ no-step cdclW-stgy (toS (rough-state-from-init-state-of
    (do-all-cdclW-stgy (do-cdclW-stgy-step' Sa))))
assume a2: do-cdclW-stgy-step' Sa ≠ Sa
have do-all-cdclW-stgy Sa = do-all-cdclW-stgy (do-cdclW-stgy-step' Sa)
  by (metis (full-types) do-all-cdclW-stgy.simps)
then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa)))
  using a2 a1 by presburger
qed

```

```

lemma do-all-cdclW-stgy-is-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)))
proof (induction S rule: do-all-cdclW-stgy-induct)
case (1 S) note IH = this(1)
show ?case
proof (cases do-cdclW-stgy-step' S = S)
case True
  then show ?thesis by simp
next
case False
  have f2: do-cdclW-stgy-step (id-of-I-to S) = id-of-I-to S ⟶
    rough-state-from-init-state-of (do-cdclW-stgy-step' S)
    = rough-state-of (state-of (rough-state-from-init-state-of S))
    using id-of-I-to-def rough-state-from-init-state-of-do-cdclW-stgy-step' by presburger
  have f3: do-all-cdclW-stgy S = do-all-cdclW-stgy (do-cdclW-stgy-step' S)
    by (metis (full-types) do-all-cdclW-stgy.simps)
  have cdclW-stgy (toS (rough-state-from-init-state-of S))
    (toS (rough-state-from-init-state-of (do-cdclW-stgy-step' S)))
    = cdclW-stgy (toS (rough-state-of (id-of-I-to S)))
    (toS (rough-state-of (do-cdclW-stgy-step (id-of-I-to S))))
    using id-of-I-to-def rough-state-from-init-state-of-do-cdclW-stgy-step'
    toS-rough-state-of-state-of-rough-state-from-init-state-of by presburger
  then show ?thesis
    using f3 f2 IH do-cdclW-stgy-step by fastforce
qed
qed

```

Final theorem:

```

lemma DPLL-tot-correct:
  assumes
    r: rough-state-from-init-state-of (do-all-cdclW-stgy (state-from-init-state-of
      (([], map remdups N, [], 0, None)))) = S and
    S: (M', N', U', k, E) = toS S
  shows (E ≠ Some {#} ∧ satisfiable (set (map mset N)))
    ∨ (E = Some {#} ∧ unsatisfiable (set (map mset N)))
proof -
  let ?N = map remdups N
  have inv: cdclW-all-struct-inv (toS ([], map remdups N, [], 0, None))
    unfolding cdclW-all-struct-inv-def distinct-cdclW-state-def distinct-mset-set-def by auto

```



```

then have S0: rough-state-of (state-of ([], map remdups N, [], 0, None))
  = ([], map remdups N, [], 0, None) by simp
have 1: full cdclW-stgy (toS ([], ?N, [], 0, None)) (toS S)
  unfolding full-def apply rule
    using do-all-cdclW-stgy-is-rtrancpl-cdclW-stgy[of
      state-from-init-state-of ([], map remdups N, [], 0, None)] inv
      no-step-cdclW-stgy-cdclW-all
    by (auto simp del: do-all-cdclW-stgy.simps simp: state-from-init-state-of-inverse
      r[symmetric]) +
moreover have 2: finite (set (map mset ?N)) by auto
moreover have 3: distinct-mset-set (set (map mset ?N))
  unfolding distinct-mset-set-def by auto
moreover
  have cdclW-all-struct-inv (toS S)
    by (metis (no-types) cdclW-all-struct-inv-rough-state r
      toS-rough-state-of-state-of-rough-state-from-init-state-of)
  then have cons: consistent-interp (lits-of M')
    unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def S[symmetric] by auto
moreover
  have clss (toS ([], ?N, [], 0, None)) = clss (toS S)
    apply (rule rtrancpl-cdclW-init-clss)
    using 1 unfolding full-def by (auto simp add: rtrancpl-cdclW-stgy-rtrancpl-cdclW)
  then have N': mset (map mset ?N) = N'
    using S[symmetric] by auto
have (E ≠ Some {#} ∧ satisfiable (set (map mset ?N)))
  ∨ (E = Some {#} ∧ unsatisfiable (set (map mset ?N)))
  using full-cdclW-stgy-final-state-conclusive unfolding N' apply rule
    using 1 apply simp
    using 2 apply simp
    using 3 apply simp
    using S[symmetric] N' apply auto[1]
  using S[symmetric] N' cons by (fastforce simp: true-annots-true-cl)
then show ?thesis by auto
qed

```

The Code The SML code is skipped in the documentation, but stays to ensure that some version of the exported code is working. The only difference between the generated code and the one used here is the export of the constructor `ConI`.

```

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
begin

```

```

lemma backtrack-levE:

```

$backtrack\ S\ S' \implies cdcl_W\text{-}M\text{-level-inv}\ S \implies$
 $(\bigwedge D\ L\ K\ M1\ M2.$
 $(Marked\ K\ (Suc\ (get\text{-}maximum\text{-}level\ (trail\ S)\ D))\ \# \ M1,\ M2)$
 $\in\ set\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S)) \implies$
 $get\text{-}level\ (trail\ S)\ L = get\text{-}maximum\text{-}level\ (trail\ S)\ (D + \{\#L\# \}) \implies$
 $undefined\text{-}lit\ M1\ L \implies$
 $S' \sim cons\text{-}trail\ (Propagated\ L\ (D + \{\#L\# \}))$
 $(reduce\text{-}trail\text{-}to\ M1\ (add\text{-}learned\text{-}cls\ (D + \{\#L\# \}))$
 $(update\text{-}backtrack\text{-}lvl\ (get\text{-}maximum\text{-}level\ (trail\ S)\ D)\ (update\text{-}conflicting\ None\ S)))) \implies$
 $backtrack\text{-}lvl\ S = get\text{-}maximum\text{-}level\ (trail\ S)\ (D + \{\#L\# \}) \implies$
 $conflicting\ S = Some\ (D + \{\#L\# \}) \implies P) \implies$
 P
using *assms* **by** (*induction rule: backtrack-induction-lev2*) *metis*

lemma *backtrack-no-cdcl_W-bj*:

assumes *cdcl*: *cdcl_W-bj* *T U* **and** *inv*: *cdcl_W-M-level-inv* *V*
shows $\neg backtrack\ V\ T$
using *cdcl inv*
apply (*induction rule: cdcl_W-bj.induct*)
apply (*elim skipE, force elim!: backtrack-levE[OF - inv] simp: cdcl_W-M-level-inv-def*)
apply (*elim resolveE, force elim!: backtrack-levE[OF - inv] simp: cdcl_W-M-level-inv-def*)
apply *standard*
apply (*elim backtrack-levE[OF - inv], elim backtrackE*)
apply (*force simp del: state-simp simp add: state-eq-conflicting cdcl_W-M-level-inv-decomp*)
done

abbreviation *skip-or-resolve* :: *'st* \Rightarrow *'st* \Rightarrow *bool* **where**

skip-or-resolve $\equiv (\lambda S\ T. skip\ S\ T \vee resolve\ S\ T)$

lemma *rtranclp-cdcl_W-bj-skip-or-resolve-backtrack*:

assumes *cdcl_W-bj*** *S U* **and** *inv*: *cdcl_W-M-level-inv* *S*
shows *skip-or-resolve*** *S U* $\vee (\exists T. skip\text{-}or\text{-}resolve^{**}\ S\ T \wedge backtrack\ T\ U)$
using *assms*

proof (*induction*)

case *base*

then show *?case* **by** *simp*

next

case (*step* *U V*) **note** *st* = *this(1)* **and** *bj* = *this(2)* **and** *IH* = *this(3)[OF this(4)]*

consider

(*SU*) *S* = *U*

| (*SUp*) *cdcl_W-bj⁺⁺* *S U*

using *st* **unfolding** *rtranclp-unfold* **by** *blast*

then show *?case*

proof *cases*

case *SUp*

have $\bigwedge T. skip\text{-}or\text{-}resolve^{**}\ S\ T \implies cdcl_W^{**}\ S\ T$

using *mono-rtranclp[of skip-or-resolve cdcl_W]* **other** **by** *blast*

then have *skip-or-resolve*** *S U*

using *bj IH inv backtrack-no-cdcl_W-bj rtranclp-cdcl_W-consistent-inv[OF - inv]* **by** *meson*

then show *?thesis*

using *bj* **by** (*metis (no-types, lifting) cdcl_W-bj.cases rtranclp.simps*)

next

case *SU*

then show *?thesis*

using *bj* **by** (*metis* (*no-types*, *lifting*) *cdcl_W-bj.cases rtrancpl.simps*)
qed
qed

lemma *rtrancpl-skip-or-resolve-rtrancpl-cdcl_W*:
*skip-or-resolve^{**} S T \implies 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 $\equiv \lambda C\ C'\ L'\ S.$ True

definition *inv_{NOT}* :: '*st* \Rightarrow bool **where**
inv_{NOT} $\equiv \lambda S.$ no-dup (*trail* *S*)

declare *inv_{NOT}-def*[*simp*]
end

fun *convert-marked-lit-from-W* **where**
convert-marked-lit-from-W (*Propagated* *L* -) = *Propagated* *L* () |
convert-marked-lit-from-W (*Marked* *L* -) = *Marked* *L* ()

abbreviation *convert-trail-from-W* ::
('v, 'vl, 'a) marked-lit list
 \Rightarrow ('v, unit, unit) marked-lit list **where**
convert-trail-from-W \equiv map *convert-marked-lit-from-W*

lemma *lits-of-convert-trail-from-W*[*simp*]:
lits-of (*convert-trail-from-W* *M*) = *lits-of* *M*
by (*induction rule: marked-lit-list-induct*) *simp-all*

lemma *lit-of-convert-trail-from-W*[*simp*]:
lit-of (*convert-marked-lit-from-W* *L*) = *lit-of* *L*
by (*cases* *L*) *auto*

lemma *no-dup-convert-from-W*[*simp*]:
no-dup (*convert-trail-from-W* *M*) \longleftrightarrow no-dup *M*
by (*auto simp: comp-def*)

lemma *convert-trail-from-W-true-annots*[*simp*]:
convert-trail-from-W *M* $\models_{as}\ C \longleftrightarrow M \models_{as}\ C$
by (*auto simp: true-annots-true-cl*s)

lemma *defined-lit-convert-trail-from-W*[*simp*]:
defined-lit (*convert-trail-from-W* *S*) *L* \longleftrightarrow *defined-lit* *S* *L*
by (*auto simp: defined-lit-map image-comp*)

The values 0 and {#} are dummy values.

fun *convert-marked-lit-from-NOT*
:: ('a, 'e, 'b) marked-lit \Rightarrow ('a, nat, 'a literal multiset) marked-lit **where**
convert-marked-lit-from-NOT (*Propagated* *L* -) = *Propagated* *L* {#} |
convert-marked-lit-from-NOT (*Marked* *L* -) = *Marked* *L* 0

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT \equiv map *convert-marked-lit-from-NOT*

lemma *undefined-lit-convert-trail-from-NOT*[simp]:
undefined-lit (convert-trail-from-NOT F) L \longleftrightarrow undefined-lit F L
by (*induction F rule: marked-lit-list-induct*) (*auto simp: defined-lit-map*)

lemma *lits-of-convert-trail-from-NOT*:
lits-of (convert-trail-from-NOT F) = lits-of F
by (*induction F rule: marked-lit-list-induct*) *auto*

lemma *convert-trail-from-W-from-NOT*[simp]:
convert-trail-from-W (convert-trail-from-NOT M) = M
by (*induction rule: marked-lit-list-induct*) *auto*

lemma *convert-trail-from-W-convert-lit-from-NOT*[simp]:
convert-marked-lit-from-W (convert-marked-lit-from-NOT L) = L
by (*cases L*) *auto*

abbreviation *trail_{NOT}* **where**
trail_{NOT} S \equiv convert-trail-from-W (fst S)

lemma *undefined-lit-convert-trail-from-W*[iff]:
undefined-lit (convert-trail-from-W M) L \longleftrightarrow undefined-lit M L
by (*auto simp: defined-lit-map image-comp*)

lemma *lit-of-convert-marked-lit-from-NOT*[iff]:
lit-of (convert-marked-lit-from-NOT L) = lit-of L
by (*cases L*) *auto*

sublocale *state_W \subseteq dp11-state*
 $\lambda S.$ convert-trail-from-W (trail S)
clauses
 $\lambda L S.$ cons-trail (convert-marked-lit-from-NOT L) S
 $\lambda S.$ tl-trail S
 $\lambda C S.$ add-learned-cls C S
 $\lambda C S.$ remove-cls C S
by *unfold-locales (auto simp: map-tl o-def)*

context *state_W*
begin
declare *state-simp_{NOT}*[*simp del*]
end

sublocale *cdcl_W \subseteq cdcl_{NOT}-merge-bj-learn-ops*
 $\lambda S.$ convert-trail-from-W (trail S)
clauses
 $\lambda L S.$ cons-trail (convert-marked-lit-from-NOT L) S
 $\lambda S.$ tl-trail S
 $\lambda C S.$ add-learned-cls C S
 $\lambda C S.$ remove-cls C S
 $\lambda -.$ True
 $\lambda - S.$ conflicting S = None
 $\lambda C C' L' S.$ backjump-l-cond C C' L' S \wedge distinct-mset (C' + {#L'#}) \wedge \neg tautology (C' + {#L'#})
by *unfold-locales*

sublocale *cdcl_W \subseteq cdcl_{NOT}-merge-bj-learn-proxy*
 $\lambda S.$ convert-trail-from-W (trail S)

```

clauses
λL S. cons-trail (convert-marked-lit-from-NOT L) S
λS. tl-trail S
λC S. add-learned-cls C S
λC S. remove-cls C S
λ- -. True
λ- S. conflicting S = None backjump-l-cond invNOT
proof (unfold-locales, goal-cases)
  case 2
  then show ?case using cdclNOT-merged-bj-learn-no-dup-inv by (auto simp: comp-def)
next
case (1 C' S C F' K F L)
moreover
  let ?C' = remdups-mset C'
  have L ∉ # C'
    using ⟨F ⊨as CNot C'⟩ ⟨undefined-lit F L⟩ Marked-Propagated-in-iff-in-lits-of
    in-CNot-implies-uminus(2) by blast
  then have distinct-mset (?C' + {#L#})
    by (metis count-mset-set(3) distinct-mset-remdups-mset distinct-mset-single-add
    less-irrefl-nat mem-set-mset-iff remdups-mset-def)
moreover
  have no-dup F
    using ⟨invNOT S⟩ ⟨convert-trail-from-W (trail S) = F' @ Marked K () # F⟩
    unfolding invNOT-def
    by (smt comp-apply distinct.simps(2) distinct-append list.simps(9) map-append
    no-dup-convert-from-W)
  then have consistent-interp (lits-of F)
    using distinctconsistent-interp by blast
  then have ¬ tautology (C')
    using ⟨F ⊨as CNot C'⟩ consistent-CNot-not-tautology true-annots-true-cls by blast
  then have ¬ tautology (?C' + {#L#})
    using ⟨F ⊨as CNot C'⟩ ⟨undefined-lit F L⟩ by (metis CNot-remdups-mset
    Marked-Propagated-in-iff-in-lits-of add commute in-CNot-uminus tautology-add-single
    tautology-remdups-mset true-annot-singleton true-annots-def)
show ?case
proof –
  have f2: no-dup (convert-trail-from-W (trail S))
    using ⟨invNOT S⟩ unfolding invNOT-def by (simp add: o-def)
  have f3: atm-of L ∈ atms-of-msu (clauses S)
    ∪ atm-of ‘lits-of (convert-trail-from-W (trail S))’
    using ⟨convert-trail-from-W (trail S) = F' @ Marked K () # F⟩
    ⟨atm-of L ∈ atms-of-msu (clauses S) ∪ atm-of ‘lits-of (F' @ Marked K () # F)⟩ by auto
  have f4: clauses S ⊨pm remdups-mset C' + {#L#}
    by (metis (no-types) ⟨L ∉ # C'⟩ ⟨clauses S ⊨pm C' + {#L#}⟩ remdups-mset-singleton-sum(2)
    true-clss-cls-remdups-mset union-commute)
  have F ⊨as CNot (remdups-mset C')
    by (simp add: ⟨F ⊨as CNot C'⟩)
  then show ?thesis
    using f4 f3 f2 ⟨¬ tautology (remdups-mset C' + {#L#})⟩
    backjump-l.intros[OF - f2] calculation(2–5,9)
    state-eqNOT-ref unfolding backjump-l-cond-def by blast
qed
qed

```

sublocale *cdcl_W* ⊆ *cdclNOT-merge-bj-learn-proxy2*

$\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$
clauses
 $\lambda L S. \text{cons-trail (convert-marked-lit-from-NOT } L) S$
 $\lambda S. \text{tl-trail } S$
 $\lambda C S. \text{add-learned-cls } C S$
 $\lambda C S. \text{remove-cls } C S \lambda - -. \text{True inv}_{NOT}$
 $\lambda - S. \text{conflicting } S = \text{None backjump-l-cond}$
by *unfold-locales*

sublocale $cdcl_W \subseteq cdcl_{NOT\text{-merge-bj-learn}}$
 $\lambda S. \text{convert-trail-from-}W \text{ (trail } S)$
clauses
 $\lambda L S. \text{cons-trail (convert-marked-lit-from-NOT } L) S$
 $\lambda S. \text{tl-trail } S$
 $\lambda C S. \text{add-learned-cls } C S$
 $\lambda C S. \text{remove-cls } C S \lambda - -. \text{True inv}_{NOT}$
 $\lambda - S. \text{conflicting } S = \text{None backjump-l-cond}$
apply *unfold-locales*
using *dpll-bj-no-dup apply (simp add: comp-def)*
using *cdcl_{NOT-no-dup by (auto simp add: comp-def cdcl_{NOT.simps})}*

context $cdcl_W$
begin

Notations are lost while proving locale inclusion:

notation *state-eq_{NOT} (infix \sim_{NOT} 50)*

19.2 Additional Lemmas between NOT and W states

lemma *trail_W-eq-reduce-trail-to_{NOT}-eq:*
 $\text{trail } S = \text{trail } T \implies \text{trail (reduce-trail-to}_{NOT} F S) = \text{trail (reduce-trail-to}_{NOT} F T)$
proof (*induction F S arbitrary: T rule: reduce-trail-to_{NOT}.induct*)
case ($1 F S T$) **note** $IH = \text{this}(1)$ **and** $tr = \text{this}(2)$
then have $\square = \text{convert-trail-from-}W \text{ (trail } S)$
 $\vee \text{length } F = \text{length (convert-trail-from-}W \text{ (trail } S))$
 $\vee \text{trail (reduce-trail-to}_{NOT} F (\text{tl-trail } S)) = \text{trail (reduce-trail-to}_{NOT} F (\text{tl-trail } T))$
using IH **by** (*metis (no-types) trail-tl-trail*)
then show $\text{trail (reduce-trail-to}_{NOT} F S) = \text{trail (reduce-trail-to}_{NOT} F T)$
using tr **by** (*metis (no-types) reduce-trail-to_{NOT}.elims*)
qed

lemma *trail-reduce-trail-to_{NOT}-add-learned-cls:*
 $\text{no-dup (trail } S) \implies$
 $\text{trail (reduce-trail-to}_{NOT} M (\text{add-learned-cls } D S)) = \text{trail (reduce-trail-to}_{NOT} M S)$
by (*rule trail_W-eq-reduce-trail-to_{NOT}-eq simp*)

lemma *reduce-trail-to_{NOT}-reduce-trail-convert:*
 $\text{reduce-trail-to}_{NOT} C S = \text{reduce-trail-to (convert-trail-from-NOT } C) S$
apply (*induction C S rule: reduce-trail-to_{NOT}.induct*)
apply (*subst reduce-trail-to_{NOT}.simps, subst reduce-trail-to.simps*)
by *auto*

lemma *reduce-trail-to-length:*
 $\text{length } M = \text{length } M' \implies \text{reduce-trail-to } M S = \text{reduce-trail-to } M' S$
apply (*induction M S arbitrary: rule: reduce-trail-to.induct*)
apply (*rename-tac F S; case-tac trail S $\neq \square$; case-tac length (trail S) $\neq \text{length } M'$*)

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 = None then 0 else 1)*
> length (trail T) + (if conflicting T = None then 0 else 1)

using *assms* **by** (*induction rule: cdcl_W-bj.induct*)

(*force dest:arg-cong[of - - length]*)

intro: get-all-marked-decomposition-exists-prepend

elim!: backtrack-levE

simp: cdcl_W-M-level-inv-def)+

lemma *wf-cdcl_W-bj*:

wf {(b,a). cdcl_W-bj a b ∧ cdcl_W-M-level-inv a}

apply (*rule wfP-if-measure[of λ-. True*

- λT. length (trail T) + (if conflicting T = None then 0 else 1), simplified)

using *cdcl_W-bj-measure* **by** *blast*

lemma *cdcl_W-bj-exists-normal-form*:

assumes *lev: cdcl_W-M-level-inv S*

shows $\exists T. \text{full } cdcl_W\text{-bj } S \ T$

proof –

obtain *T* **where** *T: full (λa b. cdcl_W-bj a b ∧ cdcl_W-M-level-inv a) S T*

using *wf-exists-normal-form-full[OF wf-cdcl_W-bj]* **by** *auto*

then have *cdcl_W-bj** S T*

by (*auto dest: 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

$\exists M. \text{trail } S = M @ \text{trail } T \wedge (\forall m \in \text{set } M. \neg \text{is-marked } m)$ **and**

T ~ delete-trail-and-rebuild (trail T) S

using *assms* **by** (*induction rule: rtrancp-induct*)

(*auto simp del: state-simp simp: state-eq-def state-access-simp*)

19.3.2 More backjumping

Backjumping after skipping or jump directly **lemma** *rtrancp-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 rtrancp-mono[of skip cdclW] assms(3) rtrancp-cdclW-all-struct-inv-inv mono-rtrancp
    by (auto dest!: bj other cdclW-bj.skip)
  then have cdclW-M-level-inv V
    unfolding cdclW-all-struct-inv-def by auto
  then obtain N k M1 M2 K D L U i where
    V: state V = (trail V, N, U, k, Some (D + {#L#})) and
    W: state W = (Propagated L (D + {#L#}) # M1, N, {#D + {#L#}#} + U,
      get-maximum-level (trail V) D, None) and
    decomp: (Marked K (Suc i) # M1, M2)
      ∈ set (get-all-marked-decomposition (trail V)) and
    k = get-maximum-level (trail V) (D + {#L#}) and
    lev-L: get-level (trail V) L = 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 (trail V) D) (update-conflicting None V))))and
    lev-l-D: backtrack-lvl V = get-maximum-level (trail V) (D + {#L#}) and
    conflicting V = Some (D + {#L#}) and
    i: i = get-maximum-level (trail V) D
    using bt by (elim backtrack-levE)
    (auto simp: cdclW-M-level-inv-decomp state-eq-def simp del: state-simp)+
  let ?D = (D + {#L#})
  obtain L' C' where
    T: state T = (Propagated L' C' # trail V, N, U, k, Some ?D) and
    V ~ tl-trail T and
    -L' ∉ # ?D and
    ?D ≠ {#}
    using skip V by force

  let ?M = Propagated L' C' # trail V
  have cdclW** S T using bj cdclW-bj.skip mono-rtrancp[of skip cdclW S T] other st by meson
  then have inv': cdclW-all-struct-inv T
    using rtrancp-cdclW-all-struct-inv-inv inv by blast
  have M-lev: cdclW-M-level-inv T using inv' unfolding cdclW-all-struct-inv-def by auto
  then have n-d': no-dup ?M
    using T unfolding cdclW-M-level-inv-def by auto

  have k > 0
    using decomp M-lev T V unfolding cdclW-M-level-inv-def by auto
  then have atm-of L ∈ atm-of ' lits-of (trail V)
    using lev-L get-rev-level-ge-0-atm-of-in V by fastforce

```



```

then have L-L': atm-of L ≠ atm-of L'
  using n-d' unfolding lits-of-def by auto
have L'-M: atm-of L' ∉ atm-of ' lits-of (trail V)
  using n-d' unfolding lits-of-def by auto
have ?M ⊨as CNot ?D
  using inv' T unfolding cdclW-conflicting-def cdclW-all-struct-inv-def by auto
then have L' ∉ # ?D
  using L-L' L'-M unfolding true-annots-def by (auto simp add: true-annot-def true-cls-def
    atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set Ball-mset-def
    split: split-if-asm)
have [simp]: trail (reduce-trail-to M1 T) = M1
  by (metis (mono-tags, lifting) One-nat-def Pair-inject T ⟨V ∼ tl-trail T⟩ decomp
    diff-less in-get-all-marked-decomposition-trail-update-trail length-greater-0-conv
    length-tl lessI list.distinct(1) reduce-trail-to-length-ne state-eq-trail
    trail-reduce-trail-to-length-le trail-tl-trail)
have skip** S V
  using st skip by auto
have no-dup (trail S)
  using inv unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def by auto
then have [simp]: init-cls S = N and [simp]: learned-cls S = U
  using rtrancp-skip-state-decomp[OF ⟨skip** S V⟩] V
  by (auto simp del: state-simp simp: state-eq-def state-access-simp)
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 None 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 ?M L = 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 ?M L = get-maximum-level ?M (D + {#L#})
    using lev-l-D[symmetric] L-L' V lev-L by simp
moreover have i = get-maximum-level ?M D
  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 (rename-tac L m xs; 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: $\text{skip}^{**} S T$ **and**
bt: $\text{backtrack} S W$ **and**
inv: $\text{cdcl}_W\text{-all-struct-inv} S$
shows $\text{backtrack} T W$
using *assms*
proof –
have *M-lev*: $\text{cdcl}_W\text{-M-level-inv} S$
using *bt inv* **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def}$ **by** (*auto elim!*: backtrack-levE)
then obtain $k M M1 M2 K i D L N U$ **where**
S: $\text{state } S = (M, N, U, k, \text{Some } (D + \{\#L\#}))$ **and**
W: $\text{state } W = (\text{Propagated } L (D + \{\#L\#}) \# M1, N, \{\#D + \{\#L\# \# \} + U, \text{get-maximum-level } M D,$
None) **and**
decomp: $(\text{Marked } K (i+1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$ **and**
lev-l: $\text{get-level } M L = k$ **and**
lev-l-D: $\text{get-level } M L = \text{get-maximum-level } M (D + \{\#L\#})$ **and**
i: $i = \text{get-maximum-level } M D$ **and**
undef: $\text{undefined-lit } M1 L$
using *bt* **by** (*elim backtrack-levE*)
(simp-all add: cdcl_W-M-level-inv-decomp state-eq-def del: state-simp)
let $?D = (D + \{\#L\#})$

have [*simp*]: $\text{no-dup } (\text{trail } S)$
using *M-lev* **by** (*auto simp: cdcl_W-M-level-inv-decomp*)
have $\text{cdcl}_W\text{-all-struct-inv} T$
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*]: $\text{no-dup } (\text{trail } T)$
unfolding $\text{cdcl}_W\text{-all-struct-inv-def}$ $\text{cdcl}_W\text{-M-level-inv-def}$ **by** *auto*

obtain $MS M_T$ **where** $M: M = MS @ M_T$ **and** $M_T: M_T = \text{trail } T$ **and** $nm: \forall m \in \text{set } MS. \neg \text{is-marked } m$
using *rtrancpl-skip-state-decomp*(1)[*OF skip*] *S M-lev* **by** *auto*
have *T*: $\text{state } T = (M_T, N, U, k, \text{Some } ?D)$
using M_T *rtrancpl-skip-state-decomp*(2)[*of S T*] *skip S*
by (*auto simp del: state-simp simp: state-eq-def state-access-simp*)

have $\text{cdcl}_W\text{-all-struct-inv} T$
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 $M_T \models_{\text{as}} \text{CNot } ?D$
unfolding $\text{cdcl}_W\text{-all-struct-inv-def}$ $\text{cdcl}_W\text{-conflicting-def}$ **using** *T* **by** *blast*
have $\forall L \in \# ?D. \text{atm-of } L \in \text{atm-of ' lits-of } M_T$
proof –
have $f1: \bigwedge l. \neg M_T \models_{\text{a}} \{\# - l\# \} \vee \text{atm-of } l \in \text{atm-of ' lits-of } M_T$
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 $\langle M_T \models_{\text{as}} \text{CNot } (D + \{\#L\#}) \rangle$ *multi-member-split* **by** *fastforce*
then show *?thesis*
using *f1* **by** (*meson* $\langle M_T \models_{\text{as}} \text{CNot } (D + \{\#L\#}) \rangle$ *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  $H: \bigwedge L. L \in \#D \implies \text{get-level } M L = \text{get-level } M_T L$ 
  unfolding M by (fastforce simp: lits-of-def)
have [simp]: get-maximum-level M ?D = get-maximum-level MT ?D
  by (metis  $\langle M_T \models_{as} CNot (D + \{\#L\# \}) \rangle$  M nm ball-msetI true-annots-CNot-all-atms-defined
    get-maximum-level-skip-un-marked-not-present)

have lev-l': get-level MT L = k
  using lev-l by (auto simp: H)
have [simp]: trail (reduce-trail-to M1 T) = M1
  using T decomp M nm by (smt MT append-assoc beginning-not-marked-invert
    get-all-marked-decomposition-exists-prepend reduce-trail-to-trail-tl-trail-decomp)
have W:  $W \sim \text{cons-trail } (\text{Propagated } L (D + \{\#L\# \})) (\text{reduce-trail-to } M1$ 
   $(\text{add-learned-cls } (D + \{\#L\# \}) (\text{update-backtrack-lvl } i (\text{update-conflicting } None T))))$ 
  using W T i decomp undef by (auto simp del: state-simp simp: state-eq-def)

have lev-l-D': get-level MT L = get-maximum-level MT (D +  $\{\#L\# \}$ )
  using lev-l-D by (auto simp: H)
have [simp]: get-maximum-level M D = get-maximum-level MT D
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':  $i = \text{get-maximum-level } M_T D$ 
  using i by auto
have Marked K (i + 1) # M1 ∈ set (map fst (get-all-marked-decomposition M))
  using Set.imageI[OF decomp, of fst] by auto
then have Marked K (i + 1) # M1 ∈ set (map fst (get-all-marked-decomposition MT))
  using fst-get-all-marked-decomposition-prepend-not-marked[OF nm] unfolding M by auto
then obtain M2' where decomp': (Marked K (i+1) # M1, M2') ∈ set (get-all-marked-decomposition
MT)
  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
       $bt: \text{backtrack } V T$  and
       $\text{skip-or-resolve}^{**} S V$ 
      by blast
    have  $cdcl_W^{**} S V$ 
      using  $\langle \text{skip-or-resolve}^{**} S V \rangle \text{rtrancp-skip-or-resolve-rtrancp-cdcl}_W$  by blast
    then have  $cdcl_W\text{-}M\text{-level-inv } V$  and  $cdcl_W\text{-}M\text{-level-inv } S$ 
      using  $\text{rtrancp-cdcl}_W\text{-consistent-inv inv}$  by blast+
    with  $bj\ bt$  have False using  $\text{backtrack-no-cdcl}_W\text{-bj}$  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 rtrancp.simps)+
qed

lemma resolve-skip-deterministic:
   $\text{resolve } S\ T \implies \text{skip } S\ U \implies \text{False}$ 
by fastforce

lemma backtrack-unique:
assumes
   $bt\text{-}T: \text{backtrack } S\ T$  and
   $bt\text{-}U: \text{backtrack } S\ U$  and
   $inv: \text{cdcl}_W\text{-all-struct-inv } S$ 
shows  $T \sim U$ 
proof –
  have  $lev: \text{cdcl}_W\text{-}M\text{-level-inv } S$ 
    using  $inv$  unfolding  $\text{cdcl}_W\text{-all-struct-inv-def}$  by auto
then obtain  $M\ N\ U'\ k\ D\ L\ i\ K\ M1\ M2$  where
   $S: \text{state } S = (M, N, U', k, \text{Some } (D + \{\#L\#}))$  and
   $\text{decomp}: (\text{Marked } K\ (i+1)\ \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } M)$  and
   $\text{get-level } M\ L = k$  and
   $\text{get-level } M\ L = \text{get-maximum-level } M\ (D + \{\#L\#})$  and
   $\text{get-maximum-level } M\ D = i$  and
   $T: \text{state } T = (\text{Propagated } L\ (D + \{\#L\#}))\ \# M1, N, \{\#D + \{\#L\#\}\# + U', i, \text{None}\}$  and
   $\text{undef}: \text{undefined-lit } M1\ L$ 
using  $bt\text{-}T$  by (elim backtrack-levE)
  (force simp: cdclW-M-level-inv-def state-eq-def simp del: state-simp)+

obtain  $D'\ L'\ i'\ K'\ M1'\ M2'$  where
   $S': \text{state } S = (M, N, U', k, \text{Some } (D' + \{\#L'\#}))$  and
   $\text{decomp}': (\text{Marked } K'\ (i'+1)\ \# M1', M2') \in \text{set } (\text{get-all-marked-decomposition } M)$  and
   $\text{get-level } M\ L' = k$  and
   $\text{get-level } M\ L' = \text{get-maximum-level } M\ (D' + \{\#L'\#})$  and
   $\text{get-maximum-level } M\ D' = i'$  and
   $U: \text{state } U = (\text{Propagated } L'\ (D' + \{\#L'\#}))\ \# M1', N, \{\#D' + \{\#L'\#\}\# + U', i', \text{None}\}$  and
   $\text{undef}: \text{undefined-lit } M1'\ L'$ 
using  $bt\text{-}U\ lev\ S$  by (elim backtrack-levE)

```

```

  (force simp: cdclW-M-level-inv-def state-eq-def simp del: state-simp)+
obtain c where M: M = c @ M2 @ Marked K (i + 1) # M1
  using decomp by auto
obtain c' where M': M = c' @ M2' @ Marked K' (i' + 1) # M1'
  using decomp' by auto
have marked: get-all-levels-of-marked M = rev [1..l+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 ¬ ?thesis
  then have L' ∈ # D
    using S unfolding S' by (fastforce simp: multiset-eq-iff split: split-if-asm)
  then have get-maximum-level M D ≥ k
    using ⟨get-level M L' = k⟩ get-maximum-level-ge-get-level by blast
  then show False using ⟨get-maximum-level M D = i⟩ ⟨i < k⟩ by auto
qed
then have [simp]: D = D'
  using S S' by auto
have [simp]: i=i' using ⟨get-maximum-level M D' = i'⟩ ⟨get-maximum-level M D = i⟩ by auto

```

Automation in a step later...

```

have H: ∧ a A B. insert a A = B ⟹ a : B
  by blast
have get-all-levels-of-marked (c@M2) = rev [i+2..l+k] and
  get-all-levels-of-marked (c'@M2') = rev [i+2..l+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 (λL. ¬is-marked L ∨ level-of L ≠ Suc i) (c @ M2) = [] and
  dropWhile (λL. ¬is-marked L ∨ level-of L ≠ Suc i) (c' @ M2') = []
  unfolding dropWhile-eq-Nil-conv Ball-def
  by (intro allI; rename-tac x; case-tac x; auto dest!: H simp add: in-set-conv-decomp)+

then have M1 = M1'
  using arg-cong[OF M, of dropWhile (λL. ¬is-marked L ∨ level-of L ≠ 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 ¬resolve U V
proof (rule ccontr)
  assume resolve: ¬¬resolve U V

obtain L C M N U' k D where

```

U : state $U = (\text{Propagated } L \ ((C + \{\#L\#\}) \# M, N, U', k, \text{Some } (D + \{\#-L\#\})) \text{and}$
 $\text{get-maximum-level } (\text{Propagated } L \ (C + \{\#L\#\}) \# M) \ D = k \text{ and}$
 $\text{state } V = (M, N, U', k, \text{Some } (D \# \cup C))$
using *resolve* **by** *auto*
have $\text{cdcl}_W\text{-all-struct-inv } U$
using $\text{mono-rtrancpl}[\text{of skip cdcl}_W]$ **by** $(\text{meson } bj \text{ cdcl}_W\text{-bj.skip inv local.skip other}$
 $\text{rtrancpl-cdcl}_W\text{-all-struct-inv-inv})$
then have $[iff]: \text{no-dup } (\text{trail } S) \text{ cdcl}_W\text{-M-level-inv } S$ **and** $[iff]: \text{no-dup } (\text{trail } U)$
using *inv unfolding* $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$ **by** *blast+*
then have
 S : *init-clss* $S = N$
learned-clss $S = U'$
backtrack-lvl $S = k$
conflicting $S = \text{Some } (D + \{\#-L\#\})$
using $\text{rtrancpl-skip-state-decomp}(2)[OF \text{ skip}] \ U$
by $(\text{auto simp del: state-simp simp: state-eq-def state-access-simp})$
obtain M_0 **where**
 $\text{tr-S: trail } S = M_0 @ \text{trail } U$ **and**
 $\text{nm: } \forall m \in \text{set } M_0. \neg \text{is-marked } m$
using $\text{rtrancpl-skip-state-decomp}[OF \text{ skip}]$ **by** *blast*

obtain $M' \ D' \ L' \ i \ K \ M1 \ M2$ **where**
 S' : state $S = (M', N, U', k, \text{Some } (D' + \{\#L'\#\}))$ **and**
 $\text{decomp: } (\text{Marked } K \ (i+1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } M')$ **and**
 $\text{get-level } M' \ L' = k$ **and**
 $\text{get-level } M' \ L' = \text{get-maximum-level } M' \ (D' + \{\#L'\#\})$ **and**
 $\text{get-maximum-level } M' \ D' = i$ **and**
 $\text{undef: undefined-lit } M1 \ L'$ **and**
 T : state $T = (\text{Propagated } L' \ (D' + \{\#L'\#\}) \# M1, N, \{\#D' + \{\#L'\#\}\# + U', i, \text{None})$
using *bt* **by** $(\text{elim backtrack-levE}) \ (\text{fastforce simp: } S \text{ state-eq-def simp del: state-simp}) +$
obtain c **where** $M: M' = c @ M2 @ \text{Marked } K \ (i + 1) \# M1$
using $\text{get-all-marked-decomposition-exists-prepend}[OF \text{ decomp}]$ **by** *auto*
have *marked: get-all-levels-of-marked* $M' = \text{rev } [1..<1+k]$
using *inv S' unfolding* $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$ **by** *auto*
then have $i < k$
unfolding M **by** $(\text{force simp add: rev-swap[symmetric] dest!: arg-cong[of - - set]})$

have $DD': D' + \{\#L'\#\} = D + \{\#-L\#\}$
using $S \ S'$ **by** *auto*
have $[simp]: L' = -L$
proof (rule ccontr)
assume $\neg ?thesis$
then have $-L \in \# \ D'$
using DD' **by** $(\text{metis add-diff-cancel-right' diff-single-trivial diff-union-swap}$
 $\text{multi-self-add-other-not-self})$
moreover
have $M': M' = M_0 @ \text{Propagated } L \ ((C + \{\#L\#\}) \# M$
using $\text{tr-S } U \ S \ S'$ **by** $(\text{auto simp: lits-of-def})$
have *no-dup* M'
using *inv U S' unfolding* $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$ **by** *auto*
have *atm-L-notin-M: atm-of* $L \notin \text{atm-of ' (lits-of } M)$
using $\langle \text{no-dup } M' \rangle \ M' \ U \ S \ S'$ **by** $(\text{auto simp: lits-of-def})$
have *get-all-levels-of-marked* $M' = \text{rev } [1..<1+k]$
using *inv U S' unfolding* $\text{cdcl}_W\text{-all-struct-inv-def cdcl}_W\text{-M-level-inv-def}$ **by** *auto*
then have *get-all-levels-of-marked* $M = \text{rev } [1..<1+k]$

```

    using nm M' S' U by (simp add: get-all-levels-of-marked-no-marked)
  then have get-lev-L:
    get-level(Propagated L (C + {#L#}) # M) L = k
    using get-level-get-rev-level-get-all-levels-of-marked[OF atm-L-notin-M,
      of [Propagated L ((C + {#L#}))]] by simp
  have atm-of L ∉ atm-of ' (lits-of (rev M0))
    using ⟨no-dup M'⟩ M' U S' by (auto simp: lits-of-def)
  then have get-level M' L = k
    using get-rev-level-notin-end[of L rev M0
      rev M @ Propagated L (C + {#L#}) # [] 0]
    using tr-S get-lev-L M' U S' by (simp add: nm lits-of-def)
  ultimately have get-maximum-level M' D' ≥ k
    by (metis get-maximum-level-ge-get-level get-rev-level-uminus)
  then show False
    using ⟨i < k⟩ unfolding ⟨get-maximum-level M' D' = i⟩ by auto
qed
have [simp]: D = D' using DD' by auto
have cdclW** S U
  using bj cdclW-bj.skip local.skip mono-rtrancpl[of skip cdclW S U] other by meson
then have cdclW-all-struct-inv U
  using inv rtrancpl-cdclW-all-struct-inv-inv by blast
then have Propagated L ( (C + {#L#}) ) # M ⊨as CNot (D' + {#L'#})
  using cdclW-all-struct-inv-def cdclW-conflicting-def U by auto
then have ∀ L' ∈ #D. atm-of L' ∈ atm-of ' lits-of (Propagated L ( (C + {#L#}) ) # M)
  by (metis CNot-plus CNot-singleton Un-insert-right ⟨D = D'⟩ 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 M' D = k
  using tr-S nm U S'
  get-maximum-level-skip-un-marked-not-present[of D
    Propagated L (C + {#L#}) # M M0]
  unfolding ⟨get-maximum-level (Propagated L (C + {#L#}) # M) D = k⟩
  unfolding ⟨D = D'⟩
  by simp
then show False
  using ⟨get-maximum-level M' D' = i⟩ ⟨i < k⟩ by auto
qed

```

lemma *if-can-apply-resolve-no-more-backtrack:*

```

assumes
  skip: skip** S U and
  resolve: resolve S T and
  inv: cdclW-all-struct-inv S
shows ¬backtrack U V
using assms
by (meson if-can-apply-backtrack-no-more-resolve rtrancpl.rtrancpl-refl
  rtrancpl-skip-backtrack-backtrack)

```

lemma *if-can-apply-backtrack-skip-or-resolve-is-skip:*

```

assumes
  bt: backtrack S T and
  skip: skip-or-resolve** S U and
  inv: cdclW-all-struct-inv S
shows skip** S U
using assms(2,3,1)

```

by induction (simp-all add: if-can-apply-backtrack-no-more-resolve)

lemma *cdcl_W-bj-bj-decomp*:

assumes *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
    ( $\lambda S\ T.$  skip-or-resolve S T  $\wedge$  no-step backtrack S)** S T and
    resolve T U and
    no-step backtrack T and
    skip** U W
  | (S) skip** S W
using IH by auto
then show ?thesis
proof cases
case (RS T U)
have cdclW** S T
  using RS(1) cdclW-bj.resolve cdclW-o.bj other skip
  mono-rtrancpl[of ( $\lambda S\ T.$  skip-or-resolve S T  $\wedge$  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.prems)
{ fix U'
  assume skip** U U' and skip** U' W
  have cdclW-all-struct-inv U'
    using  $\langle$ cdclW-all-struct-inv U $\rangle$   $\langle$ 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  $\langle$ skip** U' W $\rangle$ ] res by blast
}
with  $\langle$ skip** U W $\rangle$ 
have ( $\lambda S\ T.$  skip-or-resolve S T  $\wedge$  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  $\bigwedge U'. \textit{skip}^{**} U' V \implies \textit{skip}^{**} U' W$ 
    using skip by auto
  then have ( $\lambda S\ T.$  skip-or-resolve S T  $\wedge$  no-step backtrack S)** U V
    using IH H by blast
  moreover have ( $\lambda S\ T.$  skip-or-resolve S T  $\wedge$  no-step backtrack S)** V W
    by (simp add: local.skip r-into-rtrancpl st step.prems)
  ultimately show ?case by simp
qed
then show ?thesis
proof -
  have f1:  $\forall p\ pa\ pb\ pc. \neg p\ (pa)\ pb \vee \neg p^{**}\ pb\ pc \vee p^{**}\ pa\ pc$ 
    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 ( $\lambda p$  pa. skip-or-resolve p pa  $\wedge$  no-step backtrack p)** T W
proof -
  have ( $\exists$  vr19 vr16 vr17 vr18. vr19 (vr16::'st) vr17  $\wedge$  vr19** vr17 vr18
     $\wedge$   $\neg$  vr19** vr16 vr18)
     $\vee$   $\neg$  (skip-or-resolve T U  $\wedge$  no-step backtrack T)
     $\vee$   $\neg$  ( $\lambda uu$  uua. skip-or-resolve uu uua  $\wedge$  no-step backtrack uu)** U W
     $\vee$  ( $\lambda uu$  uua. skip-or-resolve uu uua  $\wedge$  no-step backtrack uu)** T W
  by force
then show ?thesis
  by (metis (no-types)  $\langle \lambda S$  T. skip-or-resolve S T  $\wedge$  no-step backtrack S>** U W  $\rangle$ 
     $\langle$  skip-or-resolve T U  $\wedge$  no-step backtrack T  $\rangle$  f1)
qed
then have ( $\lambda p$  pa. skip-or-resolve p pa  $\wedge$  no-step backtrack p)** S W
  using RS(1) by force
then show ?thesis
  using no-bt res by blast
qed
next
case S
{ fix U'
  assume skip** S U' and skip** U' W
  then have cdclW** S U'
    using mono-rtrancpl[of skip cdclW S U'] by (simp add: cdclW-o.bj other skip)
  then have cdclW-all-struct-inv U'
    by (metis (no-types, hide-lams)  $\langle$  cdclW-all-struct-inv S  $\rangle$ 
      rtrancpl-cdclW-all-struct-inv-inv)
  then have no-step backtrack U'
    using if-can-apply-backtrack-no-more-resolve[OF  $\langle$  skip** U' W  $\rangle$ ] res by blast
}
with S
have ( $\lambda S$  T. skip-or-resolve S T  $\wedge$  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  $\bigwedge U'. \text{skip** } U' V \implies \text{skip** } U' W$ 
    using skip by auto
  then have ( $\lambda S$  T. skip-or-resolve S T  $\wedge$  no-step backtrack S)** S V
    using IH H by blast
  moreover have ( $\lambda S$  T. skip-or-resolve S T  $\wedge$  no-step backtrack S)** V W
    by (simp add: local.skip r-into-rtrancpl st step.prem)
  ultimately show ?case by simp
qed
then show ?thesis using res no-bt by blast
qed
qed
qed

```

The case distinction is needed, since $T \sim V$ does not imply that $R^{**} T V$.

lemma *cdcl_W-bj-strongly-confluent*:
 assumes

```

    cdclW-bj** S V and
    cdclW-bj** S T and
    n-s: no-step cdclW-bj V and
    inv: cdclW-all-struct-inv S
  shows T ~ V ∨ cdclW-bj** T V
  using assms(2)
proof induction
  case base
  then show ?case by (simp add: assms(1))
next
  case (step T U) note st = this(1) and s-o-r = this(2) and IH = this(3)
  have cdclW** S T
    using st mono-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 ~ V
    | (bj-TV) cdclW-bj** T V
  using IH by blast
  then show ?case
  proof cases
    case TV
    have no-step cdclW-bj T
      using ⟨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': cdclW-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
    end
  end

```

```

next
  case resolve
  then have  $U \sim U'$ 
    by (meson  $T-U'$   $cdcl_W$ -bj.simps if-can-apply-backtrack-no-more-resolve inv- $T$ 
        resolve-skip-deterministic resolve-unique rtrancpl.rtrancpl-refl)
  then show ?thesis
    using  $\langle cdcl_W$ -bj**  $U' V \rangle$  unfolding rtrancpl-unfold
    by (meson  $T-U'$  bj  $cdcl_W$ -consistent-inv lev- $T$  other state-eq-ref state-eq-sym
        trancpl- $cdcl_W$ -bj-state-eq-compatible)
next
  case skip
  consider
    (sk) skip  $T U'$ 
  | (bt) backtrack  $T U'$ 
  using  $T-U'$  by (meson  $cdcl_W$ -bj.cases local.skip resolve-skip-deterministic)
  then show ?thesis
  proof cases
    case sk
    then show ?thesis
      using  $\langle cdcl_W$ -bj**  $U' V \rangle$  unfolding rtrancpl-unfold
      by (meson  $T-U'$  bj  $cdcl_W$ -all-inv(3)  $cdcl_W$ -all-struct-inv-def inv- $T$  local.skip other
          trancpl- $cdcl_W$ -bj-state-eq-compatible skip-unique state-eq-ref)
    next
    case bt
    have  $skip^{++} T U$ 
      using local.skip by blast
    then show ?thesis
      using bt by (metis  $\langle cdcl_W$ -bj**  $U' V \rangle$  backtrack inv- $T$  trancpl-unfold-begin
          rtrancpl-skip-backtrack-backtrack-end trancpl-into-rtrancpl)
  qed
qed
qed
qed

```

lemma $cdcl_W$ -bj-unique-normal-form:

```

assumes
   $ST$ :  $cdcl_W$ -bj**  $S T$  and  $SU$ :  $cdcl_W$ -bj**  $S U$  and
   $n$ -s- $U$ : no-step  $cdcl_W$ -bj  $U$  and
   $n$ -s- $T$ : no-step  $cdcl_W$ -bj  $T$  and
  inv:  $cdcl_W$ -all-struct-inv  $S$ 
shows  $T \sim U$ 
proof -
  have  $T \sim U \vee cdcl_W$ -bj**  $T U$ 
    using  $ST SU cdcl_W$ -bj-strongly-confluent inv  $n$ -s- $U$  by blast
  then show ?thesis
    by (metis (no-types)  $n$ -s- $T$  rtrancpl-unfold state-eq-ref trancpl-unfold-begin)
qed

```

lemma full- $cdcl_W$ -bj-unique-normal-form:

```

assumes full  $cdcl_W$ -bj  $S T$  and full  $cdcl_W$ -bj  $S U$  and
  inv:  $cdcl_W$ -all-struct-inv  $S$ 
shows  $T \sim U$ 
  using  $cdcl_W$ -bj-unique-normal-form assms unfolding full-def by blast

```

19.4 CDCL FW

inductive $cdcl_W\text{-merge-restart} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**
fw-r-propagate: $\text{propagate } S \ S' \Longrightarrow cdcl_W\text{-merge-restart } S \ S' \mid$
fw-r-conflict: $\text{conflict } S \ T \Longrightarrow \text{full } cdcl_W\text{-bj } T \ U \Longrightarrow cdcl_W\text{-merge-restart } S \ U \mid$
fw-r-decide: $\text{decide } S \ S' \Longrightarrow cdcl_W\text{-merge-restart } S \ S' \mid$
fw-r-rf: $cdcl_W\text{-rf } S \ S' \Longrightarrow cdcl_W\text{-merge-restart } S \ S'$

lemma $cdcl_W\text{-merge-restart-cdcl}_W$:
assumes $cdcl_W\text{-merge-restart } S \ T$
shows $cdcl_W^{**} \ S \ T$
using *assms*
proof *induction*
case (*fw-r-conflict* $S \ T \ U$) **note** $\text{confl} = \text{this}(1)$ **and** $\text{bj} = \text{this}(2)$
have $cdcl_W \ S \ T$ **using** confl **by** (*simp* *add*: $cdcl_W.\text{intros } r\text{-into-rtranclp}$)
moreover
have $cdcl_W\text{-bj}^{**} \ T \ U$ **using** bj **unfolding** *full-def* **by** *auto*
then have $cdcl_W^{**} \ T \ U$ **by** (*metis* $cdcl_W\text{-o.bj mono-rtranclp other}$)
ultimately show $?case$ **by** *auto*
qed (*simp-all* *add*: $cdcl_W\text{-o.intros } cdcl_W.\text{intros } r\text{-into-rtranclp}$)

lemma $cdcl_W\text{-merge-restart-conflicting-true-or-no-step}$:
assumes $cdcl_W\text{-merge-restart } S \ T$
shows $\text{conflicting } T = \text{None} \vee \text{no-step } cdcl_W \ T$
using *assms*
proof *induction*
case (*fw-r-conflict* $S \ T \ U$) **note** $\text{confl} = \text{this}(1)$ **and** $n\text{-s} = \text{this}(2)$
{ fix $D \ V$
assume $cdcl_W \ U \ V$ **and** $\text{conflicting } U = \text{Some } D$
then have *False*
using $n\text{-s}$ **unfolding** *full-def*
by (*induction rule*: $cdcl_W\text{-all-rules-induct}$) (*auto* *dest*!: $cdcl_W\text{-bj.intros}$)
}
then show $?case$ **by** (*cases* $\text{conflicting } U$) *fastforce* +
qed (*auto simp add*: $cdcl_W\text{-rf.simps}$)

inductive $cdcl_W\text{-merge} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**
fw-propagate: $\text{propagate } S \ S' \Longrightarrow cdcl_W\text{-merge } S \ S' \mid$
fw-conflict: $\text{conflict } S \ T \Longrightarrow \text{full } cdcl_W\text{-bj } T \ U \Longrightarrow cdcl_W\text{-merge } S \ U \mid$
fw-decide: $\text{decide } S \ S' \Longrightarrow cdcl_W\text{-merge } S \ S' \mid$
fw-forget: $\text{forget } S \ S' \Longrightarrow cdcl_W\text{-merge } S \ S'$

lemma $cdcl_W\text{-merge-cdcl}_W\text{-merge-restart}$:
 $cdcl_W\text{-merge } S \ T \Longrightarrow cdcl_W\text{-merge-restart } S \ T$
by (*meson* $cdcl_W\text{-merge.cases } cdcl_W\text{-merge-restart.simps forget}$)

lemma $rtranclp\text{-cdcl}_W\text{-merge-rtranclp-cdcl}_W\text{-merge-restart}$:
 $cdcl_W\text{-merge}^{**} \ S \ T \Longrightarrow cdcl_W\text{-merge-restart}^{**} \ S \ T$
using $rtranclp\text{-mono}$ [of $cdcl_W\text{-merge } cdcl_W\text{-merge-restart}$] $cdcl_W\text{-merge-cdcl}_W\text{-merge-restart}$ **by** *blast*

lemma $cdcl_W\text{-merge-rtranclp-cdcl}_W$:
 $cdcl_W\text{-merge } S \ T \Longrightarrow cdcl_W^{**} \ S \ T$
using $cdcl_W\text{-merge-cdcl}_W\text{-merge-restart } cdcl_W\text{-merge-restart-cdcl}_W$ **by** *blast*

lemma $rtranclp\text{-cdcl}_W\text{-merge-rtranclp-cdcl}_W$:
 $cdcl_W\text{-merge}^{**} \ S \ T \Longrightarrow cdcl_W^{**} \ S \ T$

```

using rtrancpl-mono[of cdclW-merge cdclW**] cdclW-merge-rtrancpl-cdclW by auto

lemma cdclW-merge-is-cdclNOT-merged-bj-learn:
  assumes
    inv: cdclW-all-struct-inv S and
    cdclW:cdclW-merge S T
  shows cdclNOT-merged-bj-learn S T
    ∨ (no-step cdclW-merge T ∧ conflicting T ≠ None)
  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, None) and
    CL: C + {#L#} ∈# clauses S and
    M-C: M ⊨as CNot C and
    undef: undefined-lit (trail S) L and
    T: T ∼ cons-trail (Propagated L (C + {#L#})) S
  using propa by auto
  have propagateNOT S T
    apply (rule propagateNOT.propagateNOT[of - C L])
    using H CL T undef M-C by (auto simp: state-eqNOT-def state-eq-def clauses-def
      simp del: state-simp)
  then show ?case
    using cdclNOT-merged-bj-learn.intros(2) by blast
next
  case (fw-decide S T) note dec = this(1) and inv = this(2)
  then obtain L where
    undef-L: undefined-lit (trail S) L and
    atm-L: atm-of L ∈ atms-of-msu (init-clss S) and
    T: T ∼ cons-trail (Marked L (Suc (backtrack-lvl S)))
      (update-backtrack-lvl (Suc (backtrack-lvl S)) S)
    by auto
  have decideNOT S T
    apply (rule decideNOT.decideNOT)
    using undef-L apply simp
    using atm-L inv unfolding cdclW-all-struct-inv-def no-strange-atm-def clauses-def apply auto[]
    using T undef-L unfolding state-eq-def state-eqNOT-def by (auto simp: clauses-def)
  then show ?case using cdclNOT-merged-bj-learn-decideNOT by blast
next
  case (fw-forget S T) note rf = this(1) and inv = this(2)
  then obtain M N C U k where
    S: state S = (M, N, {#C#} + U, k, None) and
    ¬ M ⊨asm clauses S and
    C ∉ set (get-all-mark-of-propagated (trail S)) and
    C-init: C ∉# init-clss S and
    C-le: C ∈# learned-clss S and
    T: T ∼ remove-cls C S
    by auto
  have init-clss S ⊨pm C
    using inv C-le unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def
    by (meson mem-set-mset-iff true-clss-clss-in-imp-true-clss-cls)
  then have S-C: clauses S - replicate-mset (count (clauses S) C) C ⊨pm C
    using C-init C-le unfolding clauses-def by (simp add: Un-Diff)
  moreover have H: init-clss S + (learned-clss S - replicate-mset (count (learned-clss S) C) C)
    = init-clss S + learned-clss S - replicate-mset (count (learned-clss S) C) C

```

```

using C-le C-init by (metis clauses-def clauses-remove-cls diff-zero gr0I
  init-clss-remove-cls learned-clss-remove-cls plus-multiset.rep-eq replicate-mset-0
  semiring-normalization-rules(5))
have forgetNOT S T
apply (rule forgetNOT.forgetNOT)
  using S-C apply blast
  using S apply simp
  using  $\langle C \in \# \text{ learned-clss } S \rangle$  apply (simp add: clauses-def)
using T C-le C-init by (auto
  simp: state-eq-def Un-Diff state-eqNOT-def clauses-def ac-simps H
  simp del: state-simp)
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 = Some 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 rtrancpl-cdclW-bj-skip-or-resolve-backtrack unfolding full-def by meson
then show ?case
proof cases
  case no-bt
  then have conflicting U ≠ None
    using confl by (induction rule: rtrancpl-induct) auto
  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-rtrancpl[of skip-or-resolve cdclW] rtrancpl-skip-or-resolve-rtrancpl-cdclW
  by blast
then have cdclW-M-level-inv T'
  using rtrancpl-cdclW-consistent-inv ⟨cdclW-M-level-inv T⟩ by blast
then obtain M1 M2 i D L K where
  confl-T': conflicting T' = Some (D + {#L#}) and
  M1-M2: (Marked K (i+1) # M1, M2) ∈ set (get-all-marked-decomposition (trail T')) and
  get-level (trail T') L = backtrack-lvl T' and
  get-level (trail T') L = get-maximum-level (trail T') (D + {#L#}) and
  get-maximum-level (trail T') D = 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 None 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)  $\models_{as}$  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
        trail-reduce-trail-toNOT-add-learned-cls)[]
      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 add: trail-reduce-trail-toNOT-add-learned-cls)
    using ⟨init-clss T' + learned-clss S  $\models_{pm}$  D + {#L#}⟩ unfolding clauses-def apply simp
    apply (metis ⟨tl (trail U)  $\models_{as}$  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 $cdcl_{NOT}\text{-restart}$ where

$cdcl_{NOT}\text{-restart} \equiv \text{restart-ops.cdcl}_{NOT}\text{-raw-restart } cdcl_{NOT} \text{ restart}$

lemma $cdcl_W\text{-merge-restart-is-cdcl}_{NOT}\text{-merged-bj-learn-restart-no-step}$:

assumes

inv : $cdcl_W\text{-all-struct-inv } S$ and

$cdcl_W$: $cdcl_W\text{-merge-restart } S \ T$

shows $cdcl_{NOT}\text{-restart}^{**} \ S \ T \vee (\text{no-step } cdcl_W\text{-merge } T \wedge \text{conflicting } T \neq \text{None})$

proof –

consider

(fw) $cdcl_W\text{-merge } S \ T$

| (fw-r) $\text{restart } S \ T$

using $cdcl_W$ **by** (meson $cdcl_W\text{-merge-restart.simps } cdcl_W\text{-rf.cases fw-conflict fw-decide fw-forget}$
fw-propagate)

then show ?thesis

proof cases

case fw

then have IH: $cdcl_{NOT}\text{-merged-bj-learn } S \ T \vee (\text{no-step } cdcl_W\text{-merge } T \wedge \text{conflicting } T \neq \text{None})$

using inv $cdcl_W\text{-merge-is-cdcl}_{NOT}\text{-merged-bj-learn}$ **by** blast

have $invS$: $inv_{NOT} \ S$

using inv **unfolding** $cdcl_W\text{-all-struct-inv-def } cdcl_W\text{-M-level-inv-def}$ **by** auto

have ff2: $cdcl_{NOT}^{++} \ S \ T \longrightarrow cdcl_{NOT}^{**} \ S \ T$

by (meson $\text{trancpl-into-rtrancpl}$)

have ff3: $\text{no-dup } (\text{convert-trail-from-} W \ (\text{trail } S))$

using $invS$ **by** (simp add: comp-def)

have $cdcl_{NOT} \leq cdcl_{NOT}\text{-restart}$

by (auto simp: $\text{restart-ops.cdcl}_{NOT}\text{-raw-restart.simps}$)

then show ?thesis

using ff3 ff2 IH $cdcl_{NOT}\text{-merged-bj-learn-is-trancpl-cdcl}_{NOT}$

$\text{rtrancpl-mono}[of \ cdcl_{NOT} \ cdcl_{NOT}\text{-restart}] \ invS \ \text{predicate2D}$ **by** blast

next
case *fw-r*
then show *?thesis* **by** (*blast intro: restart-ops.cdcl_{NOT}-raw-restart.intros*)
qed
qed

abbreviation $\mu_{FW} :: 'st \Rightarrow nat$ **where**

$\mu_{FW} S \equiv (if\ no\text{-}step\ cdcl_W\text{-}merge\ S\ then\ 0\ else\ 1 + \mu_{CDCL}'\text{-}merged\ (set\text{-}mset\ (init\text{-}clss\ S))\ S)$

lemma *cdcl_W-merge- μ_{FW} -decreasing*:

assumes

inv: cdcl_W-all-struct-inv S and

fw: cdcl_W-merge S T

shows $\mu_{FW} T < \mu_{FW} S$

proof –

let *?A = init-clss S*

have *atm-clauses: atms-of-msu (clauses S) \subseteq atms-of-msu ?A*

using *inv unfolding cdcl_W-all-struct-inv-def no-strange-atm-def clauses-def* **by** *auto*

have *atm-trail: atm-of ' lits-of (trail S) \subseteq atms-of-msu ?A*

using *inv unfolding cdcl_W-all-struct-inv-def no-strange-atm-def clauses-def* **by** *auto*

have *n-d: no-dup (trail S)*

using *inv unfolding cdcl_W-all-struct-inv-def* **by** (*auto simp: cdcl_W-M-level-inv-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-merge-restart-cdcl_W[of S T] inv rtranclp-cdcl_W-init-clss*

unfolding *cdcl_W-all-struct-inv-def*

by (*meson cdcl_W-merge.simps cdcl_W-merge-restart.simps cdcl_W-rf.simps fw*)

consider

(merged) cdcl_{NOT}-merged-bj-learn S T

| *(n-s) no-step cdcl_W-merge T*

using *cdcl_W-merge-is-cdcl_{NOT}-merged-bj-learn inv fw* **by** *blast*

then show *?thesis*

proof *cases*

case *merged*

then show *?thesis*

using *cdcl_{NOT}-decreasing-measure'[OF - - atm-clauses] atm-trail n-d*

by (*auto split: split-if simp: comp-def*)

next

case *n-s*

then show *?thesis* **by** *simp*

qed

qed

lemma *wf-cdcl_W-merge: wf {(T, S). cdcl_W-all-struct-inv S \wedge cdcl_W-merge S T}*

apply (*rule wfP-if-measure[of - - μ_{FW}]*)

using *cdcl_W-merge- μ_{FW} -decreasing* **by** *blast*

lemma *cdcl_W-all-struct-inv-tranclp-cdcl_W-merge-tranclp-cdcl_W-merge-cdcl_W-all-struct-inv:*

assumes

inv: cdcl_W-all-struct-inv b

cdcl_W-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 trancpl-into-rtrancpl[OF st] cdclW-merge-rtrancpl-cdclW
  assms(1) rtrancpl-cdclW-all-struct-inv-inv rtrancpl-mono[of cdclW-merge cdclW**] by fastforce
then have ( $\lambda S T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T$ )++ c d
  using fw by auto
then show ?case using IH by auto
qed

lemma wf-trancpl-cdclW-merge: wf {(T, S). cdclW-all-struct-inv S  $\wedge$  cdclW-merge++ S T}
  using wf-trancpl[OF wf-cdclW-merge]
  apply (rule wf-subset)
  by (auto simp: trancpl-set-trancpl
    cdclW-all-struct-inv-trancpl-cdclW-merge-trancpl-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 rtrancpl-cdclW-conflicting-true-cdclW-merge-restart:
  assumes cdclW** S V and inv: cdclW-M-level-inv S and conflicting S = None
  shows (cdclW-merge-restart** S V  $\wedge$  conflicting V = None)
     $\vee (\exists T U. \text{cdcl}_W\text{-merge-restart** } S T \wedge \text{conflicting } V \neq \text{None} \wedge \text{conflict } T U \wedge \text{cdcl}_W\text{-bj** } U V)$ 
  using assms
proof induction
  case base
  then show ?case by simp
next
case (step U V) note st = this(1) and cdclW = this(2) and IH = this(3)[OF this(4–)] and
  confl[simp] = this(5) and inv = this(4)
from cdclW
show ?case
  proof (cases)
  case propagate
  moreover then have conflicting U = None
    by auto
  moreover have conflicting V = None
    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 = None
    by auto
  moreover have conflicting V  $\neq$  None
    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 = None
by auto
ultimately show ?thesis using IH cdclW-merge-restart.fw-r-decide[of U V] by auto
next
case bj
moreover {
assume skip-or-resolve U V
have f1: cdclW-bj++ U V
by (simp add: local.bj tranclp.r-into-trancl)
obtain T T' :: 'st where
f2: cdclW-merge-restart** S U
  ∨ cdclW-merge-restart** S T ∧ conflicting U ≠ None
  ∧ conflict T T' ∧ cdclW-bj** T' U
using IH confl by blast
then have ?thesis
proof -
have conflicting V ≠ None ∧ conflicting U ≠ None
using ⟨skip-or-resolve U V⟩ by auto
then show ?thesis
by (metis (no-types) IH f1 rtranclp-trans tranclp-into-rtranclp)
qed
}
moreover {
assume backtrack U V
then have conflicting U ≠ None by auto
then obtain T T' where
cdclW-merge-restart** S T and
conflicting U ≠ None and
conflict T T' and
cdclW-bj** T' U
using IH confl by meson
have invU: cdclW-M-level-inv U
using inv rtranclp-cdclW-consistent-inv step.hyps(1) by blast
then have conflicting V = None
using ⟨backtrack U V⟩ inv by (auto elim: backtrack-levE
simp: cdclW-M-level-inv-decomp)
have full cdclW-bj T' V
apply (rule rtranclp-fullI[of cdclW-bj T' U V])
using ⟨cdclW-bj** T' U⟩ apply fast
using ⟨backtrack U V⟩ backtrack-is-full1-cdclW-bj invU unfolding full1-def full-def
by blast
then have ?thesis
using cdclW-merge-restart.fw-r-conflict[of T T' V] ⟨conflict T T'⟩
⟨cdclW-merge-restart** S T⟩ ⟨conflicting V = None⟩ by auto
}
ultimately show ?thesis by (auto simp: cdclW-bj.simps)
qed
next
case rf

```

moreover then have *conflicting* $U = \text{None}$ and *conflicting* $V = \text{None}$
 by (auto simp: *cdcl_W-rf.simps*)
 ultimately show ?thesis using *IH cdcl_W-merge-restart.fw-r-rf*[of $U\ V$] by auto
 qed
 qed

lemma *no-step-cdcl_W-no-step-cdcl_W-merge-restart*: *no-step cdcl_W S* \implies *no-step cdcl_W-merge-restart S*
 by (auto simp: *cdcl_W.simps cdcl_W-merge-restart.simps cdcl_W-o.simps cdcl_W-bj.simps*)

lemma *no-step-cdcl_W-merge-restart-no-step-cdcl_W*:

assumes
 conflicting S = None and
 cdcl_W-M-level-inv S and
 no-step cdcl_W-merge-restart S
 shows *no-step cdcl_W S*

proof –

{ fix S'
 assume *conflict S S'*
 then have *cdcl_W S S'* using *cdcl_W.conflict* by auto
 then have *cdcl_W-M-level-inv S'*
 using *assms(2) cdcl_W-consistent-inv* by blast
 then obtain S'' where *full cdcl_W-bj S' S''*
 using *cdcl_W-bj-exists-normal-form*[of S'] by auto
 then have *False*
 using $\langle \text{conflict } S\ S' \rangle$ *assms(3) fw-r-conflict* by blast
 }

then show ?thesis

using *assms unfolding cdcl_W.simps cdcl_W-merge-restart.simps cdcl_W-o.simps cdcl_W-bj.simps*
 by fastforce

qed

lemma *rtrancpl-cdcl_W-merge-restart-no-step-cdcl_W-bj*:

assumes
 *cdcl_W-merge-restart** S T* and
 conflicting S = None
 shows *no-step cdcl_W-bj T*
 using *assms*
 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* $\neq \text{None}$, we cannot say anything.

Remark that this theorem does not say anything about well-foundedness: even if you know that one relation is well-founded, it only states that the normal forms are shared.

lemma *conflicting-true-full-cdcl_W-iff-full-cdcl_W-merge*:

assumes *conf*: *conflicting S = None* and *lev*: *cdcl_W-M-level-inv S*
 shows *full cdcl_W S V* \longleftrightarrow *full cdcl_W-merge-restart S V*

proof

assume *full*: *full cdcl_W-merge-restart S V*
 then have *st*: *cdcl_W** S V*
 using *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 cdclW-merge-restart V
  using full unfolding full-def by auto
have n-s-bj: no-step cdclW-bj V
  using rtrancp-cdclW-merge-restart-no-step-cdclW-bj confl full unfolding full-def by auto
have  $\bigwedge S'. \text{conflict } V S' \implies \text{cdcl}_W\text{-M-level-inv } S'$ 
  using cdclW.conflict cdclW-consistent-inv lev rtrancp-cdclW-consistent-inv st by blast
then have  $\bigwedge S'. \text{conflict } V S' \implies \text{False}$ 
  using n-s n-s-bj cdclW-bj-exists-normal-form cdclW-merge-restart.simps by meson
then have n-s-cdclW: no-step cdclW V
  using n-s n-s-bj by (auto simp: cdclW.simps cdclW-o.simps cdclW-merge-restart.simps)
then show full cdclW S V using st unfolding full-def by auto
next
assume full: full cdclW S V
have no-step cdclW-merge-restart V
  using full no-step-cdclW-no-step-cdclW-merge-restart unfolding full-def by blast
moreover
consider
  (fw) cdclW-merge-restart** S V and conflicting V = None
| (bj) T U where
  cdclW-merge-restart** S T and
  conflicting V  $\neq$  None and
  conflict T U and
  cdclW-bj** U V
  using full rtrancp-cdclW-conflicting-true-cdclW-merge-restart confl lev unfolding full-def
  by meson
then have cdclW-merge-restart** S V
proof cases
  case fw
  then show ?thesis by fast
next
  case (bj T U)
  have no-step cdclW-bj V
    using full unfolding full-def by (meson cdclW-o.bj other)
  then have full cdclW-bj U V
    using  $\langle \text{cdcl}_W\text{-bj}^{**} U V \rangle$  unfolding full-def by auto
  then have cdclW-merge-restart T V
    using  $\langle \text{conflict } T U \rangle$  cdclW-merge-restart.fw-r-conflict by blast
  then show ?thesis using  $\langle \text{cdcl}_W\text{-merge-restart}^{**} S T \rangle$  by auto
qed
ultimately show full cdclW-merge-restart S V unfolding full-def by fast
qed

```

lemma *init-state-true-full-cdcl_W-iff-full-cdcl_W-merge:*
shows full cdcl_W (init-state N) V \longleftrightarrow full cdcl_W-merge-restart (init-state N) V
by (rule conflicting-true-full-cdcl_W-iff-full-cdcl_W-merge) auto

19.5 FW with strategy

19.5.1 The intermediate step

inductive cdcl_W-s' :: 'st \Rightarrow 'st \Rightarrow bool **where**
conflict': full1 cdcl_W-cp S S' \implies cdcl_W-s' S S' |
decide': decide S S' \implies no-step cdcl_W-cp S \implies full cdcl_W-cp S' S'' \implies cdcl_W-s' S S'' |
bj': full1 cdcl_W-bj S S' \implies no-step cdcl_W-cp S \implies full cdcl_W-cp S' S'' \implies cdcl_W-s' S S''

inductive-cases $cdcl_W-s'E: cdcl_W-s' S T$

lemma $rtrancpl-cdcl_W-bj-full1-cdclp-cdcl_W-stgy:$

$cdcl_W-bj^{**} S S' \implies full\ cdcl_W-cp\ S' S'' \implies cdcl_W-stgy^{**} S S''$

proof (*induction rule: converse-rtrancpl-induct*)

case *base*

then show *?case* **by** (*metis* $cdcl_W-stgy.conflict'$ *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* $cdcl_W-cp\ T$

using bj **by** (*auto* *simp* *add: cdcl_W-bj.simps*)

consider

(U) $U = S'$

| (U') U' **where** $cdcl_W-bj\ U\ U'$ **and** $cdcl_W-bj^{**}\ U'\ S'$

using st **by** (*metis* *converse-rtrancplE*)

then show *?case*

proof *cases*

case U

then show *?thesis*

using (*no-step* $cdcl_W-cp\ T$) $cdcl_W-o.bj\ local.bj\ other'$ *step.prem*s **by** (*meson* *r-into-rtrancpl*)

next

case U' **note** $U' = this(1)$

have *no-step* $cdcl_W-cp\ U$

using U' **by** (*fastforce* *simp: cdcl_W-cp.simps* *cdcl_W-bj.simps*)

then have *full* $cdcl_W-cp\ U\ U$

by (*simp* *add: full-unfold*)

then have $cdcl_W-stgy\ T\ U$

using (*no-step* $cdcl_W-cp\ T$) $cdcl_W-stgy.simps\ local.bj\ cdcl_W-o.bj$ **by** *meson*

then show *?thesis* **using** IH **by** *auto*

qed

qed

lemma $cdcl_W-s'-is-rtrancpl-cdcl_W-stgy:$

$cdcl_W-s' S T \implies cdcl_W-stgy^{**} S T$

apply (*induction rule: cdcl_W-s'.induct*)

apply (*auto* *intro: cdcl_W-stgy.intros*)[]

apply (*meson* *decide* *other' r-into-rtrancpl*)

by (*metis* *full1-def* *rtrancpl-cdcl_W-bj-full1-cdclp-cdcl_W-stgy* *trancpl-into-rtrancpl*)

lemma $cdcl_W-cp-cdcl_W-bj-bissimulation:$

assumes

full $cdcl_W-cp\ T\ U$ **and**

$cdcl_W-bj^{**}\ T\ T'$ **and**

$cdcl_W-all-struct-inv\ T$ **and**

no-step $cdcl_W-bj\ T'$

shows *full* $cdcl_W-cp\ T'\ U$

$\vee (\exists U' U''. full\ cdcl_W-cp\ T'\ U'' \wedge full1\ cdcl_W-bj\ U\ U' \wedge full\ cdcl_W-cp\ U'\ U'' \wedge cdcl_W-s'^{**}\ U\ U'')$

using *assms*(2,1,3,4)

proof (*induction rule: 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 $cdcl_W^{**}\ T\ T''$

```

  by (metis (no-types, lifting) cdclW-o.bj local.bj mono-rtrancp[of cdclW-bj cdclW T T'] other
    st rtrancp.rtrancp-into-rtrancp)
then have inv-T'': cdclW-all-struct-inv T''
  using inv rtrancp-cdclW-all-struct-inv-inv by blast
have cdclW-bj++ T T''
  using local.bj st by auto
have full1 cdclW-bj T T''
  by (metis <cdclW-bj++ T T''> full1-def step.prem(3))
then have T = U
proof -
  obtain Z where cdclW-bj T Z
    by (meson trancpD <cdclW-bj++ T T''>)
  { assume cdclW-cp++ T U
    then obtain Z' where cdclW-cp T Z'
      by (meson trancpD)
    then have False
      using <cdclW-bj T Z> by (fastforce simp: cdclW-bj.simps cdclW-cp.simps)
  }
  then show ?thesis
    using full unfolding full-def rtrancp-unfold by blast
qed
obtain U'' where full cdclW-cp T'' U''
  using cdclW-cp-normalized-element-all-inv inv-T'' by blast
moreover then have cdclW-stgy** U U''
  by (metis <T = U> <cdclW-bj++ T T''> 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 full 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 show ?case
  using <full1 cdclW-bj T T''> <full cdclW-cp T'' U''> unfolding <T = U> by blast
qed

```

lemma *cdcl_W-cp-cdcl_W-bj-bissimulation'*:

```

assumes
  full cdclW-cp T U and
  cdclW-bj** T T' and
  cdclW-all-struct-inv T and
  no-step cdclW-bj T'
shows full cdclW-cp T' U
  ∨ (∃ U'. full1 cdclW-bj U U' ∧ (∀ U''. full cdclW-cp U' U'' → full cdclW-cp T' U''
    ∧ cdclW-s** U U'))
using assms(2,1,3,4)
proof (induction rule: rtrancp-induct)
  case base
  then show ?case by blast
next
  case (step T' T'') note st = this(1) and bj = this(2) and IH = this(3)[OF this(4,5)] and

```



```

  full = this(4) and inv = this(5)
have cdclW** T T''
  by (metis (no-types, lifting) cdclW-o.bj local.bj mono-rtrancp[of cdclW-bj cdclW T T''] other st
    rtrancp.rtrancp-into-rtrancp)
then have inv-T'': cdclW-all-struct-inv T''
  using inv rtrancp-cdclW-all-struct-inv-inv by blast
have cdclW-bj++ T T''
  using local.bj st by auto
have full1 cdclW-bj T T''
  by (metis ⟨cdclW-bj++ T T'⟩ full1-def step.prem(3))
then have T = U
  proof -
    obtain Z where cdclW-bj T Z
      by (meson trancpD ⟨cdclW-bj++ T T'⟩)
    { assume cdclW-cp++ T U
      then obtain Z' where cdclW-cp T Z'
        by (meson trancpD)
      then have False
        using ⟨cdclW-bj T Z⟩ by (fastforce simp: cdclW-bj.simps cdclW-cp.simps)
    }
    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
  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 ≠ None)*
using *assms(1)*
proof *induction*

```

case base
then show ?case by simp
next
case (step T V) note st = this(1) and o = this(2) and IH = this(3)
from o show ?case
  proof cases
    case conflict'
      then have f2: cdclW-s' T V
        using cdclW-s'.conflict' by blast
      obtain ss :: 'st where
        f3: S = T  $\vee$  cdclW-stgy** S ss  $\wedge$  cdclW-stgy ss T
        by (metis (full-types) rtranclp.simps st)
      obtain ssa :: 'st where
        cdclW-cp T ssa
        using conflict' by (metis (no-types) full1-def tranclpD)
      then have S = T
        using f3 by (metis (no-types) cdclW-stgy.simps full-def full1-def)
      then show ?thesis
        using f2 by blast
  next
  case (other' U) note o = this(1) and n-s = this(2) and full = this(3)
  then show ?thesis
    using o
    proof (cases rule: cdclW-o-rule-cases)
      case decide
        then have cdclW-s'** S T
          using IH by auto
        then show ?thesis
          by (meson decide decide' full n-s rtranclp.rtrancl-into-rtrancl)
    next
    case backtrack
    consider
      (s') cdclW-s'** S T
      | (bj) S' where cdclW-s'** S S' and cdclW-bj++ S' T and conflicting T  $\neq$  None
      using IH by blast
    then show ?thesis
      proof cases
        case s'
        moreover
          have cdclW-M-level-inv T
            using inv local.step(1) rtranclp-cdclW-stgy-consistent-inv by auto
          then have full1 cdclW-bj T U
            using backtrack-is-full1-cdclW-bj backtrack by blast
          then have cdclW-s' T V
            using full bj' n-s by blast
          ultimately show ?thesis by auto
        next
        case (bj S') note S-S' = this(1) and bj-T = this(2)
        have no-step cdclW-cp S'
          using bj-T by (fastforce simp: cdclW-cp.simps cdclW-bj.simps dest!: tranclpD)
        moreover
          have cdclW-M-level-inv T
            using inv local.step(1) rtranclp-cdclW-stgy-consistent-inv by auto
          then have full1 cdclW-bj T U
            using backtrack-is-full1-cdclW-bj backtrack by blast

```

```

    then have full1 cdclW-bj S' U
      using bj-T unfolding full1-def by fastforce
    ultimately have cdclW-s' S' V using full by (simp add: bj')
    then show ?thesis using S-S' by auto
  qed
next
case skip
then have [simp]:  $U = V$ 
  using full converse-rtrancpE unfolding full-def by fastforce

consider
  (s') cdclW-s'^** S T
  | (bj) S' where cdclW-s'^** S S' and cdclW-bj^{++} S' T and conflicting T ≠ None
  using IH by blast
then show ?thesis
proof cases
case s'
have cdclW-bj^{++} T V
  using skip by force
moreover have conflicting V ≠ None
  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 cdclW-bj^{++} S' V
  using skip bj-T by (metis ⟨U = V⟩ cdclW-bj.skip trancp.simps)

  moreover have conflicting V ≠ None
    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') cdclW-s'^** S T
  | (bj) S' where cdclW-s'^** S S' and cdclW-bj^{++} S' T and conflicting T ≠ None
  using IH by blast
then show ?thesis
proof cases
case s'
have cdclW-bj^{++} T V
  using resolve by force
moreover have conflicting V ≠ None
  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 cdclW-bj^{++} S' V
  using resolve bj-T by (metis ⟨U = V⟩ cdclW-bj.resolve trancp.simps)
moreover have conflicting V ≠ None
  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  $\langle$ cdclW-o S S' $\rangle$  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) ∧ 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 rtrancpl-cdclW-bj-full1-cdclp-cdclW-stgy
      rtrancpl-cdclW-stgy-rtrancpl-cdclW rtrancpl-into-trancpl2)
qed

lemma trancpl-cdclW-s'-trancpl-cdclW:
  cdclW-s'++ S S' ⇒ cdclW++ S S'
  apply (induct rule: trancpl.induct)
  using cdclW-s'-trancpl-cdclW apply blast
  by (meson cdclW-s'-trancpl-cdclW trancpl-trans)

lemma rtrancpl-cdclW-s'-rtrancpl-cdclW:
  cdclW-s'** S S' ⇒ cdclW** S S'
  using rtrancpl-unfold[of cdclW-s' S S'] trancpl-cdclW-s'-trancpl-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 ⇔ full cdclW-s' S T (is ?S ⇔ ?S')
proof
  assume ?S'
  then have cdclW** S T
    using rtrancpl-cdclW-s'-rtrancpl-cdclW[of S T] unfolding full-def by blast
  then have inv': cdclW-all-struct-inv T
    using rtrancpl-cdclW-all-struct-inv-inv inv by blast
  have cdclW-stgy** S T
    using ⟨?S'⟩ unfolding full-def
    using cdclW-s'-is-rtrancpl-cdclW-stgy rtrancpl-mono[of cdclW-s' cdclW-stgy**] by auto
  then show ?S
    using ⟨?S'⟩ inv' cdclW-stgy-cdclW-s'-connected' unfolding full-def by blast
next
  assume ?S
  then have inv-T: cdclW-all-struct-inv T
    by (metis asms full-def rtrancpl-cdclW-all-struct-inv-inv rtrancpl-cdclW-stgy-rtrancpl-cdclW)
  consider
    (s') cdclW-s'** S T
  | (st) S' where cdclW-s'** S S' and cdclW-bj++ S' T and conflicting T ≠ None
  using rtrancpl-cdclW-stgy-connected-to-rtrancpl-cdclW-s'[of S T] inv ⟨?S⟩
  unfolding full-def cdclW-all-struct-inv-def
  by blast
  then show ?S'
  proof cases
    case s'
    then show ?thesis
      by (metis ⟨full cdclW-stgy S T⟩ inv-T cdclW-all-struct-inv-def cdclW-s'.sims
        cdclW-stgy.conflict' cdclW-then-exists-cdclW-stgy-step full-def
        n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o)
  next
    case (st S')
    have full cdclW-cp T T
      using option-full-cdclW-cp st(3) by blast
    moreover

```

```

have n-s: no-step cdclW-bj T
  by (metis  $\langle \text{full } \text{cdcl}_W\text{-stgy } S \ T \rangle \text{ bj inv-}T \text{ cdcl}_W\text{-all-struct-inv-def}$ 
     $\text{cdcl}_W\text{-then-exists-cdcl}_W\text{-stgy-step full-def}$ )
then have full1 cdclW-bj S' T
  using st(2) unfolding full1-def by blast
moreover have no-step cdclW-cp S'
  using st(2) by (fastforce dest!: tranclpD simp: cdclW-cp.simps cdclW-bj.simps)
ultimately have cdclW-s' S' T
  using cdclW-s'.bj'[of S' T T] by blast
then have cdclW-sl* S T
  using st(1) by auto
moreover have no-step cdclW-s' T
  using inv-T by (metis  $\langle \text{full } \text{cdcl}_W\text{-cp } T \ T \rangle \langle \text{full } \text{cdcl}_W\text{-stgy } S \ T \rangle \text{ cdcl}_W\text{-all-struct-inv-def}$ 
     $\text{cdcl}_W\text{-then-exists-cdcl}_W\text{-stgy-step full-def n-step-cdcl}_W\text{-stgy-iff-no-step-cdcl}_W\text{-cl-cdcl}_W\text{-o}$ )
ultimately show ?thesis
  unfolding full-def by blast
qed
qed

```

lemma *conflict-step-cdcl_W-stgy-step:*

```

assumes
  conflict S T
  cdclW-all-struct-inv S
shows  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$ 
proof –
obtain U where full cdclW-cp S U
  using cdclW-cp-normalized-element-all-inv assms by blast
then have full1 cdclW-cp S U
  by (metis cdclW-cp.conflict' assms(1) full-unfold)
then show ?thesis using cdclW-stgy.conflict' by blast
qed

```

lemma *decide-step-cdcl_W-stgy-step:*

```

assumes
  decide S T
  cdclW-all-struct-inv S
shows  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$ 
proof –
obtain U where full cdclW-cp T U
  using cdclW-cp-normalized-element-all-inv by (meson assms(1) assms(2) cdclW-all-struct-inv-inv
     $\text{cdcl}_W\text{-cp-normalized-element-all-inv decide other}$ )
then show ?thesis
  by (metis assms cdclW-cp-normalized-element-all-inv cdclW-stgy.conflict' decide full-unfold
    other')
qed

```

lemma *rtranclp-cdcl_W-cp-conflicting-Some:*

```

cdclW-cp** S T  $\implies$  conflicting S = Some 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 cdclW-bj T U  $\implies$  cdclW-merge-cp S U |
propagate'[intro]: propagate++ S S'  $\implies$  cdclW-merge-cp S S'

```

lemma *cdcl_W-merge-restart-cases[consumes 1, case-names conflict propagate]:*


```

assumes
  cdclW-merge-cp S U and
   $\bigwedge T. \text{conflict } S \ T \implies \text{full } \text{cdcl}_W\text{-bj } T \ U \implies P \text{ and}$ 
  propagate++ S U  $\implies$  P
shows P
using assms unfolding cdclW-merge-cp.simps by auto

lemma cdclW-merge-cp-tranclp-cdclW-merge:
  cdclW-merge-cp S T  $\implies$  cdclW-merge++ S T
apply (induction rule: cdclW-merge-cp.induct)
  using cdclW-merge.simps apply auto[1]
using tranclp-mono[of propagate cdclW-merge] fw-propagate by blast

lemma rtranclp-cdclW-merge-cp-rtranclp-cdclW:
  cdclW-merge-cp** S T  $\implies$  cdclW** S T
apply (induction rule: rtranclp-induct)
apply simp
unfolding cdclW-merge-cp.simps by (meson cdclW-merge-restart-cdclW fw-r-conflict
  rtranclp-propagate-is-rtranclp-cdclW rtranclp-trans tranclp-into-rtranclp)

lemma full1-cdclW-bj-no-step-cdclW-bj:
  full1 cdclW-bj S T  $\implies$  no-step cdclW-cp S
by (metis rtranclp-unfold cdclW-cp-conflicting-not-empty option.exhaust full1-def
  rtranclp-cdclW-merge-restart-no-step-cdclW-bj tranclpD)

inductive cdclW-s'-without-decide where
  conflict'-without-decide[intro]: full1 cdclW-cp S S'  $\implies$  cdclW-s'-without-decide S S' |
  bj'-without-decide[intro]: full1 cdclW-bj S S'  $\implies$  no-step cdclW-cp S  $\implies$  full cdclW-cp S' S''
   $\implies$  cdclW-s'-without-decide S S''

lemma rtranclp-cdclW-s'-without-decide-rtranclp-cdclW:
  cdclW-s'-without-decide** S T  $\implies$  cdclW** S T
apply (induction rule: rtranclp-induct)
apply simp
by (meson cdclW-s'.simps cdclW-s'-tranclp-cdclW cdclW-s'-without-decide.simps
  rtranclp-tranclp-tranclp tranclp-into-rtranclp)

lemma rtranclp-cdclW-s'-without-decide-rtranclp-cdclW-s':
  cdclW-s'-without-decide** S T  $\implies$  cdclW-s'*** S T
proof (induction rule: rtranclp-induct)
  case base
  then show ?case by simp
next
  case (step y z) note a2 = this(2) and a1 = this(3)
  have cdclW-s' y z
  using a2 by (metis (no-types) bj' cdclW-s'.conflict' cdclW-s'-without-decide.cases)
  then show cdclW-s'*** S z
  using a1 by (meson r-into-rtranclp rtranclp-trans)
qed

lemma rtranclp-cdclW-merge-cp-is-rtranclp-cdclW-s'-without-decide:
assumes
  cdclW-merge-cp** S V
  conflicting S = None
shows

```

```

(cdcW-s'-without-decide** S V)
∨ (∃ T. cdcW-s'-without-decide** S T ∧ propagate++ T V)
∨ (∃ T U. cdcW-s'-without-decide** S T ∧ full1 cdcW-bj T U ∧ propagate** U V)
using assms
proof (induction rule: rtrancp-induct)
  case base
  then show ?case by simp
next
case (step U V) note st = this(1) and cp = this(2) and IH = this(3)[OF this(4)]
from cp show ?case
  proof (cases rule: cdcW-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 cdcW-cp U U'
  by (simp add: conflict-is-full1-cdcW-cp local.conflict(1))
  consider
    (s') cdcW-s'-without-decide** S U
  | (propa) T' where cdcW-s'-without-decide** S T' and propagate++ T' U
  | (bj-prop) T' T'' where
    cdcW-s'-without-decide** S T' and
    full1 cdcW-bj T' T'' and
    propagate** T'' U
  using IH by blast
  then show ?thesis
  proof cases
    case s'
    have cdcW-s'-without-decide U U'
    using full1-U-U' conflict'-without-decide by blast
    then have cdcW-s'-without-decide** S U'
    using ⟨cdcW-s'-without-decide** S U⟩ by auto
    moreover have U' = V ∨ full1 cdcW-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 cdcW-cp T' U'
  using rtrancp-mono[of propagate cdcW-cp] T'-U cdcW-cp.propagate' full1-U-U'
  rtrancp-full1I[of cdcW-cp T'] by (metis (full-types) predicate2D predicate2I
    trancp-into-rtrancp)
  have cdcW-s'-without-decide** S U'
  using ⟨full1 cdcW-cp T' U'⟩ conflict'-without-decide s' by force
  have full1 cdcW-bj U' V ∨ V = U'
  by (metis (lifting) full-unfold local.bj)
  then show ?thesis
  using ⟨cdcW-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 cdcW-cp T'
  using bj-T' full1-cdcW-bj-no-step-cdcW-bj by blast
  moreover have full1 cdcW-cp T'' U'
  using rtrancp-mono[of propagate cdcW-cp] T''-U cdcW-cp.propagate' full1-U-U'
  rtrancp-full1I[of cdcW-cp T''] by blast
  ultimately have cdcW-s'-without-decide T' U'

```

```

    using bj'-without-decide[of T' T'' U] bj-T' by (simp add: full-unfold)
  then have cdclW-s'-without-decide** S U'
    using s' rtrancp.intros(2)[of - S T' U] by blast
  then show ?thesis
    by (metis full-unfold local.bj rtrancp.rtrancp-refl)
qed
qed
qed

lemma rtrancp-cdclW-s'-without-decide-is-rtrancp-cdclW-merge-cp:
  assumes
    cdclW-s'-without-decide** S V and
    confl: conflicting S = None
  shows
    (cdclW-merge-cp** S V ∧ conflicting V = None)
    ∨ (cdclW-merge-cp** S V ∧ conflicting V ≠ None ∧ no-step cdclW-cp V ∧ no-step cdclW-bj V)
    ∨ (∃ T. cdclW-merge-cp** S T ∧ conflict T V)
  using assms(1)
proof (induction)
  case base
  then show ?case using confl by auto
next
  case (step U V) note st = this(1) and s = this(2) and IH = this(3)
  from s show ?case
  proof (cases rule: cdclW-s'-without-decide.cases)
    case conflict'-without-decide
    then have rt: cdclW-cp++ U V unfolding full1-def by fast
    then have conflicting U = None
      using trancp-cdclW-cp-propagate-with-conflict-or-not[of U V]
      conflict by (auto dest!: trancpD simp: rtrancp-unfold)
    then have cdclW-merge-cp** S U using IH by auto
    consider
      (propa) propagate++ U V
      | (confl') conflict U V
      | (propa-confl') U' where propagate++ U U' conflict U' V
    using trancp-cdclW-cp-propagate-with-conflict-or-not[OF rt] unfolding rtrancp-unfold
    by fastforce
  then show ?thesis
  proof cases
    case propa
    then have cdclW-merge-cp U V
      by auto
    moreover have conflicting V = None
      using propa unfolding trancp-unfold-end by auto
    ultimately show ?thesis using ⟨cdclW-merge-cp** S U⟩ by force
  next
    case confl'
    then show ?thesis using ⟨cdclW-merge-cp** S U⟩ by auto
  next
    case propa-confl' note propa = this(1) and confl' = this(2)
    then have cdclW-merge-cp U U' by auto
    then have cdclW-merge-cp** S U' using ⟨cdclW-merge-cp** S U⟩ by auto
    then show ?thesis using ⟨cdclW-merge-cp** S U⟩ confl' by auto
  qed
qed

```

```

next
case (bj'-without-decide U') note full-bj = this(1) and cp = this(3)
then have conflicting U ≠ None
  using full-bj unfolding full1-def by (fastforce dest!: tranclpD simp: cdclW-bj.simps)
with IH obtain T where
  S-T: cdclW-merge-cp** S T and T-U: conflict T U
  using full-bj unfolding full1-def by (blast dest: tranclpD)
then have cdclW-merge-cp T U'
  using cdclW-merge-cp.conflict'[of T U U'] full-bj by (simp add: full-unfold)
then have S-U': cdclW-merge-cp** S U' using S-T by auto
consider
  (n-s) U' = V
  | (propa) propagate++ U' V
  | (confl') conflict U' V
  | (propa-confl') U'' where propagate++ U' U'' conflict U'' V
  using tranclp-cdclW-cp-propagate-with-conflict-or-not cp
  unfolding rtranclp-unfold full-def by metis
then show ?thesis
proof cases
case propa
  then have cdclW-merge-cp U' V by auto
  moreover have conflicting V = None
    using propa unfolding tranclp-unfold-end by auto
  ultimately show ?thesis using S-U' by force
next
case confl'
  then show ?thesis using S-U' by auto
next
case propa-confl' note propa = this(1) and confl = this(2)
  have cdclW-merge-cp U' U'' using propa by auto
  then show ?thesis using S-U' confl by (meson rtranclp.rtrancl-into-rtrancl)
next
case n-s
  then show ?thesis
    using S-U' apply (cases conflicting V = None)
    using full-bj apply simp
    by (metis cp full-def full-unfold full-bj)
qed
qed
qed

```

lemma *no-step-cdclW-s'-no-ste-cdclW-merge-cp:*
assumes
cdclW-all-struct-inv S
conflicting S = None
no-step cdclW-s' S
shows *no-step cdclW-merge-cp S*
using *assms* **apply** (auto simp: cdclW-s'.simps cdclW-merge-cp.simps)
using *conflict-is-full1-cdclW-cp* **apply** blast
using *cdclW-cp-normalized-element-all-inv cdclW-cp.propagate'* **by** (metis cdclW-cp.propagate'
full-unfold tranclpD)

The *no-step decide S* is needed, since *cdclW-merge-cp* is *cdclW-s'* without *decide*.

lemma *conflicting-true-no-step-cdclW-merge-cp-no-step-s'-without-decide:*
assumes

confl: conflicting $S = \text{None}$ and
inv: $\text{cdcl}_W\text{-M-level-inv } S$ and
n-s: $\text{no-step cdcl}_W\text{-merge-cp } S$
shows $\text{no-step cdcl}_W\text{-s'-without-decide } S$
proof (rule *ccontr*)
assume $\neg \text{no-step cdcl}_W\text{-s'-without-decide } S$
then obtain T **where**
cdcl_W: $\text{cdcl}_W\text{-s'-without-decide } S \ T$
by *auto*
then have $\text{inv-T: cdcl}_W\text{-M-level-inv } T$
using $\text{rtrancp-cdcl}_W\text{-s'-without-decide-rtrancp-cdcl}_W[\text{of } S \ T]$
 $\text{rtrancp-cdcl}_W\text{-consistent-inv inv}$ **by** *blast*
from cdcl_W **show** *False*
proof *cases*
case *conflict'-without-decide*
have $\text{no-step propagate } S$
using *n-s* **by** *blast*
then have $\text{conflict } S \ T$
using $\text{local.conflict' trancp-cdcl}_W\text{-cp-propagate-with-conflict-or-not}[\text{of } S \ T]$
unfolding *full1-def* **by** (*metis full1-def local.conflict'-without-decide rtrancp-unfold trancp-unfold-begin*)
moreover
then obtain T' **where** $\text{full cdcl}_W\text{-bj } T \ T'$
using $\text{cdcl}_W\text{-bj-exists-normal-form inv-T}$ **by** *blast*
ultimately show *False* **using** $\text{cdcl}_W\text{-merge-cp.conflict' n-s}$ **by** *meson*
next
case (*bj'-without-decide S'*)
then show *?thesis*
using *confl* **unfolding** *full1-def* **by** (*fastforce simp: cdcl_W-bj.simps dest: trancpD*)
qed
qed

lemma *conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp*:
assumes
inv: $\text{cdcl}_W\text{-all-struct-inv } S$ and
n-s: $\text{no-step cdcl}_W\text{-s'-without-decide } S$
shows $\text{no-step cdcl}_W\text{-merge-cp } S$
proof (rule *ccontr*)
assume $\neg ?thesis$
then obtain T **where** $\text{cdcl}_W\text{-merge-cp } S \ T$
by *auto*
then show *False*
proof *cases*
case (*conflict' S'*)
then show *False* **using** $\text{n-s conflict'-without-decide conflict-is-full1-cdcl}_W\text{-cp}$ **by** *blast*
next
case *propagate'*
moreover
have $\text{cdcl}_W\text{-all-struct-inv } T$
using *inv* **by** (*meson local.propagate' rtrancp-cdcl_W-all-struct-inv-inv rtrancp-propagate-is-rtrancp-cdcl_W trancp-into-rtrancp*)
then obtain U **where** $\text{full cdcl}_W\text{-cp } T \ U$
using $\text{cdcl}_W\text{-cp-normalized-element-all-inv}$ **by** *auto*
ultimately have $\text{full1 cdcl}_W\text{-cp } S \ U$
using $\text{trancp-full-full1I}[\text{of cdcl}_W\text{-cp } S \ T \ U] \ \text{cdcl}_W\text{-cp.propagate'}$

$\text{trancpl-mono[of propagate cdcl}_W\text{-cp] by blast}$
then show $\text{False using conflict'-without-decide n-s by blast}$
qed
qed

lemma $\text{no-step-cdcl}_W\text{-merge-cp-no-step-cdcl}_W\text{-cp}:$
 $\text{no-step cdcl}_W\text{-merge-cp } S \implies \text{cdcl}_W\text{-M-level-inv } S \implies \text{no-step cdcl}_W\text{-cp } S$
using $\text{cdcl}_W\text{-bj-exists-normal-form cdcl}_W\text{-consistent-inv[OF cdcl}_W\text{.conflict, of } S]$
by $(\text{metis cdcl}_W\text{-cp.cases cdcl}_W\text{-merge-cp.simps trancpl.intros(1))$

lemma $\text{conflicting-not-true-rtrancpl-cdcl}_W\text{-merge-cp-no-step-cdcl}_W\text{-bj}:$
assumes
 $\text{conflicting } S = \text{None and}$
 $\text{cdcl}_W\text{-merge-cp}^{**} S T$
shows $\text{no-step cdcl}_W\text{-bj } T$
using $\text{assms(2,1) by (induction)}$
 $(\text{fastforce simp: cdcl}_W\text{-merge-cp.simps full-def trancpl-unfold-end cdcl}_W\text{-bj.simps})+$

lemma $\text{conflicting-true-full-cdcl}_W\text{-merge-cp-iff-full-cdcl}_W\text{-s'-without-decode}:$
assumes
 $\text{confl: conflicting } S = \text{None and}$
 $\text{inv: cdcl}_W\text{-all-struct-inv } S$
shows
 $\text{full cdcl}_W\text{-merge-cp } S V \longleftrightarrow \text{full cdcl}_W\text{-s'-without-decode } S V \text{ (is ?fw } \longleftrightarrow ?s')$

proof

assume $?fw$
then have $st: \text{cdcl}_W\text{-merge-cp}^{**} S V$ **and** $n\text{-s: no-step cdcl}_W\text{-merge-cp } V$
unfolding $\text{full-def by blast+}$
have $\text{inv-V: cdcl}_W\text{-all-struct-inv } V$
using $\text{rtrancpl-cdcl}_W\text{-merge-cp-rtrancpl-cdcl}_W[\text{of } S V] \langle ?fw \rangle$ **unfolding** full-def
by $(\text{simp add: inv rtrancpl-cdcl}_W\text{-all-struct-inv-inv})$
consider
 $(s') \text{ cdcl}_W\text{-s'-without-decode}^{**} S V$
 $| (\text{propa}) T \text{ where } \text{cdcl}_W\text{-s'-without-decode}^{**} S T \text{ and } \text{propagate}^{++} T V$
 $| (\text{bj}) T U \text{ where } \text{cdcl}_W\text{-s'-without-decode}^{**} S T \text{ and full1 cdcl}_W\text{-bj } T U \text{ and } \text{propagate}^{**} U V$
using $\text{rtrancpl-cdcl}_W\text{-merge-cp-is-rtrancpl-cdcl}_W\text{-s'-without-decode confl st n-s by metis}$
then have $\text{cdcl}_W\text{-s'-without-decode}^{**} S V$

proof cases

case s'
then show $?thesis .$

next

case propa **note** $s' = \text{this(1)}$ **and** $\text{propa} = \text{this(2)}$
have $\text{no-step cdcl}_W\text{-cp } V$
using $\text{no-step-cdcl}_W\text{-merge-cp-no-step-cdcl}_W\text{-cp n-s inv-V}$
unfolding $\text{cdcl}_W\text{-all-struct-inv-def by blast}$
then have $\text{full1 cdcl}_W\text{-cp } T V$
using $\text{propa trancpl-mono[of propagate cdcl}_W\text{-cp] cdcl}_W\text{-cp.propagate' unfolding full1-def}$
by blast
then have $\text{cdcl}_W\text{-s'-without-decode } T V$
using $\text{conflict'-without-decide by blast}$
then show $?thesis$ **using** s' **by** auto

next

case bj **note** $s' = \text{this(1)}$ **and** $\text{bj} = \text{this(2)}$ **and** $\text{propa} = \text{this(3)}$
have $\text{no-step cdcl}_W\text{-cp } V$
using $\text{no-step-cdcl}_W\text{-merge-cp-no-step-cdcl}_W\text{-cp n-s inv-V}$

```

    unfolding cdclW-all-struct-inv-def by blast
  then have full cdclW-cp U V
    using propa rtranclp-mono[of propagate cdclW-cp] cdclW-cp.propagate' unfolding full-def
    by blast
  moreover have no-step cdclW-cp T
    using bj unfolding full1-def by (fastforce dest!: 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 = None)
case False
{ fix ss :: 'st
  have ff1:  $\forall s \text{ sa. } \neg \text{cdcl}_W\text{-s'} s \text{ sa} \vee \text{full1 cdcl}_W\text{-cp s sa}$ 
     $\vee (\exists sb. \text{decide s sb} \wedge \text{no-step cdcl}_W\text{-cp s} \wedge \text{full cdcl}_W\text{-cp sb sa})$ 
     $\vee (\exists sb. \text{full1 cdcl}_W\text{-bj s sb} \wedge \text{no-step cdcl}_W\text{-cp s} \wedge \text{full cdcl}_W\text{-cp sb sa})$ 
    by (metis cdclW-s'.cases)
  have ff2:  $(\forall p \text{ s sa. } \neg \text{full1 p (s::'st) sa} \vee p^{++} s \text{ sa} \wedge \text{no-step p sa})$ 
     $\wedge (\forall p \text{ s sa. } (\neg p^{++} (s::'st) sa \vee (\exists s. p \text{ sa s})) \vee \text{full1 p s sa})$ 
    by (meson full1-def)
  obtain ssa :: ('st  $\Rightarrow$  'st  $\Rightarrow$  bool)  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
    ff3:  $\forall p \text{ s sa. } \neg p^{++} s \text{ sa} \vee p \text{ s (ssa p s sa)} \wedge p^{**} (ssa p s sa) \text{ sa}$ 
    by (metis (no-types) tranclpD)
  then have a3:  $\neg \text{cdcl}_W\text{-cp}^{++} V \text{ ss}$ 
    using False by (metis option-full-cdclW-cp full-def)
  have  $\bigwedge s. \neg \text{cdcl}_W\text{-bj}^{++} V s$ 
    using ff3 False by (metis confl st
      conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj)
  then have  $\neg \text{cdcl}_W\text{-s'-without-decide V ss}$ 
    using ff1 a3 ff2 by (metis cdclW-s'-without-decide.cases)
}
then show ?thesis
  by fastforce
next
case True
  then show ?thesis
    using conflicting-true-no-step-cdclW-merge-cp-no-step-s'-without-decide n-s inv-V
    unfolding cdclW-all-struct-inv-def by blast
qed
ultimately show ?s' unfolding full-def by blast
next
assume s': ?s'
then have st: cdclW-s'-without-decide** S V and n-s: no-step cdclW-s'-without-decide V
  unfolding full-def by auto
then have cdclW** S V
  using rtranclp-cdclW-s'-without-decide-rtranclp-cdclW st by blast
then have inv-V: cdclW-all-struct-inv V using inv rtranclp-cdclW-all-struct-inv-inv by blast
then have n-s-cp-V: no-step cdclW-cp V
  using cdclW-cp-normalized-element-all-inv[of V] full-fullI[of cdclW-cp V] n-s
  conflict'-without-decide conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp
  no-step-cdclW-merge-cp-no-step-cdclW-cp
  unfolding cdclW-all-struct-inv-def by presburger
have n-s-bj: no-step cdclW-bj V
proof (rule ccontr)

```

```

assume  $\neg ?thesis$ 
then obtain  $W$  where  $W: cdcl_W\text{-}bj\ V\ W$  by blast
have  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ W$ 
  using  $W\ cdcl_W.simps\ cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv\ inv\text{-}V$  by blast
then obtain  $W'$  where  $full1\ cdcl_W\text{-}bj\ V\ W'$ 
  using  $cdcl_W\text{-}bj\text{-}exists\text{-}normal\text{-}form[of\ W]\ full\text{-}fullI[of\ cdcl_W\text{-}bj\ V\ W]\ W$ 
  unfolding  $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}def$ 
  by blast
moreover
  then have  $cdcl_W^{++}\ V\ W'$ 
    using  $trancp\text{-}mono[of\ cdcl_W\text{-}bj\ cdcl_W]\ cdcl_W.other\ cdcl_W\text{-}o.bj$  unfolding  $full1\text{-}def$  by blast
  then have  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ W'$ 
    by (meson  $inv\text{-}V\ rtrancp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv\ trancp\text{-}into\text{-}rtrancp$ )
  then obtain  $X$  where  $full\ cdcl_W\text{-}cp\ W'\ X$ 
    using  $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv$  by blast
ultimately show False
  using  $bj'\text{-}without\text{-}decide\ n\text{-}s\text{-}cp\text{-}V\ n\text{-}s$  by blast
qed
from  $s'$  consider
  (cp-true)  $cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V$  and  $conflicting\ V = None$ 
| (cp-false)  $cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V$  and  $conflicting\ V \neq None$  and  $no\text{-}step\ cdcl_W\text{-}cp\ V$  and
   $no\text{-}step\ cdcl_W\text{-}bj\ V$ 
| (cp-conf)  $T$  where  $cdcl_W\text{-}merge\text{-}cp^{**}\ S\ T\ conflict\ T\ V$ 
using  $rtrancp\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}is\text{-}rtrancp\text{-}cdcl_W\text{-}merge\text{-}cp[of\ S\ V]\ confl$ 
unfolding  $full\text{-}def$  by meson
then have  $cdcl_W\text{-}merge\text{-}cp^{**}\ S\ V$ 
proof cases
  case cp-conf note  $S\text{-}T = this(1)$  and  $conf\text{-}V = this(2)$ 
  have  $full\ cdcl_W\text{-}bj\ V\ V$ 
    using  $conf\text{-}V\ n\text{-}s\text{-}bj$  unfolding  $full\text{-}def$  by fast
  then have  $cdcl_W\text{-}merge\text{-}cp\ T\ V$ 
    using  $cdcl_W\text{-}merge\text{-}cp.conflict'\ conf\text{-}V$  by auto
  then show  $?thesis$  using  $S\text{-}T$  by auto
qed fast+
moreover
  then have  $cdcl_W^{**}\ S\ V$  using  $rtrancp\text{-}cdcl_W\text{-}merge\text{-}cp\text{-}rtrancp\text{-}cdcl_W$  by blast
  then have  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ V$ 
    using  $inv\ rtrancp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$  by blast
  then have  $no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ V$ 
    using  $conflicting\text{-}true\text{-}no\text{-}step\text{-}s'\text{-}without\text{-}decide\text{-}no\text{-}step\text{-}cdcl_W\text{-}merge\text{-}cp\ s'$ 
    unfolding  $full\text{-}def$  by blast
ultimately show  $?fw$  unfolding  $full\text{-}def$  by auto
qed

lemma  $conflicting\text{-}true\text{-}full1\text{-}cdcl_W\text{-}merge\text{-}cp\text{-}iff\text{-}full1\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decode$ :
assumes
   $confl: conflicting\ S = None$  and
   $inv: cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ 
shows
   $full1\ cdcl_W\text{-}merge\text{-}cp\ S\ V \longleftrightarrow full1\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S\ V$ 
proof  $-$ 
  have  $full\ cdcl_W\text{-}merge\text{-}cp\ S\ V = full\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S\ V$ 
    using  $confl\ conflicting\text{-}true\text{-}full\text{-}cdcl_W\text{-}merge\text{-}cp\text{-}iff\text{-}full\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decode\ inv$ 
    by blast
  then show  $?thesis$  unfolding  $full\text{-}unfold\ full1\text{-}def$ 

```


by (metis (mono-tags) tranclp-unfold-begin)
qed

lemma *conflicting-true-full1-cdcl_W-merge-cp-imp-full1-cdcl_W-s'-without-decode:*

assumes

fw: full1 cdcl_W-merge-cp *S V* **and**

inv: cdcl_W-all-struct-inv *S*

shows

full1 cdcl_W-s'-without-decode *S V*

proof –

have *conflicting S* = None

using *fw* **unfolding** full1-def **by** (auto dest!: tranclpD simp: cdcl_W-merge-cp.simps)

then show ?thesis

using *conflicting-true-full1-cdcl_W-merge-cp-iff-full1-cdcl_W-s'-without-decode fw inv* **by** blast

qed

inductive cdcl_W-merge-stgy **where**

fw-s-cp[intro]: full1 cdcl_W-merge-cp *S T* \implies cdcl_W-merge-stgy *S T* |

fw-s-decide[intro]: decide *S T* \implies no-step cdcl_W-merge-cp *S* \implies full cdcl_W-merge-cp *T U*
 \implies cdcl_W-merge-stgy *S U*

lemma *cdcl_W-merge-stgy-tranclp-cdcl_W-merge:*

assumes *fw*: cdcl_W-merge-stgy *S T*

shows cdcl_W-merge⁺⁺ *S T*

proof –

{ **fix** *S T*

assume full1 cdcl_W-merge-cp *S T*

then have cdcl_W-merge⁺⁺ *S T*

using tranclp-mono[of cdcl_W-merge-cp cdcl_W-merge⁺⁺] cdcl_W-merge-cp-tranclp-cdcl_W-merge

unfolding full1-def

by auto

} **note** full1-cdcl_W-merge-cp-cdcl_W-merge = *this*

show ?thesis

using *fw*

apply (induction rule: cdcl_W-merge-stgy.induct)

using full1-cdcl_W-merge-cp-cdcl_W-merge **apply** simp

unfolding full-unfold **by** (auto dest!: full1-cdcl_W-merge-cp-cdcl_W-merge *fw-decide*)

qed

lemma *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-merge:*

assumes *fw*: cdcl_W-merge-stgy^{**} *S T*

shows cdcl_W-merge^{**} *S T*

using *fw* cdcl_W-merge-stgy-tranclp-cdcl_W-merge rtranclp-mono[of cdcl_W-merge-stgy cdcl_W-merge⁺⁺]

unfolding tranclp-rtranclp-rtranclp **by** blast

lemma *cdcl_W-merge-stgy-rtranclp-cdcl_W:*

cdcl_W-merge-stgy *S T* \implies cdcl_W^{**} *S T*

apply (induction rule: cdcl_W-merge-stgy.induct)

using rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W **unfolding** full1-def

apply (simp add: tranclp-into-rtranclp)

using rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W cdcl_W-o.decide cdcl_W-other **unfolding** full-def

by (meson r-into-rtranclp rtranclp-trans)

lemma *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W:*

cdcl_W-merge-stgy^{**} *S T* \implies cdcl_W^{**} *S T*

using *rtrancpl-mono*[of *cdcl_W-merge-stgy cdcl_W***] *cdcl_W-merge-stgy-rtrancpl-cdcl_W* **by** *auto*

lemma *cdcl_W-merge-stgy-cases*[*consumes 1, case-names fw-s-cp fw-s-decide*]:
assumes
cdcl_W-merge-stgy S U
full1 cdcl_W-merge-cp S U \implies P
 $\bigwedge T. \text{decide } S \ T \implies \text{no-step } cdcl_W\text{-merge-cp } S \implies \text{full } cdcl_W\text{-merge-cp } T \ U \implies P$
shows *P*
using *assms* **by** (*auto simp: cdcl_W-merge-stgy.simps*)

inductive *cdcl_W-s'-w* :: '*st \Rightarrow 'st \Rightarrow bool* **where**
conflict': *full1 cdcl_W-s'-without-decide S S' \implies cdcl_W-s'-w S S' |*
decide': *decide S S' \implies no-step cdcl_W-s'-without-decide S \implies full cdcl_W-s'-without-decide S' S''*
 $\implies \text{cdcl}_W\text{-s'-w } S \ S''$

lemma *cdcl_W-s'-w-rtrancpl-cdcl_W*:
*cdcl_W-s'-w S T \implies cdcl_W** S T*
apply (*induction rule: cdcl_W-s'-w.induct*)
using *rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W unfolding full1-def*
apply (*simp add: trancpl-into-rtrancpl*)
using *rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W unfolding full-def*
by (*meson decide other rtrancpl-into-trancpl2 trancpl-into-rtrancpl*)

lemma *rtrancpl-cdcl_W-s'-w-rtrancpl-cdcl_W*:
*cdcl_W-s'-w** S T \implies cdcl_W** S T*
using *rtrancpl-mono*[of *cdcl_W-s'-w cdcl_W***] *cdcl_W-s'-w-rtrancpl-cdcl_W* **by** *auto*

lemma *no-step-cdcl_W-cp-no-step-cdcl_W-s'-without-decide*:
assumes *no-step cdcl_W-cp S and conflicting S = None and inv: cdcl_W-M-level-inv S*
shows *no-step cdcl_W-s'-without-decide S*
by (*metis assms cdcl_W-cp.conflict' cdcl_W-cp.propagate' cdcl_W-merge-restart-cases trancplD*
conflicting-true-no-step-cdcl_W-merge-cp-no-step-s'-without-decide)

lemma *no-step-cdcl_W-cp-no-step-cdcl_W-merge-restart*:
assumes *no-step cdcl_W-cp S and conflicting S = None*
shows *no-step cdcl_W-merge-cp S*
by (*metis assms(1) cdcl_W-cp.conflict' cdcl_W-cp.propagate' cdcl_W-merge-restart-cases trancplD*)

lemma *after-cdcl_W-s'-without-decide-no-step-cdcl_W-cp*:
assumes *cdcl_W-s'-without-decide S T*
shows *no-step cdcl_W-cp T*
using *assms* **by** (*induction rule: cdcl_W-s'-without-decide.induct*) (*auto simp: full1-def full-def*)

lemma *no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-cp*:
cdcl_W-all-struct-inv S \implies no-step cdcl_W-s'-without-decide S \implies no-step cdcl_W-cp S
by (*simp add: conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp*
no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp cdcl_W-all-struct-inv-def)

lemma *after-cdcl_W-s'-w-no-step-cdcl_W-cp*:
assumes *cdcl_W-s'-w S T and cdcl_W-all-struct-inv S*
shows *no-step cdcl_W-cp T*
using *assms*
proof (*induction rule: cdcl_W-s'-w.induct*)
case *conflict'*
then show *?case*
by (*auto simp: full1-def trancpl-unfold-end after-cdcl_W-s'-without-decide-no-step-cdcl_W-cp*)

next
case (*decide'* S T U)
moreover
then have $cdcl_W^{**} S$ U
using $rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W[of\ T\ U]\ cdcl_W.other[of\ S\ T]$
 $cdcl_W-o.decide$ **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 $S = T \vee no-step\ cdcl_W-cp\ T$
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 $S = T \vee no-step\ cdcl_W-cp\ T$
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 $S \text{-} T \text{ cdcl}_W\text{-all-struct-inv-inv inv other}$ by *blast*
 then obtain T' where $\text{full1 cdcl}_W\text{-bj } S \text{ } T'$
 using $\text{cdcl}_W\text{-bj-exists-normal-form}[of T] \text{ full-fullI } S \text{-} T$ unfolding $\text{cdcl}_W\text{-all-struct-inv-def}$
 by *metis*
 moreover
 then have $\text{cdcl}_W^{**} S \text{ } T'$
 using $\text{rtranclp-mono}[of \text{cdcl}_W\text{-bj cdcl}_W] \text{ cdcl}_W.\text{other cdcl}_W\text{-o.bj tranclp-into-rtranclp}[of \text{cdcl}_W\text{-bj}]$
 unfolding full1-def by (*metis (full-types) predicate2D predicate2I*)
 then have $\text{cdcl}_W\text{-all-struct-inv } T'$
 using $\text{inv rtranclp-cdcl}_W\text{-all-struct-inv-inv}$ by *blast*
 then obtain U where $\text{full cdcl}_W\text{-cp } T' \text{ } U$
 using $\text{cdcl}_W\text{-cp-normalized-element-all-inv}$ by *blast*
 moreover have $\text{no-step cdcl}_W\text{-cp } S$
 using $S \text{-} T$ by (*auto simp: cdcl}_W\text{-bj.simps}*)
 ultimately show *False*
 using $\text{assms cdcl}_W\text{-s'-without-decide.intros}(2)[of S \text{ } T' \text{ } U]$ by *fast*
 qed

lemma $\text{cdcl}_W\text{-s'-w-no-step-cdcl}_W\text{-bj}$:
 assumes $\text{cdcl}_W\text{-s'-w } S \text{ } T$ and $\text{cdcl}_W\text{-all-struct-inv } S$
 shows $\text{no-step cdcl}_W\text{-bj } T$
 using *assms* apply *induction*
 using $\text{rtranclp-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W \text{ rtranclp-cdcl}_W\text{-all-struct-inv-inv}$
 $\text{no-step-cdcl}_W\text{-s'-without-decide-no-step-cdcl}_W\text{-bj}$ unfolding full1-def
 apply (*meson tranclp-into-rtranclp*)
 using $\text{rtranclp-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W \text{ rtranclp-cdcl}_W\text{-all-struct-inv-inv}$
 $\text{no-step-cdcl}_W\text{-s'-without-decide-no-step-cdcl}_W\text{-bj}$ unfolding full-def
 by (*meson cdcl}_W\text{-merge-restart-cdcl}_W \text{ fw-r-decide}*)

lemma $\text{rtranclp-cdcl}_W\text{-s'-w-no-step-cdcl}_W\text{-bj-or-eq}$:
 assumes $\text{cdcl}_W\text{-s'-w}^{**} S \text{ } T$ and $\text{cdcl}_W\text{-all-struct-inv } S$
 shows $S = T \vee \text{no-step cdcl}_W\text{-bj } T$
 using *assms* apply *induction*
 apply *simp*
 using $\text{rtranclp-cdcl}_W\text{-s'-w-rtranclp-cdcl}_W \text{ rtranclp-cdcl}_W\text{-all-struct-inv-inv}$
 $\text{cdcl}_W\text{-s'-w-no-step-cdcl}_W\text{-bj}$ by *meson*

lemma $\text{rtranclp-cdcl}_W\text{-s'-no-step-cdcl}_W\text{-s'-without-decide-decomp-into-cdcl}_W\text{-merge}$:
 assumes
 $\text{cdcl}_W\text{-s'}^{**} R \text{ } V$ and
 $\text{conflicting } R = \text{None}$ and
 $\text{inv: cdcl}_W\text{-all-struct-inv } R$
 shows $(\text{cdcl}_W\text{-merge-stgy}^{**} R \text{ } V \wedge \text{conflicting } V = \text{None})$
 $\vee (\text{cdcl}_W\text{-merge-stgy}^{**} R \text{ } V \wedge \text{conflicting } V \neq \text{None} \wedge \text{no-step cdcl}_W\text{-bj } V)$
 $\vee (\exists S \text{ } T \text{ } U. \text{cdcl}_W\text{-merge-stgy}^{**} R \text{ } S \wedge \text{no-step cdcl}_W\text{-merge-cp } S \wedge \text{decide } S \text{ } T$
 $\wedge \text{cdcl}_W\text{-merge-cp}^{**} T \text{ } U \wedge \text{conflict } U \text{ } V)$
 $\vee (\exists S \text{ } T. \text{cdcl}_W\text{-merge-stgy}^{**} R \text{ } S \wedge \text{no-step cdcl}_W\text{-merge-cp } S \wedge \text{decide } S \text{ } T$
 $\wedge \text{cdcl}_W\text{-merge-cp}^{**} T \text{ } V$
 $\wedge \text{conflicting } V = \text{None})$
 $\vee (\text{cdcl}_W\text{-merge-cp}^{**} R \text{ } V \wedge \text{conflicting } V = \text{None})$
 $\vee (\exists U. \text{cdcl}_W\text{-merge-cp}^{**} R \text{ } U \wedge \text{conflict } U \text{ } V)$
 using *assms*(1,2)
proof *induction*
 case *base*
 then show ?case by *simp*

```

next
case (step V W) note st = this(1) and s' = this(2) and IH = this(3)[OF this(4)] and
  n-s-R = this(4)
from s'
show ?case
proof cases
  case conflict'
  consider
    (s') cdclW-merge-stgy** R V
  | (dec-conf) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T U and conflict U V
  | (dec) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and decide S T
    and cdclW-merge-cp** T V and conflicting V = None
  | (cp) cdclW-merge-cp** R V
  | (cp-conf) U where cdclW-merge-cp** R U and conflict U V
  using IH by meson
then show ?thesis
proof cases
next
  case s'
  then have R = V
  by (metis full1-def inv local.conflict' tranclp-unfold-begin
    rtranclp-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq)
  consider
    (V-W) V = W
  | (propa) propagate++ V W and conflicting W = None
  | (propa-conf) V' where propagate** V V' and conflict V' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V W] conflict'
  unfolding full-unfold full1-def by meson
then show ?thesis
proof cases
  case V-W
  then show ?thesis using ⟨R = V⟩ n-s-R by simp
next
  case propa
  then show ?thesis using ⟨R = V⟩ by auto
next
  case propa-conf
  moreover
    then have cdclW-merge-cp** V V'
    by (metis rtranclp-unfold cdclW-merge-cp.propagate' r-into-rtranclp)
  ultimately show ?thesis using s' ⟨R = V⟩ by blast
qed
next
  case dec-conf note - = this(5)
  then have False using conflict' unfolding full1-def by (auto dest!: tranclpD)
  then show ?thesis by fast
next
  case dec note T-V = this(4)
  consider
    (propa) propagate++ V W and conflicting W = None
  | (propa-conf) V' where propagate** V V' and conflict V' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V W] conflict'
  unfolding full1-def by meson
then show ?thesis

```

```

proof cases
  case propa
  then show ?thesis
    by (meson T-V cdclW-merge-cp.propagate' dec rtrancpl.rtrancpl-into-rtrancpl)
  next
  case propa-confli
  then have cdclW-merge-cp** T V'
    using T-V by (metis rtrancpl-unfold cdclW-merge-cp.propagate' rtrancpl.simps)
  then show ?thesis using dec propa-confli(2) by metis
  qed
next
  case cp
  consider
    (propa) propagate++ V W and conflicting W = None
  | (propa-confli) V' where propagate** V V' and conflict V' W
  using trancpl-cdclW-cp-propagate-with-conflict-or-not[of V W] conflict'
  unfolding full1-def by meson
  then show ?thesis
  proof cases
    case propa
    then show ?thesis by (meson cdclW-merge-cp.propagate' cp rtrancpl.rtrancpl-into-rtrancpl)
  next
    case propa-confli
    then show ?thesis
      using propa-confli(2) by (metis rtrancpl-unfold cdclW-merge-cp.propagate'
        cp rtrancpl.rtrancpl-into-rtrancpl)
    qed
  next
    case cp-confli
    then show ?thesis using conflict' unfolding full1-def by (fastforce dest!: trancplD)
  qed
next
  case (decide' V')
  then have conf-V: conflicting V = None
  by auto
  consider
    (s') cdclW-merge-stgy** R V
  | (dec-confli) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
    decide S T and cdclW-merge-cp** T U and conflict U V
  | (dec) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and decide S T
    and cdclW-merge-cp** T V and conflicting V = None
  | (cp) cdclW-merge-cp** R V
  | (cp-confli) 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' = None using decide'(1) by auto
    have full: full1 cdclW-cp V' W  $\vee$  (V' = W  $\wedge$  no-step cdclW-cp W)
      using decide'(3) unfolding full-unfold by blast
    consider
      (V'-W) V' = W
    | (propa) propagate++ V' W and conflicting W = None
    | (propa-confli) V'' where propagate** V' V'' and conflict V'' W
    using trancpl-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
  tranclp-cdclW-cp-propagate-with-conflict-or-not)
then show ?thesis
proof cases
case V'-W
then show ?thesis
  using confl-V' local.decide'(1,2) s' conf-V
  no-step-cdclW-cp-no-step-cdclW-merge-restart[of V] by blast
next
case propa
then show ?thesis using local.decide'(1,2) s' by (metis cdclW-merge-cp.simps conf-V
  no-step-cdclW-cp-no-step-cdclW-merge-restart r-into-rtranclp)
next
case propa-confl
then have cdclW-merge-cp** V' V''
  by (metis rtranclp-unfold cdclW-merge-cp.propagate' r-into-rtranclp)
then show ?thesis
  using local.decide'(1,2) propa-confl(2) s' conf-V
  no-step-cdclW-cp-no-step-cdclW-merge-restart
  by metis
qed
next
case (dec) note s' = this(1) and dec = this(2) and cp = this(3) and ns-cp-T = this(4)
have full cdclW-merge-cp T V
  unfolding full-def by (simp add: conf-V local.decide'(2)
    no-step-cdclW-cp-no-step-cdclW-merge-restart ns-cp-T)
moreover have no-step cdclW-merge-cp V
  by (simp add: conf-V local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart)
moreover have no-step cdclW-merge-cp S
  by (metis dec)
ultimately have cdclW-merge-stgy S V
  using cp by blast
then have cdclW-merge-stgy** R V using s' by auto
consider
  (V'-W) V' = W
  | (propa) propagate++ V' W and conflicting W = None
  | (propa-confl) V'' where propagate** V' V'' and conflict V'' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
  unfolding full-unfold full1-def by meson
then show ?thesis
proof cases
case V'-W
moreover have conflicting V' = None
  using decide'(1) by auto
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  $cdcl_W\text{-merge-cp}^{**} V' V''$ 
      by (metis  $cdcl_W\text{-merge-cp.propagate}' rtranclp\text{-unfold tranclp-unfold-end}$ )
    ultimately show ?thesis using  $\langle cdcl_W\text{-merge-stgy}^{**} R V \rangle decide'$ 
       $\langle no\text{-step } cdcl_W\text{-merge-cp } V \rangle$  by (meson  $r\text{-into-rtranclp}$ )
  qed
next
case cp
have  $no\text{-step } cdcl_W\text{-merge-cp } V$ 
  using  $conf\text{-}V local.decide'(2) no\text{-step-cdcl}_W\text{-cp-no-step-cdcl}_W\text{-merge-restart}$  by blast
then have  $full\ cdcl_W\text{-merge-cp } R V$ 
  unfolding  $full\text{-def}$  using cp by fast
then have  $cdcl_W\text{-merge-stgy}^{**} R V$ 
  unfolding  $full\text{-unfold}$  by auto
have  $full1\ cdcl_W\text{-cp } V' W \vee (V' = W \wedge no\text{-step } cdcl_W\text{-cp } W)$ 
  using  $decide'(3) unfolding full\text{-unfold}$  by blast

consider
  ( $V' - W$ )  $V' = W$ 
| (propa)  $propagate^{++} V' W$  and  $conflicting W = None$ 
| (propa-conf)  $V''$  where  $propagate^{**} V' V''$  and  $conflict V'' W$ 
  using  $tranclp\text{-cdcl}_W\text{-cp-propagate-with-conflict-or-not}[of V' W] decide'$ 
  unfolding  $full\text{-unfold full1-def}$  by meson
then show ?thesis

proof cases
case  $V' - W$ 
moreover have  $conflicting V' = None$ 
  using  $decide'(1)$  by auto
ultimately show ?thesis
  using  $\langle cdcl_W\text{-merge-stgy}^{**} R V \rangle decide' \langle no\text{-step } cdcl_W\text{-merge-cp } V \rangle$  by blast
next
case propa
moreover then have  $cdcl_W\text{-merge-cp } V' W$ 
  by auto
ultimately show ?thesis using  $\langle cdcl_W\text{-merge-stgy}^{**} R V \rangle decide'$ 
   $\langle no\text{-step } cdcl_W\text{-merge-cp } V \rangle$  by (meson  $r\text{-into-rtranclp}$ )
next
case propa-conf
moreover then have  $cdcl_W\text{-merge-cp}^{**} V' V''$ 
  by (metis  $cdcl_W\text{-merge-cp.propagate}' rtranclp\text{-unfold tranclp-unfold-end}$ )
ultimately show ?thesis using  $\langle cdcl_W\text{-merge-stgy}^{**} R V \rangle decide'$ 
   $\langle no\text{-step } cdcl_W\text{-merge-cp } V \rangle$  by (meson  $r\text{-into-rtranclp}$ )
qed
next
case (dec-conf)
show ?thesis using  $conf\text{-}V dec-conf(5)$  by auto
next
case cp-conf
then show ?thesis using  $decide'$  apply - by (intro  $HOL.disjI2$ ) fastforce
qed
next
case (bj'  $V'$ )
then have  $\neg no\text{-step } cdcl_W\text{-bj } V$ 
  by (auto dest:  $tranclpD simp: full1-def$ )
then consider

```



```

(s') cdclW-merge-stgy** R V and conflicting V = None
| (dec-confl) S T U where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and
  decide S T and cdclW-merge-cp** T U and conflict U V
| (dec) S T where cdclW-merge-stgy** R S and no-step cdclW-merge-cp S and decide S T
  and cdclW-merge-cp** T V and conflicting V = None
| (cp) cdclW-merge-cp** R V and conflicting V = None
| (cp-confl) U where cdclW-merge-cp** R U and conflict U V
using IH by meson
then show ?thesis
proof cases
  case s' note - = this(2)
  then have False
    using bj'(1) unfolding full1-def by (force dest!: tranclpD simp: cdclW-bj.simps)
  then show ?thesis by fast
next
  case dec note - = this(5)
  then have False
    using bj'(1) unfolding full1-def by (force dest!: tranclpD simp: cdclW-bj.simps)
  then show ?thesis by fast
next
  case dec-confl
  then have cdclW-merge-cp U V'
    using bj' cdclW-merge-cp.intros(1)[of U V V'] by (simp add: full-unfold)
  then have cdclW-merge-cp** T V'
    using dec-confl(4) by simp
  consider
    (V'-W) V' = W
  | (propa) propagate++ V' W and conflicting W = None
  | (propa-confl) V'' where propagate** V' V'' and conflict V'' W
  using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] bj'(3)
  unfolding full-unfold full1-def by meson
  then show ?thesis
  proof cases
    case V'-W
    then have no-step cdclW-cp V'
      using bj'(3) unfolding full-def by auto
    then have no-step cdclW-merge-cp V'
      by (metis cdclW-cp.propagate' cdclW-merge-cp.cases tranclpD
        no-step-cdclW-cp-no-conflict-no-propagate(1) )
    then have full1 cdclW-merge-cp T V'
      unfolding full1-def using ⟨cdclW-merge-cp U V'⟩ dec-confl(4) by auto
    then have full cdclW-merge-cp T V'
      by (simp add: full-unfold)
    then have cdclW-merge-stgy S V'
      using dec-confl(3) cdclW-merge-stgy.fw-s-decide ⟨no-step cdclW-merge-cp S⟩ by blast
    then have cdclW-merge-stgy** R V'
      using ⟨cdclW-merge-stgy** R S⟩ by auto
  show ?thesis
  proof cases
    assume conflicting W = None
    then show ?thesis using ⟨cdclW-merge-stgy** R V'⟩ ⟨V' = W⟩ by auto
  next
    assume conflicting W ≠ None
    then show ?thesis
      using ⟨cdclW-merge-stgy** R V'⟩ ⟨V' = W⟩ by (metis ⟨cdclW-merge-cp U V'⟩

```

```

      conflicting-not-true-rtrancp-cdclW-merge-cp-no-step-cdclW-bj dec-confl(5)
      r-into-rtrancp 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)
      rtrancp.rtrancp-into-rtrancp)
  next
    case propa-confl
    moreover then have cdclW-merge-cp** V' V''
      by (metis cdclW-merge-cp.propagate' rtrancp-unfold trancp-unfold-end)
    ultimately show ?thesis by (meson ⟨cdclW-merge-cp** T V'⟩ dec-confl(1-3) rtrancp-trans)
  qed
next
  case cp note - = this(2)
  then show ?thesis using bj'(1) ⟨¬ no-step cdclW-bj V'⟩
    conflicting-not-true-rtrancp-cdclW-merge-cp-no-step-cdclW-bj by auto
next
  case cp-confl
  then have cdclW-merge-cp U V' by (simp add: cdclW-merge-cp.conflict' full-unfold
    local.bj'(1))
  consider
    (V'-W) V' = W
  | (propa) propagate++ V' W and conflicting W = None
  | (propa-confl) V'' where propagate** V' V'' and conflict V'' W
  using trancp-cdclW-cp-propagate-with-conflict-or-not[of V' W] bj'
  unfolding full-unfold full1-def by meson
  then show ?thesis

proof cases
  case V'-W
  show ?thesis
  proof cases
    assume conflicting V' = None
    then show ?thesis
      using V'-W ⟨cdclW-merge-cp U V'⟩ cp-confl(1) by force
  next
    assume confl: conflicting V' ≠ None
    then have no-step cdclW-merge-stgy V'
      by (fastforce simp: cdclW-merge-stgy.simps full1-def full-def
        cdclW-merge-cp.simps dest!: trancpD)
    have no-step cdclW-merge-cp V'
      using confl by (auto simp: full1-def full-def cdclW-merge-cp.simps
        dest!: trancpD)
    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!: trancpD)

```

```

    ultimately have  $cdcl_W\text{-merge-stgy}^{**} R V'$  by auto
  show ?thesis by (metis  $V'-W \langle cdcl_W\text{-merge-cp } U V' \rangle \langle cdcl_W\text{-merge-stgy}^{**} R V' \rangle$ 
    conflicting-not-true-rtrancpl-cdclW-merge-cp-no-step-cdclW-bj cp-conf(1)
    rtrancpl.rtrancpl-into-rtrancpl step.premis)
qed
next
case propa
moreover then have  $cdcl_W\text{-merge-cp } V' W$ 
  by auto
ultimately show ?thesis using  $\langle cdcl_W\text{-merge-cp } U V' \rangle$  cp-conf(1) by force
next
case propa-conf
moreover then have  $cdcl_W\text{-merge-cp}^{**} V' V''$ 
  by (metis  $cdcl_W\text{-merge-cp.propagate}' rtrancpl\text{-unfold } trancpl\text{-unfold-end}$ )
ultimately show ?thesis
  using  $\langle cdcl_W\text{-merge-cp } U V' \rangle$  cp-conf(1) by (metis rtrancpl.rtrancpl-into-rtrancpl
    rtrancpl-trans)
qed
qed
qed
qed

```

lemma *decide-rtrancpl-cdcl_W-s'-rtrancpl-cdcl_W-s'*:

```

assumes
  dec: decide  $S T$  and
   $cdcl_W\text{-s}^{**} T U$  and
   $n\text{-s-}S$ : no-step  $cdcl_W\text{-cp } S$  and
  no-step  $cdcl_W\text{-cp } U$ 
shows  $cdcl_W\text{-s}^{**} S U$ 
using assms(2,4)
proof induction
case (step  $U V$ ) note  $st = this(1)$  and  $s' = this(2)$  and  $IH = this(3)$  and  $n\text{-s} = this(4)$ 
consider
  ( $TU$ )  $T = U$ 
  | ( $s'\text{-st}$ )  $T'$  where  $cdcl_W\text{-s}' T T'$  and  $cdcl_W\text{-s}^{**} T' U$ 
  using  $st[\text{unfolded } rtrancpl\text{-unfold}]$  by (auto dest!:  $trancplD$ )
then show ?case
proof cases
case TU
then show ?thesis
proof -
  assume  $a1: T = U$ 
  then have  $f2: cdcl_W\text{-s}' T V$ 
    using  $s'$  by force
  obtain  $ss :: 'st$  where
     $cdcl_W\text{-s}^{**} S T \vee cdcl_W\text{-cp } T ss$ 
    using  $a1$  step.IH by blast
  then show ?thesis
    using  $f2$  by (metis (full-types)  $cdcl_W\text{-s}'.decide'$   $cdcl_W\text{-s}'E$  dec full1-is-full  $n\text{-s-}S$ 
      rtrancpl-unfold trancpl-unfold-end)
qed
next
case ( $s'\text{-st } T'$ ) note  $s'\text{-}T' = this(1)$  and  $st = this(2)$ 
have  $cdcl_W\text{-s}^{**} S T'$ 
  using  $s'\text{-}T'$ 

```

```

proof cases
  case conflict'
  then have  $cdcl_W-s' S T'$ 
    using  $dec\ cdcl_W-s'.decide' n-s-S$  by ( $simp\ add: full-unfold$ )
  then show  $?thesis$ 
    using  $st$  by  $auto$ 
next
  case ( $decide' T''$ )
  then have  $cdcl_W-s' S T$ 
    using  $dec\ cdcl_W-s'.decide' n-s-S$  by ( $simp\ add: full-unfold$ )
  then show  $?thesis$  using  $decide' s'-T'$  by  $auto$ 
next
  case  $bj'$ 
  then have  $False$ 
    using  $dec\ unfolding\ full1-def$  by ( $fastforce\ dest!: tranclpD\ simp: cdcl_W-bj.simps$ )
  then show  $?thesis$  by  $fast$ 
qed
then show  $?thesis$  using  $s' st$  by  $auto$ 
qed
next
  case  $base$ 
  then have  $full\ cdcl_W-cp\ T\ T$ 
    by ( $simp\ add: full-unfold$ )
  then show  $?case$ 
    using  $cdcl_W-s'.simps\ dec\ n-s-S$  by  $auto$ 
qed

lemma  $rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-s'$ :
  assumes
     $cdcl_W-merge-stgy^{**} R\ V$  and
     $inv: cdcl_W-all-struct-inv\ R$ 
  shows  $cdcl_W-s'^{**} R\ V$ 
  using  $assms(1)$ 
proof induction
  case  $base$ 
  then show  $?case$  by  $simp$ 
next
  case ( $step\ S\ T$ ) note  $st = this(1)$  and  $fw = this(2)$  and  $IH = this(3)$ 
  have  $cdcl_W-all-struct-inv\ S$ 
    using  $inv\ rtranclp-cdcl_W-all-struct-inv-inv\ rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W\ st$  by  $blast$ 
from  $fw$  show  $?case$ 
  proof ( $cases\ rule: cdcl_W-merge-stgy-cases$ )
  case  $fw-s-cp$ 
  then show  $?thesis$ 
  proof –
    assume  $a1: full1\ cdcl_W-merge-cp\ S\ T$ 
    obtain  $ss :: ('st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st \Rightarrow 'st$  where
       $f2: \bigwedge p\ s\ sa\ pa\ sb\ sc\ sd\ pb\ se\ sf. (\neg full1\ p\ (s::'st)\ sa \vee p^{++}\ s\ sa)$ 
       $\wedge (\neg pa\ (sb::'st)\ sc \vee \neg full1\ pa\ sd\ sb) \wedge (\neg pb^{++}\ se\ sf \vee pb\ sf\ (ss\ pb\ sf))$ 
       $\vee full1\ pb\ se\ sf)$ 
    by ( $metis\ (no-types)\ full1-def$ )
    then have  $f3: cdcl_W-merge-cp^{++}\ S\ T$ 
    using  $a1$  by  $auto$ 
    obtain  $ssa :: ('st \Rightarrow 'st \Rightarrow bool) \Rightarrow 'st \Rightarrow 'st \Rightarrow 'st$  where
       $f4: \bigwedge p\ s\ sa. \neg p^{++}\ s\ sa \vee p\ s\ (ssa\ p\ s\ sa)$ 

```

```

    by (meson tranclp-unfold-begin)
  then have f5:  $\bigwedge s. \neg \text{full1 } \text{cdcl}_W\text{-merge-cp } s \ S$ 
    using f3 f2 by (metis (full-types))
  have  $\bigwedge s. \neg \text{full } \text{cdcl}_W\text{-merge-cp } s \ S$ 
    using f4 f3 by (meson full-def)
  then have  $S = R$ 
    using f5 by (metis (no-types)  $\text{cdcl}_W\text{-merge-stgy.simps } \text{rtranclp-unfold } st$ 
       $\text{tranclp-unfold-end}$ )
  then show ?thesis
    using f2 a1 by (metis (no-types)  $\langle \text{cdcl}_W\text{-all-struct-inv } S \rangle$ 
       $\text{conflicting-true-full1-cdcl}_W\text{-merge-cp-imp-full1-cdcl}_W\text{-s'-without-decode}$ 
       $\text{rtranclp-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W\text{-s' } \text{rtranclp-unfold}$ )
qed
next
case (fw-s-decide  $S'$ ) note dec = this(1) and n-S = this(2) and full = this(3)
moreover then have conflicting  $S' = \text{None}$ 
  by auto
ultimately have full  $\text{cdcl}_W\text{-s'-without-decide } S' \ T$ 
  by (meson  $\langle \text{cdcl}_W\text{-all-struct-inv } S \rangle$   $\text{cdcl}_W\text{-merge-restart-cdcl}_W$  fw-r-decide
     $\text{rtranclp-cdcl}_W\text{-all-struct-inv-inv}$ 
     $\text{conflicting-true-full-cdcl}_W\text{-merge-cp-iff-full-cdcl}_W\text{-s'-without-decode}$ )
then have a1:  $\text{cdcl}_W\text{-s}^{f**} S' \ T$ 
  unfolding full-def by (metis (full-types)  $\text{rtranclp-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W\text{-s'}$ )
have  $\text{cdcl}_W\text{-merge-stgy}^{f**} S \ T$ 
  using fw by blast
then have  $\text{cdcl}_W\text{-s}^{f**} S \ T$ 
  using decide-rtranclp-cdcl $_W\text{-s'-rtranclp-cdcl}_W\text{-s'}$  a1 by (metis  $\langle \text{cdcl}_W\text{-all-struct-inv } S \rangle$  dec
    n-S no-step-cdcl $_W\text{-merge-cp-no-step-cdcl}_W\text{-cp}$   $\text{cdcl}_W\text{-all-struct-inv-def}$ 
     $\text{rtranclp-cdcl}_W\text{-merge-stgy'-no-step-cdcl}_W\text{-cp-or-eq}$ )
then show ?thesis using IH by auto
qed
qed

```

lemma $\text{rtranclp-cdcl}_W\text{-merge-stgy-distinct-mset-clauses}$:

```

  assumes invR:  $\text{cdcl}_W\text{-all-struct-inv } R$  and
    st:  $\text{cdcl}_W\text{-merge-stgy}^{f**} R \ S$  and
    dist:  $\text{distinct-mset } (\text{clauses } R)$  and
    R:  $\text{trail } R = []$ 
  shows  $\text{distinct-mset } (\text{clauses } S)$ 
  using  $\text{rtranclp-cdcl}_W\text{-stgy-distinct-mset-clauses}[OF \text{ invR } - \text{ dist } R]$ 
    invR st  $\text{rtranclp-mono}[of \text{ cdcl}_W\text{-s' } \text{cdcl}_W\text{-stgy}^{f**}]$   $\text{cdcl}_W\text{-s'-is-rtranclp-cdcl}_W\text{-stgy}$ 
  by (auto dest!:  $\text{cdcl}_W\text{-s'-is-rtranclp-cdcl}_W\text{-stgy } \text{rtranclp-cdcl}_W\text{-merge-stgy-rtranclp-cdcl}_W\text{-s'}$ )

```

lemma $\text{no-step-cdcl}_W\text{-s'-no-step-cdcl}_W\text{-merge-stgy}$:

```

  assumes
    inv:  $\text{cdcl}_W\text{-all-struct-inv } R$  and  $s'$ :  $\text{no-step } \text{cdcl}_W\text{-s' } R$ 
  shows  $\text{no-step } \text{cdcl}_W\text{-merge-stgy } R$ 

```

proof —

```

{ fix ss :: 'st
  obtain ssa :: 'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
    ff1:  $\bigwedge s \ sa. \neg \text{cdcl}_W\text{-merge-stgy } s \ sa \vee \text{full1 } \text{cdcl}_W\text{-merge-cp } s \ sa \vee \text{decide } s \ (ssa \ s \ sa)$ 
    using  $\text{cdcl}_W\text{-merge-stgy.cases}$  by moura
  obtain ssb :: ('st  $\Rightarrow$  'st  $\Rightarrow$  bool)  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
    ff2:  $\bigwedge p \ s \ sa. \neg p^{++} \ s \ sa \vee p \ s \ (ssb \ p \ s \ sa)$ 
    by (meson tranclp-unfold-begin)

```

```

obtain ssc :: 'st ⇒ 'st where
  ff3:  $\bigwedge s \text{ sa } sb. (\neg \text{cdcl}_W\text{-all-struct-inv } s \vee \neg \text{cdcl}_W\text{-cp } s \text{ sa} \vee \text{cdcl}_W\text{-s' } s \text{ (ssc } s))$ 
     $\wedge (\neg \text{cdcl}_W\text{-all-struct-inv } s \vee \neg \text{cdcl}_W\text{-o } s \text{ sb} \vee \text{cdcl}_W\text{-s' } s \text{ (ssc } s))$ 
  using n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o by moura
then have ff4:  $\bigwedge s. \neg \text{cdcl}_W\text{-o } R \text{ } s$ 
  using s' inv by blast
have ff5:  $\bigwedge s. \neg \text{cdcl}_W\text{-cp}^{++} R \text{ } s$ 
  using ff3 ff2 s' by (metis inv)
have  $\bigwedge s. \neg \text{cdcl}_W\text{-bj}^{++} R \text{ } s$ 
  using ff4 ff2 by (metis bj)
then have  $\bigwedge s. \neg \text{cdcl}_W\text{-s'-without-decide } R \text{ } s$ 
  using ff5 by (simp add: cdclW-s'-without-decide.simps full1-def)
then have  $\neg \text{cdcl}_W\text{-s'-without-decide}^{++} R \text{ } ss$ 
  using ff2 by blast
then have  $\neg \text{cdcl}_W\text{-merge-stgy } R \text{ } ss$ 
  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).  $\text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \text{ } T$ }
  using wf-tranclp-cdclW-merge by (rule wf-subset) (auto simp: cdclW-merge-cp-tranclp-cdclW-merge)

lemma wf-cdclW-merge-stgy:
  wf{(T, S).  $\text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-stgy } S \text{ } T$ }
  using wf-tranclp-cdclW-merge by (rule wf-subset)
  (auto simp add: cdclW-merge-stgy-tranclp-cdclW-merge)

lemma cdclW-merge-cp-obtain-normal-form:
  assumes inv:  $\text{cdcl}_W\text{-all-struct-inv } R$ 
  obtains S where full cdclW-merge-cp R S
proof –
  obtain S where full ( $\lambda S \text{ } T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \text{ } T$ ) R S
  using wf-exists-normal-form-full[OF wf-cdclW-merge-cp] by blast
then have
  st:  $(\lambda S \text{ } T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \text{ } T)^{**} R \text{ } S$  and
  n-s: no-step ( $\lambda S \text{ } T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge-cp } S \text{ } T$ ) S
  unfolding full-def by blast+
have  $\text{cdcl}_W\text{-merge-cp}^{**} R \text{ } S$ 
  using st by induction auto
moreover
  have  $\text{cdcl}_W\text{-all-struct-inv } S$ 
  using st inv
  apply (induction rule: rtranclp-induct)
  apply simp
  by (meson r-into-rtranclp rtranclp-cdclW-all-struct-inv-inv
    rtranclp-cdclW-merge-cp-rtranclp-cdclW)
then have no-step cdclW-merge-cp S
  using n-s by auto
ultimately show ?thesis
  using that unfolding full-def by blast
qed

```

lemma *no-step-cdcl_W-merge-stgy-no-step-cdcl_W-s'*:
assumes
inv: *cdcl_W-all-struct-inv* *R* **and**
confl: *conflicting* *R* = *None* **and**
n-s: *no-step cdcl_W-merge-stgy* *R*
shows *no-step cdcl_W-s'* *R*
proof (*rule ccontr*)
assume \neg *?thesis*
then obtain *S* **where** *cdcl_W-s'* *R S* **by** *auto*
then show *False*
proof *cases*
case *conflict'*
then obtain *S'* **where** *full1 cdcl_W-merge-cp* *R S'*
by (*metis* (*full-types*) *cdcl_W-merge-cp-obtain-normal-form cdcl_W-s'-without-decide.simps confl*
conflicting-true-no-step-cdcl_W-merge-cp-no-step-s'-without-decide full-def full-unfold inv
cdcl_W-all-struct-inv-def)
then show *False* **using** *n-s* **by** *blast*
next
case (*decide' R'*)
then have *cdcl_W-all-struct-inv* *R'*
using *inv cdcl_W-all-struct-inv-inv cdcl_W.other cdcl_W-o.decide* **by** *meson*
then obtain *R''* **where** *full cdcl_W-merge-cp* *R' R''*
using *cdcl_W-merge-cp-obtain-normal-form* **by** *blast*
moreover have *no-step cdcl_W-merge-cp* *R*
by (*simp add: confl local.decide'(2) no-step-cdcl_W-cp-no-step-cdcl_W-merge-restart*)
ultimately show *False* **using** *n-s cdcl_W-merge-stgy.intros local.decide'(1)* **by** *blast*
next
case (*bj' R'*)
then show *False*
using *confl no-step-cdcl_W-cp-no-step-cdcl_W-s'-without-decide inv*
unfolding *cdcl_W-all-struct-inv-def* **by** *blast*
qed
qed

lemma *rtranclp-cdcl_W-merge-cp-no-step-cdcl_W-bj*:
assumes *conflicting* *R* = *None* **and** *cdcl_W-merge-cp*** *R S*
shows *no-step cdcl_W-bj* *S*
using *assms conflicting-not-true-rtranclp-cdcl_W-merge-cp-no-step-cdcl_W-bj* **by** *blast*

lemma *rtranclp-cdcl_W-merge-stgy-no-step-cdcl_W-bj*:
assumes *confl: conflicting* *R* = *None* **and** *cdcl_W-merge-stgy*** *R S*
shows *no-step cdcl_W-bj* *S*
using *assms(2)*
proof *induction*
case *base*
then show *?case*
using *confl* **by** (*auto simp: cdcl_W-bj.simps*)[]
next
case (*step S T*) **note** *st = this(1)* **and** *fw = this(2)* **and** *IH = this(3)*
have *confl-S: conflicting* *S* = *None*
using *fw apply cases*
by (*auto simp: full1-def cdcl_W-merge-cp.simps dest!: tranclpD*)
from *fw* **show** *?case*
proof *cases*
case *fw-s-cp*

```

    then show ?thesis
      using rtrancpl-cdclW-merge-cp-no-step-cdclW-bj confl-S
      by (simp add: full1-def trancpl-into-rtrancpl)
  next
    case (fw-s-decide S')
    moreover then have conflicting S' = None by auto
    ultimately show ?thesis
      using conflicting-not-true-rtrancpl-cdclW-merge-cp-no-step-cdclW-bj
      unfolding full-def by meson
  qed
qed

lemma full-cdclW-s'-full-cdclW-merge-restart:
  assumes
    conflicting R = None and
    inv: cdclW-all-struct-inv R
  shows full cdclW-s' R V  $\longleftrightarrow$  full cdclW-merge-stgy R V (is ?s'  $\longleftrightarrow$  ?fw)
proof
  assume ?s'
  then have cdclW-s'^** R V unfolding full-def by blast
  have cdclW-all-struct-inv V
    using ⟨cdclW-s'^** R V⟩ inv rtrancpl-cdclW-all-struct-inv-inv rtrancpl-cdclW-s'-rtrancpl-cdclW
    by blast
  then have n-s: no-step cdclW-merge-stgy V
    using no-step-cdclW-s'-no-step-cdclW-merge-stgy by (meson ⟨full cdclW-s' R V⟩ full-def)
  have n-s-bj: no-step cdclW-bj V
    by (metis ⟨cdclW-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ bj full-def
    n-step-cdclW-stgy-iff-no-step-cdclW-cl-cdclW-o)
  have n-s-cp: no-step cdclW-merge-cp V
  proof -
    { fix ss :: 'st
      obtain ssa :: 'st  $\Rightarrow$  'st where
        ff1:  $\forall s. \neg$  cdclW-all-struct-inv s  $\vee$  cdclW-s'-without-decide s (ssa s)
           $\vee$  no-step cdclW-merge-cp s
        using conflicting-true-no-step-s'-without-decide-no-step-cdclW-merge-cp by moura
      have ( $\forall p$  s sa.  $\neg$  full p (s::'st) sa  $\vee$  p** s sa  $\wedge$  no-step p sa) and
        ( $\forall p$  s sa. ( $\neg$  p** (s::'st) sa  $\vee$  ( $\exists s. p$  sa s))  $\vee$  full p s sa)
        by (meson full-def)+
      then have  $\neg$  cdclW-merge-cp V ss
        using ff1 by (metis (no-types) ⟨cdclW-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ cdclW-s'.sims
          cdclW-s'-without-decide.cases) }
    then show ?thesis
      by blast
  }
  qed
consider
  (fw-no-confl) cdclW-merge-stgy** R V and conflicting V = None
| (fw-confl) cdclW-merge-stgy** R V and conflicting V  $\neq$  None 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 = None
| (cp-no-confl) cdclW-merge-cp** R V and conflicting V = None
| (cp-confl) U where cdclW-merge-cp** R U and conflict U V
using rtrancpl-cdclW-s'-no-step-cdclW-s'-without-decide-decomp-into-cdclW-merge[OF
  ⟨cdclW-s'^** R V⟩ assms] by auto

```



```

then show ?fw
proof cases
  case fw-no-confl
  then show ?thesis using n-s unfolding full-def by blast
next
  case fw-confl
  then show ?thesis using n-s unfolding full-def by blast
next
  case fw-dec-confl
  have cdclW-merge-cp U V
    using n-s-bj by (metis cdclW-merge-cp.simps full-unfold fw-dec-confl(5))
  then have full1 cdclW-merge-cp T V
    unfolding full1-def by (metis fw-dec-confl(4) n-s-cp tranclp-unfold-end)
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case fw-dec-no-confl
  then have full cdclW-merge-cp T V
    using n-s-cp unfolding full-def by blast
  then have cdclW-merge-stgy S V using ⟨decide S T⟩ ⟨no-step cdclW-merge-cp S⟩ by auto
  then show ?thesis using n-s ⟨cdclW-merge-stgy** R S⟩ unfolding full-def by auto
next
  case cp-no-confl
  then have full cdclW-merge-cp R V
    by (simp add: full-def n-s-cp)
  then have R = V ∨ cdclW-merge-stgy++ R V
    by (metis (no-types) full-unfold fw-s-cp rtranclp-unfold tranclp-unfold-end)
  then show ?thesis
    by (simp add: full-def n-s rtranclp-unfold)
next
  case cp-confl
  have full cdclW-bj V V
    using n-s-bj unfolding full-def by blast
  then have full1 cdclW-merge-cp R V
    unfolding full1-def by (meson cdclW-merge-cp.conflict' cp-confl(1,2) n-s-cp
      rtranclp-into-tranclp1)
  then show ?thesis using n-s unfolding full-def by auto
qed
next
assume ?fw
then have cdclW** R V using rtranclp-mono[of cdclW-merge-stgy cdclW**]
  cdclW-merge-stgy-rtranclp-cdclW unfolding full-def by auto
then have inv': cdclW-all-struct-inv V using inv rtranclp-cdclW-all-struct-inv-inv by blast
have cdclW-s'** R V
  using ⟨?fw⟩ by (simp add: full-def inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s')
moreover have no-step cdclW-s' V
proof cases
  assume conflicting V = None
  then show ?thesis
    by (metis inv' ⟨full cdclW-merge-stgy R V⟩ full-def
      no-step-cdclW-merge-stgy-no-step-cdclW-s')
next
  assume confl-V: conflicting V ≠ None
  then have no-step cdclW-bj V
    using rtranclp-cdclW-merge-stgy-no-step-cdclW-bj by (meson ⟨full cdclW-merge-stgy R V⟩

```

```

    assms(1) full-def)
  then show ?thesis using confl-V by (fastforce simp: cdclW-s'.simps full1-def cdclW-cp.simps
    dest!: trancplD)
qed
ultimately show ?s' unfolding full-def by blast
qed

```

```

lemma full-cdclW-stgy-full-cdclW-merge:
  assumes
    conflicting R = None and
    inv: cdclW-all-struct-inv R
  shows full cdclW-stgy R V  $\longleftrightarrow$  full cdclW-merge-stgy R V
  by (simp add: assms(1) full-cdclW-s'-full-cdclW-merge-restart full-cdclW-stgy-iff-full-cdclW-s'
    inv)

```

```

lemma full-cdclW-merge-stgy-final-state-conclusive':
  fixes S' :: 'st
  assumes full: full cdclW-merge-stgy (init-state N) S'
  and no-d: distinct-mset-mset N
  shows (conflicting S' = Some {#}  $\wedge$  unsatisfiable (set-mset N))
     $\vee$  (conflicting S' = None  $\wedge$  trail S'  $\models_{asm}$  N  $\wedge$  satisfiable (set-mset N))
proof -
  have cdclW-all-struct-inv (init-state N)
    using no-d unfolding cdclW-all-struct-inv-def by auto
  moreover have conflicting (init-state N) = None
    by auto
  ultimately show ?thesis
    by (simp add: full full-cdclW-stgy-final-state-conclusive-from-init-state
      full-cdclW-stgy-full-cdclW-merge no-d)
qed

```

end

19.6 Adding Restarts

```

locale cdclW-restart =
  cdclW trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail
  add-init-clss
  add-learned-clss remove-clss update-backtrack-lvl update-conflicting init-state
  restart-state
for
  trail :: 'st  $\Rightarrow$  ('v, nat, 'v clause) marked-lits and
  init-clss :: 'st  $\Rightarrow$  'v clauses and
  learned-clss :: 'st  $\Rightarrow$  'v clauses and
  backtrack-lvl :: 'st  $\Rightarrow$  nat and
  conflicting :: 'st  $\Rightarrow$  'v clause option and

  cons-trail :: ('v, nat, 'v clause) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-init-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  add-learned-clss remove-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v clauses  $\Rightarrow$  'st and
  restart-state :: 'st  $\Rightarrow$  'st +

```

```

fixes  $f :: \text{nat} \Rightarrow \text{nat}$ 
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 $\text{cdcl}_W\text{-merge-with-restart}$ **where**

restart-step:

```

( $\text{cdcl}_W\text{-merge-stgy} \sim (\text{card } (\text{set-mset } (\text{learned-clss } T)) - \text{card } (\text{set-mset } (\text{learned-clss } S)))$ )  $S\ T$ 
 $\implies \text{card } (\text{set-mset } (\text{learned-clss } T)) - \text{card } (\text{set-mset } (\text{learned-clss } S)) > f\ n$ 
 $\implies \text{restart } T\ U \implies \text{cdcl}_W\text{-merge-with-restart } (S, n)\ (U, \text{Suc } n) \mid$ 

```

restart-full: $\text{full1 } \text{cdcl}_W\text{-merge-stgy } S\ T \implies \text{cdcl}_W\text{-merge-with-restart } (S, n)\ (T, \text{Suc } n)$

lemma $\text{cdcl}_W\text{-merge-with-restart } S\ T \implies \text{cdcl}_W\text{-merge-restart}^{**} (\text{fst } S)\ (\text{fst } T)$

by (*induction rule*: $\text{cdcl}_W\text{-merge-with-restart.induct}$)

```

(auto dest!:  $\text{relpowp-imp-rtranclp } \text{cdcl}_W\text{-merge-stgy-tranclp-cdcl}_W\text{-merge } \text{tranclp-into-rtranclp}$ 
 $\text{rtranclp-cdcl}_W\text{-merge-stgy-rtranclp-cdcl}_W\text{-merge } \text{rtranclp-cdcl}_W\text{-merge-tranclp-cdcl}_W\text{-merge-restart}$ 
 $\text{fw-r-rf } \text{cdcl}_W\text{-rf.restart}$ 
simp:  $\text{full1-def}$ )

```

lemma $\text{cdcl}_W\text{-merge-with-restart-rtranclp-cdcl}_W$:

$\text{cdcl}_W\text{-merge-with-restart } S\ T \implies \text{cdcl}_W^{**} (\text{fst } S)\ (\text{fst } T)$

by (*induction rule*: $\text{cdcl}_W\text{-merge-with-restart.induct}$)

```

(auto dest!:  $\text{relpowp-imp-rtranclp } \text{rtranclp-cdcl}_W\text{-merge-stgy-rtranclp-cdcl}_W\ \text{cdcl}_W.\text{rf}$ 
 $\text{cdcl}_W\text{-rf.restart } \text{tranclp-into-rtranclp } \text{simp: full1-def}$ )

```

lemma $\text{cdcl}_W\text{-merge-with-restart-increasing-number}$:

$\text{cdcl}_W\text{-merge-with-restart } S\ T \implies \text{snd } T = 1 + \text{snd } S$

by (*induction rule*: $\text{cdcl}_W\text{-merge-with-restart.induct}$) *auto*

lemma $\text{full1 } \text{cdcl}_W\text{-merge-stgy } S\ T \implies \text{cdcl}_W\text{-merge-with-restart } (S, n)\ (T, \text{Suc } n)$

using *restart-full* **by** *blast*

lemma $\text{cdcl}_W\text{-all-struct-inv-learned-clss-bound}$:

assumes *inv*: $\text{cdcl}_W\text{-all-struct-inv } S$

shows $\text{set-mset } (\text{learned-clss } S) \subseteq \text{simple-clss } (\text{atms-of-msu } (\text{init-clss } S))$

proof

fix C

assume C : $C \in \text{set-mset } (\text{learned-clss } S)$

have *distinct-mset* C

using C *inv* **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def } \text{distinct-cdcl}_W\text{-state-def } \text{distinct-mset-set-def}$

by *auto*

moreover **have** $\neg \text{tautology } C$

using C *inv* **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def } \text{cdcl}_W\text{-learned-clause-def}$ **by** *auto*

moreover

have $\text{atms-of } C \subseteq \text{atms-of-msu } (\text{learned-clss } S)$

using C **by** *auto*

then **have** $\text{atms-of } C \subseteq \text{atms-of-msu } (\text{init-clss } S)$

using *inv* **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def } \text{no-strange-atm-def}$ **by** *force*

moreover **have** *finite* $(\text{atms-of-msu } (\text{init-clss } S))$

using *inv* **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def}$ **by** *auto*

ultimately **show** $C \in \text{simple-clss } (\text{atms-of-msu } (\text{init-clss } S))$

using *distinct-mset-not-tautology-implies-in-simple-clss* *simple-clss-mono*

by *blast*

qed

lemma *cdcl_W-merge-with-restart-init-clss*:

*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: $\bigwedge i. \text{cdcl}_W\text{-merge-with-restart } (g\ i) (g\ (\text{Suc } i))$ **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-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*: $\bigwedge i. \text{snd } (g\ i) = i + \text{snd } (g\ 0)$

apply (*induct-tac i*)

apply *simp*

by (*metis Suc-eq-plus1-left add-Suc cdcl_W-merge-with-restart-increasing-number g*)

then have *snd-g-0*: $\bigwedge i. i > 0 \implies \text{snd } (g\ i) = i + \text{snd } (g\ 0)$

by *blast*

have *unbounded-f-g*: *unbounded ($\lambda i. f\ (\text{snd } (g\ i))$)*

using *f unfolding bounded-def* **by** (*metis add.commute f less-or-eq-imp-le snd-g
not-bounded-nat-exists-larger not-le le-iff-add*)

obtain *k* **where**

f-g-k: $f\ (\text{snd } (g\ k)) > \text{card } (\text{simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$ **and**

$k > \text{card } (\text{simple-clss } (\text{atms-of-msu } (\text{init-clss } (\text{fst } ?S))))$

using *not-bounded-nat-exists-larger[OF unbounded-f-g]* **by** *blast*

The following does not hold anymore with the non-strict version of cardinality in the definition.

{ fix *i*

assume *no-step cdcl_W-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**

$m: m = \text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss} (\text{fst } (g \ k))))$ **and**
 $m > f (\text{snd } (g \ k))$ **and**
 $\text{restart } T (\text{fst } (g \ (k+1)))$ **and**
 $\text{cdcl}_W\text{-merge-stgy}: (\text{cdcl}_W\text{-merge-stgy} \rightsquigarrow m) (\text{fst } (g \ k)) \ T$
using $g[\text{of } k] \ H[\text{of } \text{Suc } k]$ **by** $(\text{force simp: cdcl}_W\text{-merge-with-restart.simps full1-def})$
have $\text{cdcl}_W\text{-merge-stgy}^{**} (\text{fst } (g \ k)) \ T$
using $\text{cdcl}_W\text{-merge-stgy relpowp-imp-rtrancpl}$ **by** metis
then have $\text{cdcl}_W\text{-all-struct-inv } T$
using $\text{inv}[\text{of } k] \ \text{rtrancpl-cdcl}_W\text{-all-struct-inv-inv rtrancpl-cdcl}_W\text{-merge-stgy-rtrancpl-cdcl}_W$
by blast
moreover have $\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss} (\text{fst } (g \ k))))$
 $> \text{card} (\text{simple-clss} (\text{atms-of-msu} (\text{init-clss} (\text{fst } ?S))))$
unfolding $m[\text{symmetric}]$ **using** $\langle m > f (\text{snd } (g \ k)) \rangle \ f\text{-}g\text{-}k$ **by** linarith
then have $\text{card} (\text{set-mset} (\text{learned-clss } T))$
 $> \text{card} (\text{simple-clss} (\text{atms-of-msu} (\text{init-clss} (\text{fst } ?S))))$
by linarith
moreover
have $\text{init-clss} (\text{fst } (g \ k)) = \text{init-clss } T$
using $\langle \text{cdcl}_W\text{-merge-stgy}^{**} (\text{fst } (g \ k)) \ T \rangle \ \text{rtrancpl-cdcl}_W\text{-merge-stgy-rtrancpl-cdcl}_W$
 $\text{rtrancpl-cdcl}_W\text{-init-clss inv}$ **unfolding** $\text{cdcl}_W\text{-all-struct-inv-def}$ **by** blast
then have $\text{init-clss} (\text{fst } ?S) = \text{init-clss } T$
using $\text{init-g}[\text{of } k]$ **by** auto
ultimately show False
using $\text{cdcl}_W\text{-all-struct-inv-learned-clss-bound}$
by $(\text{simp add: } \langle \text{finite} (\text{atms-of-msu} (\text{init-clss} (\text{fst } (g \ 0)))) \rangle \ \text{simple-clss-finite}$
 $\text{card-mono leD})$
qed

lemma $\text{cdcl}_W\text{-merge-with-restart-distinct-mset-clauses}$:

assumes $\text{invR}: \text{cdcl}_W\text{-all-struct-inv} (\text{fst } R)$ **and**
 $\text{st}: \text{cdcl}_W\text{-merge-with-restart } R \ S$ **and**
 $\text{dist}: \text{distinct-mset} (\text{clauses} (\text{fst } R))$ **and**
 $R: \text{trail} (\text{fst } R) = []$
shows $\text{distinct-mset} (\text{clauses} (\text{fst } S))$
using $\text{assms}(2,1,3,4)$
proof (induction)
case $(\text{restart-full } S \ T)$
then show $?case$ **using** $\text{rtrancpl-cdcl}_W\text{-merge-stgy-distinct-mset-clauses}[\text{of } S \ T]$ **unfolding** full1-def
by $(\text{auto dest: rtrancpl-into-rtrancpl})$
next
case $(\text{restart-step } T \ S \ n \ U)$
then have $\text{distinct-mset} (\text{clauses } T)$
using $\text{rtrancpl-cdcl}_W\text{-merge-stgy-distinct-mset-clauses}[\text{of } S \ T]$ **unfolding** full1-def
by $(\text{auto dest: relpowp-imp-rtrancpl})$
then show $?case$ **using** $\langle \text{restart } T \ U \rangle$ **by** $(\text{metis clauses-restart distinct-mset-union fstI}$
 $\text{mset-le-exists-conv restart.cases state-eq-clauses})$
qed

inductive $\text{cdcl}_W\text{-with-restart}$ **where**

restart-step :

$(\text{cdcl}_W\text{-stgy} \rightsquigarrow (\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss } S)))) \ S \ T \implies$
 $\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss } S)) > f \ n \implies$
 $\text{restart } T \ U \implies$
 $\text{cdcl}_W\text{-with-restart} (S, n) (U, \text{Suc } n) \mid$
 $\text{restart-full: full1 cdcl}_W\text{-stgy } S \ T \implies \text{cdcl}_W\text{-with-restart} (S, n) (T, \text{Suc } n)$

```

lemma cdclW-with-restart-rtrancp-cdclW:
  cdclW-with-restart S T  $\implies$  cdclW** (fst S) (fst T)
apply (induction rule: cdclW-with-restart.induct)
by (auto dest!: relpoup-imp-rtrancp trancp-into-rtrancp fw-r-rf
      cdclW-rf.restart rtrancp-cdclW-stgy-rtrancp-cdclW cdclW-merge-restart-cdclW
      simp: full1-def)

lemma cdclW-with-restart-increasing-number:
  cdclW-with-restart S T  $\implies$  snd T = 1 + snd S
by (induction rule: cdclW-with-restart.induct) auto

lemma full1 cdclW-stgy S T  $\implies$  cdclW-with-restart (S, n) (T, Suc n)
using restart-full by blast

lemma cdclW-with-restart-init-clss:
  cdclW-with-restart S T  $\implies$  cdclW-M-level-inv (fst S)  $\implies$  init-clss (fst S) = init-clss (fst T)
using cdclW-with-restart-rtrancp-cdclW rtrancp-cdclW-init-clss by blast

lemma
  wf {(T, S). cdclW-all-struct-inv (fst S)  $\wedge$  cdclW-with-restart S T}
proof (rule ccontr)
assume  $\neg$  ?thesis
then obtain g where
  g:  $\bigwedge i. cdclW-with-restart (g i) (g (Suc i))$  and
  inv:  $\bigwedge i. cdclW-all-struct-inv (fst (g i))$ 
unfolding wf-iff-no-infinite-down-chain by fast
{ fix i
  have init-clss (fst (g i)) = init-clss (fst (g 0))
  apply (induction i)
  apply simp
  using g inv unfolding cdclW-all-struct-inv-def by (metis cdclW-with-restart-init-clss)
} note init-g = this
let ?S = g 0
have finite (atms-of-msu (init-clss (fst ?S)))
using inv unfolding cdclW-all-struct-inv-def by auto
have snd-g:  $\bigwedge i. snd (g i) = i + snd (g 0)$ 
apply (induct-tac i)
apply simp
by (metis Suc-eq-plus1-left add-Suc cdclW-with-restart-increasing-number g)
then have snd-g-0:  $\bigwedge i. i > 0 \implies snd (g i) = i + snd (g 0)$ 
by blast
have unbounded-f-g: unbounded ( $\lambda i. f (snd (g i))$ )
using f unfolding bounded-def by (metis add commute f less-or-eq-imp-le snd-g
      not-bounded-nat-exists-larger not-le le-iff-add)

obtain k where
  f-g-k:  $f (snd (g k)) > \text{card} (\text{simple-clss} (\text{atms-of-msu} (\text{init-clss} (\text{fst ?S}))))$  and
  k > card (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 cdclW-stgy (fst (g i))
  with g[of i]

```

```

have False
  proof (induction rule: cdclW-with-restart.induct)
    case (restart-step T S n) note H = this(1) and c = this(2) and n-s = this(4)
    obtain S' where cdclW-stgy S S'
      using H c by (metis gr-implies-not0 relpowp-E2)
    then show False using n-s by auto
  next
    case (restart-full S T)
    then show False unfolding full1-def by (auto dest: tranclpD)
  qed
} note H = this
obtain m T where
  m: m = card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k)))) and
  m > f (snd (g k)) and
  restart T (fst (g (k+1))) and
  cdclW-merge-stgy: (cdclW-stgy  $\sim$  m) (fst (g k)) T
  using g[of k] H[of Suc k] by (force simp: cdclW-with-restart.simps full1-def)
have cdclW-stgy** (fst (g k)) T
  using cdclW-merge-stgy relpowp-imp-rtranclp by metis
then have cdclW-all-struct-inv T
  using inv[of k] rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-stgy-rtranclp-cdclW by blast
moreover have card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k))))
  > card (simple-clss (atms-of-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 (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 (simp add: finite (atms-of-msu (init-clss (fst (g 0)))) simple-clss-finite
    card-mono leD)
qed

```

```

lemma cdclW-with-restart-distinct-mset-clauses:
  assumes invR: cdclW-all-struct-inv (fst R) and
  st: cdclW-with-restart R S and
  dist: distinct-mset (clauses (fst R)) and
  R: trail (fst R) = []
  shows distinct-mset (clauses (fst S))
  using assms(2,1,3,4)
proof (induction)
  case (restart-full S T)
  then show ?case using rtranclp-cdclW-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: tranclp-into-rtranclp)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T) using rtranclp-cdclW-stgy-distinct-mset-clauses[of S T]
    unfolding full1-def by (auto dest: relpowp-imp-rtranclp)

```

```

then show ?case using (restart T U) by (metis clauses-restart distinct-mset-union fstI
  mset-le-exists-conv restart.cases state-eq-clauses)
qed
end

locale luby-sequence =
  fixes ur :: nat
  assumes ur > 0
begin

lemma exists-luby-decomp:
  fixes i :: nat
  shows  $\exists k :: \text{nat}. (2^{k-1} \leq i \wedge i < 2^k - 1) \vee i = 2^k - 1$ 
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 $i = (2 :: 'a)^k - (1 :: 'a)$
- $\text{luby-sequence-core } (i - 2^{k-1} + 1)$, if $(2 :: 'a)^{k-1} \leq i$ and $i \leq (2 :: 'a)^k - (1 :: 'a)$

Then the sequence is then scaled by a constant unit run (called *ur* here), strictly positive.

```

function luby-sequence-core :: nat  $\Rightarrow$  nat where
  luby-sequence-core i =
    (if  $\exists k. i = 2^k - 1$ 
     then  $2^{k-1} - 1$ 
     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.  $2^k \leq i \wedge i < 2^{k+1}$ )
have  $2^{?k-1} \leq i \wedge i < 2^{?k}$ 
  apply (rule someI-ex)
  using 2 exists-luby-decomp by blast
then show ?case

proof -
  have  $\forall n \text{ na. } \neg (1::\text{nat}) \leq n \vee 1 \leq n \wedge \text{na}$ 
    by (meson one-le-power)
  then have f1:  $(1::\text{nat}) \leq 2^{?k-1}$ 
    using one-le-numeral by blast
  have f2:  $i - 2^{?k-1} + 2^{?k-1} = i$ 
    using  $2^{?k-1} \leq i \wedge i < 2^{?k}$  le-add-diff-inverse2 by blast
  have f3:  $2^{?k-1} \neq \text{Suc } 0$ 
    using f1  $2^{?k-1} \leq i \wedge i < 2^{?k}$  by linarith
  have  $2^{?k-1} - (1::\text{nat}) \neq 0$ 
    using  $2^{?k-1} \leq i \wedge i < 2^{?k}$  gr-implies-not0 by blast
  then have f4:  $2^{?k} \neq (1::\text{nat})$ 
    by linarith
  have f5:  $\forall n \text{ na. if na = 0 then } (n::\text{nat}) \wedge \text{na} = 1 \text{ else } n \wedge \text{na} = n * n \wedge (\text{na} - 1)$ 
    by (simp add: power-eq-if)
  then have ?k  $\neq 0$ 
    using f4 by meson
  then have  $2^{?k-1} \neq \text{Suc } 0$ 
    using f5 f3 by presburger
  then have  $\text{Suc } 0 < 2^{?k-1}$ 
    using f1 by linarith
  then show ?thesis
    using f2 less-than-iff by presburger
qed
qed

declare luby-sequence-core.simps[simp del]

lemma two-pover-n-eq-two-power-n'-eq:
  assumes H:  $(2::\text{nat}) \wedge (k::\text{nat}) - 1 = 2^{k'} - 1$ 
  shows  $k' = k$ 
proof -
  have  $(2::\text{nat}) \wedge (k::\text{nat}) = 2^{k'}$ 
    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 ^ (k - Suc 0) ≤ i and
  k': i < 2 ^ k' - Suc 0 and
  k-k': k > k'
shows False
proof -
  have 2 ^ k' - Suc 0 < 2 ^ (k - Suc 0)
    using k-k' less-eq-Suc-le by auto
  then show ?thesis
    using H k' by linarith
qed

```

lemma *luby-sequence-core-not-two-power-minus-one:*

```

assumes
  k-i: 2 ^ (k - 1) ≤ i and
  i-k: i < 2 ^ k - 1
shows luby-sequence-core i = luby-sequence-core (i - 2 ^ (k - 1) + 1)
proof -
  have H: ¬ (∃ ka. i = 2 ^ ka - 1)
  proof (rule ccontr)
    assume ¬ ?thesis
    then obtain k':nat where k': i = 2 ^ k' - 1 by blast
    have (2::nat) ^ k' - 1 < 2 ^ k - 1
      using i-k unfolding k'.
    then have (2::nat) ^ k' < 2 ^ k
      by linarith
    then have k' < k
      by simp
    have 2 ^ (k - 1) ≤ 2 ^ k' - (1::nat)
      using k-i unfolding k'.
    then have (2::nat) ^ (k-1) < 2 ^ k'
      by (metis Suc-diff-1 not-le not-less-eq zero-less-numeral zero-less-power)
    then have k-1 < k'
      by simp

    show False using ⟨k' < k⟩ ⟨k-1 < k'⟩ by linarith
  qed
  have ∧k k'. 2 ^ (k - Suc 0) ≤ i ⟹ i < 2 ^ k - Suc 0 ⟹ 2 ^ (k' - Suc 0) ≤ i ⟹
    i < 2 ^ k' - Suc 0 ⟹ k = k'
    by (meson different-luby-decomposition-false linorder-neqE-nat)
  then have k: (SOME k. 2 ^ (k - Suc 0) ≤ i ∧ i < 2 ^ k - Suc 0) = k
    using k-i i-k by auto
  show ?thesis
    apply (subst luby-sequence-core.simps[of i], subst H)

```

by (simp add: k)
qed

lemma *unbounded-luby-sequence-core: unbounded luby-sequence-core*
unfolding *bounded-def*

proof

assume $\exists b. \forall n. \text{luby-sequence-core } n \leq b$
then obtain *b* where $b: \bigwedge n. \text{luby-sequence-core } n \leq b$
by *metis*
have *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*[of $2^{(b+1)} - 1$] by *linarith*

qed

abbreviation *luby-sequence* :: *nat* \Rightarrow *nat* **where**
luby-sequence *n* \equiv *ur* * *luby-sequence-core* *n*

lemma *bounded-luby-sequence: unbounded luby-sequence*
using *bounded-const-product*[of *ur*] *luby-sequence-axioms*
luby-sequence-def *unbounded-luby-sequence-core* **by** *blast*

lemma *luby-sequence-core-0: luby-sequence-core 0 = 1*

proof –

have *0*: $(0::\text{nat}) = 2^0 - 1$
by *auto*
show *?thesis*
by (*subst 0*, *subst luby-sequence-core-two-power-minus-one*) *simp*

qed

lemma *luby-sequence-core* *n* ≥ 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** $\text{Suc } n < 2^k - 1$
| (*pow2*) *k* **where** $\text{Suc } n = 2^k - \text{Suc } 0$
using *exists-luby-decomp*[of *Suc n*] **by** *auto*

then show *?case*

proof *cases*

case *pow2*

show *?thesis*

using *luby-sequence-core-two-power-minus-one* *pow2* **by** *auto*

next

case *interv*

have *n*: $\text{Suc } n - 2^{(k-1)} + 1 < \text{Suc } n$

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 trail init-clss learned-clss backtrack-lvl conflicting cons-trail tl-trail
  add-init-clss
  add-learned-clss remove-clss update-backtrack-lvl update-conflicting init-state
  restart-state
for
  ur :: nat and
  trail :: 'st  $\Rightarrow$  ('v, 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 option and
  cons-trail :: ('v, nat, 'v clause) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-init-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  add-learned-clss remove-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v clauses  $\Rightarrow$  'st and
  restart-state :: 'st  $\Rightarrow$  'st
begin

sublocale cdclW-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
begin

```

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl_W-all-struct-inv*

```

definition cdclW-stgy-invariant where
  cdclW-stgy-invariant  $S \longleftrightarrow$ 
    conflict-is-false-with-level  $S$ 
     $\wedge$  no-clause-is-false  $S$ 
     $\wedge$  no-smaller-confl  $S$ 
     $\wedge$  no-clause-is-false  $S$ 

```

lemma *cdcl_W-stgy-cdcl_W-stgy-invariant*:
assumes
cdcl_W: cdcl_W-stgy S T and
inv-s: cdcl_W-stgy-invariant S and
inv: cdcl_W-all-struct-inv S
shows
cdcl_W-stgy-invariant T
unfolding *cdcl_W-stgy-invariant-def cdcl_W-all-struct-inv-def* **apply** *standard*
apply (rule *cdcl_W-stgy-ex-lit-of-max-level*[of *S*])
using *assms unfolding cdcl_W-stgy-invariant-def cdcl_W-all-struct-inv-def* **apply** *auto*[7]
apply *standard*
using *cdcl_W cdcl_W-stgy-not-non-negated-init-clss* **apply** *blast*
apply *standard*
apply (rule *cdcl_W-stgy-no-smaller-conflict-inv*)
using *assms unfolding cdcl_W-stgy-invariant-def cdcl_W-all-struct-inv-def* **apply** *auto*[4]
using *cdcl_W cdcl_W-stgy-not-non-negated-init-clss* **by** *auto*

lemma *rtrancp-cdcl_W-stgy-cdcl_W-stgy-invariant*:
assumes
*cdcl_W: cdcl_W-stgy** S T and*
inv-s: cdcl_W-stgy-invariant S and
inv: cdcl_W-all-struct-inv S
shows
cdcl_W-stgy-invariant T
using *assms* **apply** (induction)
apply *simp*
using *cdcl_W-stgy-cdcl_W-stgy-invariant rtrancp-cdcl_W-all-struct-inv-inv*
rtrancp-cdcl_W-stgy-rtrancp-cdcl_W **by** *blast*

abbreviation *decr-bt-lvl* **where**
decr-bt-lvl S \equiv *update-backtrack-lvl (backtrack-lvl S - 1) S*

When we add a new clause, we reduce the trail until we get to the first literal included in *C*. Then we can mark the conflict.

fun *cut-trail-wrt-clause* **where**
cut-trail-wrt-clause C [] S = S |
cut-trail-wrt-clause C (Marked L - # M) S =
(if -L \in # C then S
else cut-trail-wrt-clause C M (decr-bt-lvl (tl-trail S))) |
cut-trail-wrt-clause C (Propagated L - # M) S =
(if -L \in # C then S
else cut-trail-wrt-clause C M (tl-trail S))

definition *add-new-clause-and-update* :: *'v literal multiset \Rightarrow 'st \Rightarrow 'st* **where**
add-new-clause-and-update C S =
(if trail S \models as C Not C
then update-conflicting (Some 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:
 $\exists M. trail\ S = M @ trail\ (cut-trail-wrt-clause\ C\ (trail\ S)\ S)$
proof (induction trail S arbitrary:S rule: marked-lit-list-induct)
 case nil
 then show ?case **by** simp
next
 case (marked L l M) **note** IH = this(1)[of decr-bt-lvl (tl-trail S)] **and** M = this(2)[symmetric]
 then show ?case **using** Cons-eq-appendI **by** fastforce+
next
 case (proped L l M) **note** IH = this(1)[of tl-trail S] **and** M = this(2)[symmetric]
 then show ?case **using** Cons-eq-appendI **by** fastforce+
qed

lemma n-dup-no-dup-trail-cut-trail-wrt-clause[simp]:
assumes n-d: no-dup (trail T)
shows no-dup (trail (cut-trail-wrt-clause C (trail T) T))
proof –
obtain M **where**
 $M: trail\ T = M @ trail\ (cut-trail-wrt-clause\ C\ (trail\ T)\ T)$
using trail-cut-trail-wrt-clause[of T C] **by** auto
show ?thesis
using n-d **unfolding** arg-cong[OF M, of no-dup] **by** auto
qed

lemma cut-trail-wrt-clause-backtrack-lvl-length-marked:
assumes
 $backtrack-lvl\ T = length\ (get-all-levels-of-marked\ (trail\ T))$
shows
 $backtrack-lvl\ (cut-trail-wrt-clause\ C\ (trail\ T)\ T) =$
 $length\ (get-all-levels-of-marked\ (trail\ (cut-trail-wrt-clause\ C\ (trail\ T)\ T)))$
using assms
proof (induction trail T arbitrary:T rule: marked-lit-list-induct)
 case nil
 then show ?case **by** simp
next
 case (marked L l M) **note** IH = this(1)[of decr-bt-lvl (tl-trail T)] **and** M = this(2)[symmetric]
and bt = this(3)
 then show ?case **by** auto
next
 case (proped L l M) **note** IH = this(1)[of tl-trail T] **and** M = this(2)[symmetric] **and** bt = this(3)
 then show ?case **by** auto
qed

lemma cut-trail-wrt-clause-get-all-levels-of-marked:
assumes get-all-levels-of-marked (trail T) = rev [Suc 0..
 $Suc\ (length\ (get-all-levels-of-marked\ (trail\ T)))]$
shows
 $get-all-levels-of-marked\ (trail\ ((cut-trail-wrt-clause\ C\ (trail\ T)\ T))) = rev\ [Suc\ 0.. $$$

```

    Suc (length (get-all-levels-of-marked (trail ((cut-trail-wrt-clause C (trail T) T))))))
  using assms
proof (induction trail T arbitrary:T rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]
    and bt = this(3)
  then show ?case by (cases count C L = 0) auto
next
  case (proped L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric] and bt = this(3)
  then show ?case by (cases count C L = 0) auto
qed

lemma cut-trail-wrt-clause-CNot-trail:
  assumes trail T  $\models_{as}$  CNot C
  shows
    (trail ((cut-trail-wrt-clause C (trail T) T)))  $\models_{as}$  CNot C
  using assms
proof (induction trail T arbitrary:T rule: marked-lit-list-induct)
  case nil
  then show ?case by simp
next
  case (marked L l M) note IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]
    and bt = this(3)
  show ?case
  proof (cases count C (-L) = 0)
    case False
    then show ?thesis
      using IH M bt by (auto simp: true-annots-true-cl)
  next
    case True
    obtain mma :: 'v literal multiset where
      f6: (mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow M \models_a mma\} \longrightarrow M \models_{as} \{\{\#- l\# \mid l. l \in \# C\}$ )
      using true-annots-def by moura
    have mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow \text{trail } T \models_a mma\}$ 
      using CNot-def M bt by (metis (no-types) true-annots-def)
    then have M  $\models_{as} \{\{\#- l\# \mid l. l \in \# C\}$ 
      using f6 True M bt by force
    then show ?thesis
      using IH true-annots-true-cl M by (auto simp: CNot-def)
  qed
next
  case (proped L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric] and bt = this(3)
  show ?case
  proof (cases count C (-L) = 0)
    case False
    then show ?thesis
      using IH M bt by (auto simp: true-annots-true-cl)
  next
    case True
    obtain mma :: 'v literal multiset where
      f6: (mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow M \models_a mma\} \longrightarrow M \models_{as} \{\{\#- l\# \mid l. l \in \# C\}$ )
      using true-annots-def by moura
    have mma  $\in$   $\{\{\#- l\# \mid l. l \in \# C\} \longrightarrow \text{trail } T \models_a mma\}$ 

```

```

    using CNot-def M bt by (metis (no-types) true-annots-def)
  then have M  $\models_{as}$   $\{\{\# - l\# \} \mid l. l \in \# C\}$ 
    using f6 True M bt by force
  then show ?thesis
    using IH true-annots-true-clss M by (auto simp: CNot-def)
qed
qed

```

lemma *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*:

```

(( $\forall L \in \# C. -L \notin \text{ lits-of } (\text{trail } T)$ )  $\wedge$   $\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T) = []$ )
 $\vee$  ( $-\text{lit-of } (\text{hd } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))) \in \# C$ 
 $\wedge$   $\text{length } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)) \geq 1$ )

```

using *assms*

proof (*induction trail T arbitrary: T rule: marked-lit-list-induct*)

case *nil*

then show ?case by *simp*

next

case (*marked L l M*) note *IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]*

then show ?case by *simp force*

next

case (*proped L l M*) note *IH = this(1)[of tl-trail T] and M = this(2)[symmetric]*

then show ?case by *simp force*

qed

We can fully run *cdcl_W*-s or add a clause. Remark that we use *cdcl_W*-s to avoid an explicit *skip*, *resolve*, and *backtrack* normalisation to get rid of the conflict *C* if possible.

inductive *incremental-cdcl_W* :: '*st* \Rightarrow '*st* \Rightarrow bool **for** *S* **where**

add-confli:

trail S \models_{asm} *init-clss S* \Rightarrow *distinct-mset C* \Rightarrow *conflicting S = None* \Rightarrow

trail S \models_{as} *CNot C* \Rightarrow

full cdcl_W-stgy

(*update-conflicting (Some C) (add-init-clss C (cut-trail-wrt-clause C (trail S) S))*) *T* \Rightarrow
incremental-cdcl_W S T |

add-no-confli:

trail S \models_{asm} *init-clss S* \Rightarrow *distinct-mset C* \Rightarrow *conflicting S = None* \Rightarrow

\neg *trail S* \models_{as} *CNot C* \Rightarrow

full cdcl_W-stgy (add-init-clss C S) T \Rightarrow

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 \Rightarrow *add-learned-clss S* ($\{\#x\# \} + A$) (*add-learned-clss x T*)

declare *add-learned-clss.intros*[*intro*]

lemma *Ex-add-learned-clss*:

$\exists T. \text{ add-learned-clss } S A T$

by (*induction A arbitrary: S rule: multiset-induct*) (*auto simp: union-commute[of - $\{\#-\#\}$]*)

lemma *add-learned-clss-trail*:

assumes *add-learned-clss S U T* **and** *no-dup (trail S)*

shows *trail T = trail S*

using *assms* **by** (*induction rule: add-learned-clss.induct*) (*simp-all add: ac-simps*)

lemma *add-learned-clss-learned-clss*:

assumes *add-learned-clss* $S\ U\ T$ **and** *no-dup* (*trail* S)
shows *learned-clss* $T = U + \text{learned-clss } S$
using *assms* **by** (*induction rule*: *add-learned-clss.induct*)
(auto simp: ac-simps dest: add-learned-clss-trail)

lemma *add-learned-clss-init-clss*:
assumes *add-learned-clss* $S\ U\ T$ **and** *no-dup* (*trail* S)
shows *init-clss* $T = \text{init-clss } S$
using *assms* **by** (*induction rule*: *add-learned-clss.induct*)
(auto simp: ac-simps dest: add-learned-clss-trail)

lemma *add-learned-clss-conflicting*:
assumes *add-learned-clss* $S\ U\ T$ **and** *no-dup* (*trail* S)
shows *conflicting* $T = \text{conflicting } S$
using *assms* **by** (*induction rule*: *add-learned-clss.induct*)
(auto simp: ac-simps dest: add-learned-clss-trail)

lemma *add-learned-clss-backtrack-lvl*:
assumes *add-learned-clss* $S\ U\ T$ **and** *no-dup* (*trail* S)
shows *backtrack-lvl* $T = \text{backtrack-lvl } S$
using *assms* **by** (*induction rule*: *add-learned-clss.induct*)
(auto simp: ac-simps dest: add-learned-clss-trail)

lemma *add-learned-clss-init-state-mempty[dest!]*:
add-learned-clss (*init-state* N) $\{\#\}$ $T \implies T = \text{init-state } N$
by (*cases rule*: *add-learned-clss.cases*) *(auto simp: add-learned-clss.cases)*

For multiset larger than 1 element, there is no way to know in which order the clauses are added.
But contrary to a definition *fold-mset*, there is an element.

lemma *add-learned-clss-init-state-single[dest!]*:
add-learned-clss (*init-state* N) $\{\#C\#\}$ $T \implies T = \text{add-learned-clss } C\ (\text{init-state } N)$
by (*induction* $\{\#C\#\}$ T *rule*: *add-learned-clss.induct*)
(auto simp: add-learned-clss.cases ac-simps union-is-single split: split-if-asm)

thm *rtrancp-cdcl_W-stgy-no-smaller-conf-inv cdcl_W-stgy-final-state-conclusive*

lemma *cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv*:

assumes
inv-T: *cdcl_W-all-struct-inv* T **and**
tr-T-N[simp]: *trail* $T \models_{\text{asm}} N$ **and**
tr-C[simp]: *trail* $T \models_{\text{as}} C \text{Not } C$ **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 $?T = \text{update-conflicting } (\text{Some } C) (\text{add-init-clss } C (\text{cut-trail-wrt-clause } C (\text{trail } T) T))$

obtain M **where**

M : *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** *blast*

have $H[\text{dest}]$: $\bigwedge x. x \in \text{lits-of } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)) \implies$
 $x \in \text{lits-of } (\text{trail } T)$

using *inv-T arg-cong[OF M, of lits-of]* **by** *auto*

have $H'[\text{dest}]$: $\bigwedge x. x \in \text{set } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)) \implies x \in \text{set } (\text{trail } T)$

using *inv-T arg-cong[OF M, of set]* **by** *auto*

have $H\text{-proped}$: $\bigwedge x. x \in \text{set } (\text{get-all-mark-of-propagated } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))) \implies x \in \text{set } (\text{get-all-mark-of-propagated } (\text{trail } T))$

```

using inv-T arg-cong[OF M, of get-all-mark-of-propagated] by auto

have [simp]: no-strange-atm ?T
  using inv-T unfolding cdclW-all-struct-inv-def no-strange-atm-def add-new-clause-and-update-def
  cdclW-M-level-inv-def
  by (auto dest!: H H')

have M-lev: cdclW-M-level-inv T
  using inv-T unfolding cdclW-all-struct-inv-def by blast
then have no-dup (M @ trail (cut-trail-wrt-clause C (trail T) T))
  unfolding cdclW-M-level-inv-def unfolding M[symmetric] by auto
then have [simp]: no-dup (trail (cut-trail-wrt-clause C (trail T) T))
  by auto

have consistent-interp (lits-of (M @ trail (cut-trail-wrt-clause C (trail T) T)))
  using M-lev unfolding cdclW-M-level-inv-def unfolding M[symmetric] by auto
then have [simp]: consistent-interp (lits-of (trail (cut-trail-wrt-clause C (trail T) T)))
  unfolding consistent-interp-def by auto

have [simp]: cdclW-M-level-inv ?T

  using M-lev cut-trail-wrt-clause-get-all-levels-of-marked[of T C]
  unfolding cdclW-M-level-inv-def by (auto dest: H H')
  simp: M-lev cdclW-M-level-inv-def cut-trail-wrt-clause-backtrack-lvl-length-marked)

have [simp]:  $\bigwedge s. s \in \# \text{ learned-clss } T \implies \neg \text{tautology } s$ 
  using inv-T unfolding cdclW-all-struct-inv-def by auto

have distinct-cdclW-state T
  using inv-T unfolding cdclW-all-struct-inv-def by auto
then have [simp]: distinct-cdclW-state ?T
  unfolding distinct-cdclW-state-def by auto

have cdclW-conflicting T
  using inv-T unfolding cdclW-all-struct-inv-def by auto
have trail ?T  $\models_{as}$  CNot 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  $(a, b) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } ?T))$ 
    from in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend[OF this, of M]
    obtain b' where
       $(a, b' @ b) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } T))$ 
      using M by auto
    then have  $\text{unmark } a \cup \text{set-mset } (\text{init-clss } T) \models_{ps} \text{unmark } (b' @ b)$ 

```

```

    using decomp-T unfolding all-decomposition-implies-def by fastforce
  then have unmark a ∪ set-mset (init-clss ?T)
     $\models_{ps}$  unmark (b @ b')
    by (simp add: Un-commute)
  then show unmark a ∪ set-mset (init-clss ?T)
     $\models_{ps}$  unmark b
    by (auto simp: image-Un)
qed

have [simp]: cdclW-learned-clause ?T
  using inv-T unfolding cdclW-all-struct-inv-def cdclW-learned-clause-def
  by (auto dest!: H-proped simp: clauses-def)
show ?thesis
  using (all-decomposition-implies-m (init-clss ?T)
  (get-all-marked-decomposition (trail ?T)))
  unfolding cdclW-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
qed

lemma cdclW-all-struct-inv-add-new-clause-and-update-cdclW-stgy-inv:
  assumes
    inv-s: cdclW-stgy-invariant T and
    inv: cdclW-all-struct-inv T and
    tr-T-N[simp]: trail T  $\models_{asm}$  N and
    tr-C[simp]: trail T  $\models_{as}$  CNot C and
    [simp]: distinct-mset C
  shows cdclW-stgy-invariant (add-new-clause-and-update C T) (is cdclW-stgy-invariant ?T')
proof -
  have cdclW-all-struct-inv ?T'
    using cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv assms by blast
  then have
    no-dup-cut-T[simp]: no-dup (trail (cut-trail-wrt-clause C (trail T) T)) and
    n-d[simp]: no-dup (trail T)
    using cdclW-M-level-inv-decomp(2) cdclW-all-struct-inv-def inv
    n-dup-no-dup-trail-cut-trail-wrt-clause by blast+
  then have trail (add-new-clause-and-update C T)  $\models_{as}$  CNot C
    by (simp add: add-new-clause-and-update-def cut-trail-wrt-clause-CNot-trail
    cdclW-M-level-inv-def cdclW-all-struct-inv-def)
  obtain MT where
    MT: trail T = MT @ trail (cut-trail-wrt-clause C (trail T) T)
    using trail-cut-trail-wrt-clause by blast
  consider
    (false)  $\forall L \in \#C. - L \notin \text{ lits-of } (trail T)$  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 ≤ 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 cdclW-stgy-invariant-def no-smaller-confl-def
      cdclW-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
  next
    case not-false note C = 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.. $1 + \text{length (get-all-levels-of-marked (trail (add-new-clause-and-update C T)))}$ ]
  using  $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$  unfolding  $\text{cdcl}_W\text{-all-struct-inv-def}$   $\text{cdcl}_W\text{-M-level-inv-def}$ 
  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  $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$  unfolding  $\text{cdcl}_W\text{-all-struct-inv-def}$   $\text{cdcl}_W\text{-M-level-inv-def}$ 
    by (auto simp: add-new-clause-and-update-def)
moreover
  have no-dup (trail (cut-trail-wrt-clause C (trail T) T))
    using  $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$  unfolding  $\text{cdcl}_W\text{-all-struct-inv-def}$   $\text{cdcl}_W\text{-M-level-inv-def}$ 
    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 (trail (cut-trail-wrt-clause C (trail T) T)) (-?L)
  = 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
     $\langle \text{atm-of } ?L \notin \text{atm-of ' lits-of (tl (trail (cut-trail-wrt-clause C (trail T) T)))} \rangle$ ,
    of [hd (trail (cut-trail-wrt-clause C (trail T) T))]]

  apply (cases trail (add-init-cls C (cut-trail-wrt-clause C (trail T) T));
    cases hd (trail (cut-trail-wrt-clause C (trail T) T)))
  using l by (auto split: split-if-asm
    simp: rev-swap[symmetric] add-new-clause-and-update-def)

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  $\langle \text{cdcl}_W\text{-all-struct-inv } ?T' \rangle$  unfolding  $\text{cdcl}_W\text{-all-struct-inv-def}$   $\text{cdcl}_W\text{-M-level-inv-def}$ 
  by (auto simp: add-new-clause-and-update-def)

have [simp]: no-smaller-confl (update-conflicting (Some C)
  (add-init-cls 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  $\text{cdcl}_W\text{-stgy-invariant-def}$  by auto
  have (MT @ M') @ Marked K i  $\#$  M = trail T
    using MT 1(1) by auto
  thus False using D-T  $\langle \text{no-smaller-confl } T \rangle$  1(3) unfolding no-smaller-confl-def by blast
next
  case DC note -[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) ∈ 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 cdclW-stgy S T and
    inv-s: cdclW-stgy-invariant S and
    inv: cdclW-all-struct-inv S
  shows conflicting T = Some {#} ∧ unsatisfiable (set-mset (init-clss S))
    ∨ conflicting T = None ∧ trail T ⊨asm init-clss S ∧ satisfiable (set-mset (init-clss S))
proof -
  have no-step cdclW-stgy T
    using full unfolding full-def by blast
  moreover have cdclW-all-struct-inv T and inv-s: cdclW-stgy-invariant T
    apply (metis cdclW.rtranclp-cdclW-stgy-rtranclp-cdclW cdclW-axioms full full-def inv
      rtranclp-cdclW-all-struct-inv-inv)
    by (metis full full-def inv inv-s rtranclp-cdclW-stgy-cdclW-stgy-invariant)
  ultimately have conflicting T = Some {#} ∧ unsatisfiable (set-mset (init-clss T))
    ∨ conflicting T = None ∧ trail T ⊨asm init-clss T
    using cdclW-stgy-final-state-conclusive[of T] full
    unfolding cdclW-all-struct-inv-def cdclW-stgy-invariant-def full-def by fast
  moreover have consistent-interp (lits-of (trail T))
    using ⟨cdclW-all-struct-inv T⟩ unfolding cdclW-all-struct-inv-def cdclW-M-level-inv-def
    by auto
  moreover have init-clss S = init-clss T
    using inv unfolding cdclW-all-struct-inv-def
    by (metis rtranclp-cdclW-stgy-no-more-init-clss full full-def)
  ultimately show ?thesis
    by (metis satisfiable-carac' true-annot-def true-annots-def true-clss-def)
qed

```

lemma *incremental-cdcl_W-inv*:

```

assumes
  inc: incremental-cdclW S T and
  inv: cdclW-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows
  cdclW-all-struct-inv T and
  cdclW-stgy-invariant T
using inc
proof (induction)
case (add-confl C T)
let ?T = (update-conflicting (Some C) (add-init-cls C (cut-trail-wrt-clause C (trail S) S)))
have cdclW-all-struct-inv ?T and inv-s-T: cdclW-stgy-invariant ?T
  using add-confl.hyps(1,2,4) add-new-clause-and-update-def
  cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv inv apply auto[1]
  using add-confl.hyps(1,2,4) add-new-clause-and-update-def
  cdclW-all-struct-inv-add-new-clause-and-update-cdclW-stgy-inv inv s-inv by auto
case 1 show ?case
  by (metis add-confl.hyps(1,2,4,5) add-new-clause-and-update-def
    cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv
    rtranclp-cdclW-all-struct-inv-inv rtranclp-cdclW-stgy-rtranclp-cdclW 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
    cdclW-all-struct-inv-add-new-clause-and-update-cdclW-all-struct-inv full-def inv
    rtranclp-cdclW-stgy-cdclW-stgy-invariant)
next
case (add-no-confl C T)
case 1
have cdclW-all-struct-inv (add-init-cls C S)
  using inv distinct-mset C unfolding cdclW-all-struct-inv-def no-strange-atm-def
  cdclW-M-level-inv-def distinct-cdclW-state-def cdclW-conflicting-def cdclW-learned-clause-def
  by (auto simp: all-decomposition-implies-insert-single clauses-def)
then show ?case
  using add-no-confl(5) unfolding full-def by (auto intro: rtranclp-cdclW-stgy-cdclW-all-struct-inv)
case 2 have cdclW-stgy-invariant (add-init-cls C S)
  using s-inv ¬ trail S ⊨as CNot C inv unfolding cdclW-stgy-invariant-def no-smaller-confl-def
  eq-commute[of - trail -] cdclW-M-level-inv-def cdclW-all-struct-inv-def
  by (auto simp: true-annots-true-cls-def-iff-negation-in-model clauses-def split: split-if-asm)
then show ?case
  by (metis cdclW-all-struct-inv (add-init-cls C S) add-no-confl.hyps(5) full-def
    rtranclp-cdclW-stgy-cdclW-stgy-invariant)
qed

```

lemma *rtranclp-incremental-cdcl_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-cdcl_W* *S T* **and**

inv: *cdcl_W-all-struct-inv* *S* **and**

s-inv: *cdcl_W-stgy-invariant* *S*

shows *conflicting* *T* = *Some* $\{\#\}$ \wedge *unsatisfiable* (*set-mset* (*init-cls* *T*))

\vee *conflicting* *T* = *None* \wedge *trail* *T* \models_{asm} *init-cls* *T* \wedge *satisfiable* (*set-mset* (*init-cls* *T*))

using *inc* **apply** *induction*

apply (*metis* *Nitpick.rtranclp-unfold* *add-confl* *full-cdcl_W-stgy-inv-normal-form* *full-def* *incremental-cdcl_W-inv(1)* *incremental-cdcl_W-inv(2)* *inv s-inv*)

by (*metis* (*full-types*) *rtranclp-unfold* *add-no-confl* *full-cdcl_W-stgy-inv-normal-form* *full-def* *incremental-cdcl_W-inv(1)* *incremental-cdcl_W-inv(2)* *inv s-inv*)

lemma *tranclp-incremental-correct*:

assumes

inc: *incremental-cdcl_W⁺⁺* *S T* **and**

inv: *cdcl_W-all-struct-inv* *S* **and**

s-inv: *cdcl_W-stgy-invariant* *S*

shows *conflicting* *T* = *Some* $\{\#\}$ \wedge *unsatisfiable* (*set-mset* (*init-cls* *T*))

\vee *conflicting* *T* = *None* \wedge *trail* *T* \models_{asm} *init-cls* *T* \wedge *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-cdcl_W-inv* *s-inv* *tranclp-into-rtranclp*)

lemma *blocked-induction-with-marked*:

assumes

n-d: *no-dup* (*L* $\#$ *M*) **and**

nil: *P* \square **and**

append: $\bigwedge M L M'. P M \implies is_marked L \implies \forall m \in set M'. \neg is_marked m \implies no_dup (L \# M' @ M) \implies$

P (*L* $\#$ *M' @ M*) **and**

L: *is-marked* *L*

shows

P (*L* $\#$ *M*)

using *n-d* *L*

proof (*induction* *card* $\{L' \in set M. is_marked L'\}$ *arbitrary*: *L M*)

case 0 **note** *n* = *this*(1) **and** *n-d* = *this*(2) **and** *L* = *this*(3)

then have $\forall m \in set M. \neg is_marked m$ **by** *auto*

then show ?*case* **using** *append*[*of* \square *L M*] *L nil n-d* **by** *auto*

next

case (*Suc* *n*) **note** *IH* = *this*(1) **and** *n* = *this*(2) **and** *n-d* = *this*(3) **and** *L* = *this*(4)

have $\exists L' \in set M. is_marked L'$

proof (*rule* *ccontr*)

assume $\neg ?thesis$

then have *H*: $\{L' \in set M. is_marked L'\} = \{\}$

by *auto*

show *False* **using** *n* *unfolding* *H* **by** *auto*

qed

then obtain *L' M' M''* **where**

M: *M* = *M' @ L' # M''* **and**

L': *is-marked* *L'* **and**

nm: $\forall m \in set M'. \neg is_marked m$

```

  by (auto elim!: split-list-first-propE)
have Suc n = card {L' ∈ set M. is-marked L'}
  using n .
moreover have {L' ∈ set M. is-marked L'} = {L'} ∪ {L' ∈ set M''. is-marked L'}
  using nm L' n-d unfolding M by auto
moreover have L' ∉ {L' ∈ set M''. is-marked L'}
  using n-d unfolding M by auto
ultimately have n = card {L'' ∈ set M''. is-marked L''}
  using n L' by auto
then have P (L' # M'') using IH L' n-d M by auto
then show ?case using append[of L' # M'' L M] nm L n-d unfolding M by blast
qed

```

lemma *trail-bloc-induction*:

```

assumes
  n-d: no-dup M and
  nil: P [] and
  append:  $\bigwedge M L M'. P M \implies \text{is-marked } L \implies \forall m \in \text{set } M'. \neg \text{is-marked } m \implies \text{no-dup } (L \# M' @ M) \implies$ 
     $P (L \# M' @ M)$  and
  append-nm:  $\bigwedge M' M''. P M' \implies M = M'' @ M' \implies \forall m \in \text{set } M''. \neg \text{is-marked } m \implies P M$ 
shows
  P M
proof (cases {L' ∈ set M. is-marked L'} = {})
  case True
  then show ?thesis using append-nm[of [] M] nil by auto
next
  case False
  then have  $\exists L' \in \text{set } M. \text{is-marked } L'$ 
    by auto
  then obtain L' M' M'' where
    M: M = M' @ L' # M'' and
    L': 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'  $\implies M = M'' @ M' \implies (\forall m \in \text{set } M''. \neg \text{is-marked } m) \implies \text{Tcons } M (M'' @ M') |$ 
    Tcons M M'  $\implies \text{is-marked } L \implies M = M''' @ L \# M'' @ M' \implies (\forall m \in \text{set } M''. \neg \text{is-marked } m) \implies$ 
      Tcons M (L # M'' @ M')

```

```

lemma Tcons-same-end: Tcons M M'  $\implies \exists M''. M = M'' @ M'$ 
  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*) (*unwatched: 'v*)

abbreviation *raw-clause* :: *'v clause twl-clause* \Rightarrow *'v clause* **where**
 raw-clause C \equiv *watched C* + *unwatched C*

datatype (*'a, 'b, 'c, 'd*) *twl-state* =
 TWL-State (*trail: 'a list*) (*init-clss: 'b*)
 (*learned-clss: 'b*) (*backtrack-lvl: 'c*)
 (*conflicting: 'd option*)

type-synonym (*'v, 'lvl, 'mark*) *twl-state-abs* =
 (*'v, 'lvl, 'mark*) *marked-lit, 'v clause twl-clause multiset, 'lvl, 'v clause*) *twl-state*

abbreviation *raw-init-clss* **where**
 raw-init-clss S \equiv *image-mset raw-clause (init-clss S)*

abbreviation *raw-learned-clss* **where**
 raw-learned-clss S \equiv *image-mset raw-clause (learned-clss S)*

abbreviation *clauses* **where**
 clauses S \equiv *init-clss S* + *learned-clss S*

abbreviation *raw-clauses* **where**
 raw-clauses S \equiv *image-mset raw-clause (clauses S)*

definition
 candidates-propagate :: (*'v, 'lvl, 'mark*) *twl-state-abs* \Rightarrow (*'v literal* \times *'v clause*) *set*
where
 candidates-propagate S =
 {(*L, raw-clause C*) | *L C.*
 C \in # *clauses S* \wedge *watched C* - *mset-set (uminus ' lits-of (trail S))* = {#*L*#} \wedge
 undefined-lit (trail S) L}

definition *candidates-conflict* :: (*'v, 'lvl, 'mark*) *twl-state-abs* \Rightarrow *'v clause set* **where**
 candidates-conflict S =
 {*raw-clause C* | *C. C* \in # *clauses S* \wedge *watched C* \subseteq # *mset-set (uminus ' lits-of (trail S))*}

primrec (*nonexhaustive*) *index* :: *'a list* \Rightarrow *'a* \Rightarrow *nat* **where**
 index (a # l) c = (*if a = c then 0 else 1 + 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 swapped with a watched literal L' such that $-L'$ is in the trail.

primrec *watched-decided-most-recently* :: $('v, 'vl, 'mark) \text{ marked-lit list} \Rightarrow 'v \text{ clause twl-clause} \Rightarrow \text{bool}$
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-twl-cl* :: $('v, 'vl, 'mark) \text{ marked-lit list} \Rightarrow 'v \text{ clause twl-clause} \Rightarrow \text{bool}$ **where**
wf-twl-cl $M \ (TWL\text{-Clause } W \ UW) \longleftrightarrow$
 $\text{distinct-mset } W \wedge \text{size } W \leq 2 \wedge (\text{size } W < 2 \longrightarrow \text{set-mset } UW \subseteq \text{set-mset } W) \wedge \\ (\forall L \in \# W. -L \in \text{lits-of } M \longrightarrow (\forall L' \in \# UW. L' \notin \# W \longrightarrow -L' \in \text{lits-of } M)) \wedge \\ \text{watched-decided-most-recently } M \ (TWL\text{-Clause } W \ UW)$

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*: $\text{distinct-mset } \{\#a, b\# \} \longleftrightarrow a \neq b$
unfolding *distinct-mset-def* **by** *auto*

lemma *wf-twl-cl*-*annotation-indepndant*:
assumes $M: \text{map lit-of } M = \text{map lit-of } M'$
shows $\text{wf-twl-cl } M \ (TWL\text{-Clause } W \ UW) \longleftrightarrow \text{wf-twl-cl } M' \ (TWL\text{-Clause } W \ UW)$
proof –
have $\text{lits-of } M = \text{lits-of } M'$
using *arg-cong* [*OF* M , *of set*] **by** (*simp add: lits-of-def*)
then show *?thesis*
by (*simp add: lits-of-def* M)
qed

lemma *wf-twl-cl*-*wf-twl-cl*-*tl*:
assumes $\text{wf: wf-twl-cl } M \ C$ **and** $n\text{-d: no-dup } M$
shows $\text{wf-twl-cl } (\text{tl } M) \ C$
proof (*cases* M)
case *Nil*
then show *?thesis* **using** *wf*
by (*cases* C) (*simp add: wf-twl-cl.simps* [*of tl* -])
next

```

case (Cons l M') note  $M = \text{this}(1)$ 
obtain  $W UW$  where  $C: C = \text{TWL-Clause } W UW$ 
  by (cases C)
{ fix  $L L'$ 
  assume
     $LW: L \in \# W$  and
     $LM: - L \in \text{lits-of } M'$  and
     $L'UW: L' \in \# UW$  and
     $\text{count } W L' = 0$ 
  then have
     $L'M: - L' \in \text{lits-of } M$ 
    using wf by (auto simp: C M)
  have watched-decided-most-recently M C
    using wf by (auto simp: C)
  then have
     $\text{index } (\text{map lit-of } M) (-L) \leq \text{index } (\text{map lit-of } M) (-L')$ 
    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  $- L' \in \text{lits-of } M'$ 
    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
     $L'M: - L' \in \text{lits-of } M'$  and
     $- L \in \text{lits-of } M'$  and
     $L \notin \# W$ 
  moreover
    have  $\text{lit-of } l \neq - L'$ 
    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  $\text{index } (\text{map lit-of } M') (- L') \leq \text{index } (\text{map lit-of } M') (- L)$ 
      by (fastforce simp: M C split: split-if-asm)
}
moreover have distinct-mset W and  $\text{size } W \leq 2$  and  $(\text{size } W < 2 \longrightarrow \text{set-mset } UW \subseteq \text{set-mset } W)$ 
  using wf by (auto simp: C M)
ultimately show ?thesis by (auto simp add: M C)
qed

```

definition *wf-tw-l-state* :: $(\text{'v}, \text{'vl}, \text{'mark}) \text{ tw-l-state-abs} \Rightarrow \text{bool}$ **where**
 $\text{wf-tw-l-state } S \longleftrightarrow (\forall C \in \# \text{ clauses } S. \text{ wf-tw-l-cl } (\text{trail } S) C) \wedge \text{no-dup } (\text{trail } S)$

lemma *wf-candidates-propagate-sound*:

assumes *wf*: *wf-tw-l-state S* **and**
cand: $(L, C) \in \text{candidates-propagate } S$
shows $\text{trail } S \models_{\text{as}} C \text{Not } (\text{mset-set } (\text{set-mset } C - \{L\})) \wedge \text{undefined-lit } (\text{trail } S) L$
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
  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 (cases Cw, blast)

have l-w: L ∈# W
  by (metis Multiset.diff-le-self cw(3) cw-eq mset-leD multi-member-last twl-clause.sel(1))

have wf-c: wf-twl-cls M Cw
  using wf (Cw ∈# N + U) 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' ∈ set-mset C - {L}. -L' ∈ 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 ⊨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-twll-state S and
  c-mem: C ∈# raw-clauses S and
  l-mem: L ∈# C and
  unsat: trail S ⊨as CNot (mset-set (set-mset C - {L})) and
  undef: undefined-lit (trail S) L
shows (L, C) ∈ candidates-propagate S
proof -
  def M ≡ trail S
  def N ≡ 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 (cases Cw, blast)

have wf-c: wf-twl-clss M Cw
  using wf cw(2) unfolding wf-twl-state-def by simp

have w-nw:
  distinct-mset W
  size W < 2  $\implies$  set-mset UW  $\subseteq$  set-mset W
   $\bigwedge L L'. L \in\# W \implies -L \in \text{ lits-of } M \implies L' \in\# UW \implies L' \notin\# W \implies -L' \in \text{ lits-of } M$ 
  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}  $\subseteq$  set-mset (W - mset-set (uminus ' lits-of M))
proof clarify
  have L  $\in\#$  W
  proof (cases W)
    case empty
    thus ?thesis
    using w-nw(2) cw(1) cw-eq l-mem by auto
  next
    case (add W' La)
    thus ?thesis
    proof (cases La = L)
      case True
      thus ?thesis
      using add by simp
    next
      case False

```

```

have  $-La \in \text{ lits-of } M$ 
  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 trail S  $\models_{as} \text{ CNot } C \wedge C \in \# \text{ image-mset raw-clause } (\text{clauses } S)$ 
proof
  def  $M \equiv \text{ trail } S$ 
  def  $N \equiv \text{ init-clss } S$ 
  def  $U \equiv \text{ learned-clss } S$ 

  note  $MNU\text{-defs } [simp] = M\text{-def } N\text{-def } U\text{-def}$ 

  obtain  $Cw$  where cw:
     $C = \text{ raw-clause } Cw$ 
     $Cw \in \# N + U$ 
     $\text{watched } Cw \subseteq \# \text{ mset-set } (\text{uminus } ' \text{ lits-of } (\text{trail } S))$ 
    using cand[unfolded candidates-conflict-def, simplified] by auto

  obtain  $W UW$  where cw-eq:  $Cw = \text{ TWL-Clause } W UW$ 
    by (cases 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  $\implies \text{ set-mset } UW \subseteq \text{ set-mset } W$ 
     $\bigwedge L L'. L \in \# W \implies -L \in \text{ lits-of } M \implies L' \in \# UW \implies L' \notin \# W \implies -L' \in \text{ lits-of } M$ 
    using wf-c unfolding cw-eq by auto

  have  $\forall L \in \# C. -L \in \text{ lits-of } M$ 
  proof (cases W =  $\{\#\}$ )
    case True

```

```

then have  $C = \{\#\}$ 
  using  $cw(1)$   $cw\text{-eq}$   $w\text{-nw}(2)$  by auto
then show ?thesis
  by simp
next
case False
then obtain  $La$  where  $la: La \in\# W$ 
  using  $multiset\text{-eq}\text{-iff}$  by force
show ?thesis
proof
  fix  $L$ 
  assume  $l: L \in\# C$ 
  show  $-L \in lits\text{-of } M$ 
  proof (cases  $L \in\# W$ )
    case True
    thus ?thesis
      using  $cw(3)$   $cw\text{-eq}$  by fastforce
  next
  case False
  thus ?thesis
    by (smt  $M\text{-def}$   $l$   $add\text{-diff}\text{-cancel}\text{-left}'$   $count\text{-diff}$   $cw(1)$   $cw(3)$   $la$   $cw\text{-eq}$ 
         $diff\text{-zero}$   $elem\text{-mset}\text{-set}$   $finite\text{-image}I$   $finite\text{-lits}\text{-of}\text{-def}$   $gr0I$   $imageE$   $mset\text{-le}D$ 
         $uminus\text{-of}\text{-uminus}\text{-id}$   $twl\text{-clause}.sel(1)$   $twl\text{-clause}.sel(2)$   $w\text{-nw}(3)$ )
  qed
qed
qed
then show  $trail\ S \models_{as} CNot\ C$ 
  unfolding  $CNot\text{-def}$   $true\text{-annots}\text{-def}$  by auto

show  $C \in\# image\text{-mset}\ raw\text{-clause}\ (clauses\ S)$ 
  using  $cw$  by auto
qed

lemma  $wf\text{-candidates}\text{-conflict}\text{-complete}$ :
  assumes  $wf: wf\text{-twl}\text{-state}\ S$  and
     $c\text{-mem}: C \in\# raw\text{-clauses}\ S$  and
     $unsat: trail\ S \models_{as} CNot\ C$ 
  shows  $C \in candidates\text{-conflict}\ S$ 
proof -
  def  $M \equiv trail\ S$ 
  def  $N \equiv init\text{-clss}\ S$ 
  def  $U \equiv learned\text{-clss}\ S$ 

  note  $MNU\text{-defs}\ [simp] = M\text{-def}\ N\text{-def}\ U\text{-def}$ 

  obtain  $Cw$  where  $cw: C = raw\text{-clause}\ Cw$   $Cw \in\# N + U$ 
    using  $c\text{-mem}$  by force

  obtain  $W\ UW$  where  $cw\text{-eq}: Cw = TWL\text{-Clause}\ W\ UW$ 
    by (cases  $Cw$ , blast)

  have  $wf\text{-c}: wf\text{-twl}\text{-cls}\ M\ Cw$ 
    using  $wf\ cw(2)$  unfolding  $wf\text{-twl}\text{-state}\text{-def}$  by simp

  have  $w\text{-nw}$ :

```



```

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  $\bigwedge L. L \in \# C \implies -L \in \text{ lits-of } M$ 
  unfolding M-def using unsat[unfolded CNot-def true-annots-def, simplified] by blast
then have set-mset C  $\subseteq$  uminus ' lits-of M
  by (metis imageI mem-set-mset-iff subsetI uminus-of-uminus-id)
then have set-mset W  $\subseteq$  uminus ' lits-of M
  using cw(1) cw-eq by auto
then have subset: W  $\subseteq \#$  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-abs. wf-twL-state S}
morphisms rough-state-of-twL twL-of-rough-state
proof -
  have TWL-State ([::('v, nat, 'v clause) marked-lits)
    {#} {#} 0 None  $\in$  {S::('v, nat, 'v clause) twl-state-abs. 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

$c-mem: C \in \# \text{ clauses-twl } S$ **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*

abbreviation *update-backtrack-lvl* **where**

$update-backtrack-lvl\ k\ S \equiv$

$TWL-State\ (trail\ S)\ (init-clss\ S)\ (learned-clss\ S)\ k\ (conflicting\ S)$

abbreviation *update-conflicting* **where**

$update-conflicting\ C\ S \equiv TWL-State\ (trail\ S)\ (init-clss\ S)\ (learned-clss\ S)\ (backtrack-lvl\ S)\ C$

21.3 Abstract 2-WL

definition *tl-trail* **where**

$tl-trail\ S =$

$TWL-State\ (tl\ (trail\ S))\ (init-clss\ S)\ (learned-clss\ S)\ (backtrack-lvl\ S)\ (conflicting\ S)$

locale *abstract-twl* =

fixes

$watch :: ('v, nat, 'v\ clause)\ twl-state-abs \Rightarrow 'v\ clause \Rightarrow 'v\ clause\ twl-clause$ **and**

$rewatch :: ('v, nat, 'v\ literal\ multiset)\ marked-lit \Rightarrow ('v, nat, 'v\ clause)\ twl-state-abs \Rightarrow 'v\ clause\ twl-clause \Rightarrow 'v\ clause\ twl-clause$ **and**

$linearize :: 'v\ clauses \Rightarrow 'v\ clause\ list$ **and**

$restart-learned :: ('v, nat, 'v\ clause)\ twl-state-abs \Rightarrow 'v\ clause\ twl-clause\ multiset$

assumes

$clause-watch: no-dup\ (trail\ S) \Longrightarrow raw-clause\ (watch\ S\ C) = C$ **and**

$wf-watch: no-dup\ (trail\ S) \Longrightarrow wf-twl-cls\ (trail\ S)\ (watch\ S\ C)$ **and**

$clause-rewatch: raw-clause\ (rewatch\ L\ S\ C') = raw-clause\ C'$ **and**

$wf-rewatch:$

$no-dup\ (trail\ S) \Longrightarrow undefined-lit\ (trail\ S)\ (lit-of\ L) \Longrightarrow wf-twl-cls\ (trail\ S)\ C' \Longrightarrow$

$wf-twl-cls\ (L\ \# \ trail\ S)\ (rewatch\ L\ S\ C')$

and

$linearize: mset\ (linearize\ N) = N$ **and**

$restart-learned: restart-learned\ S \subseteq \# \ learned-clss\ S$

begin

lemma *linearize-mempty[simp]*: $linearize\ \{\#\} = []$

using $linearize\ mset-zero-iff$ **by** *blast*

definition

$cons-trail :: ('v, nat, 'v\ clause)\ marked-lit \Rightarrow ('v, nat, 'v\ clause)\ twl-state-abs \Rightarrow ('v, nat, 'v\ clause)\ twl-state-abs$

where

$cons-trail\ L\ S =$

$TWL-State\ (L\ \# \ trail\ S)\ (image-mset\ (rewatch\ L\ S)\ (init-clss\ S))$

$(image-mset\ (rewatch\ L\ S)\ (learned-clss\ S))\ (backtrack-lvl\ S)\ (conflicting\ S)$

definition

$add-init-cls :: 'v\ clause \Rightarrow ('v, nat, 'v\ clause)\ twl-state-abs \Rightarrow$
 $('v, nat, 'v\ clause)\ twl-state-abs$

where

$add-init-cls\ C\ S =$
 $TWL-State\ (trail\ S)\ (\{\#watch\ S\ C\ \# \} + init-clss\ S)\ (learned-clss\ S)\ (backtrack-lvl\ S)$
 $(conflicting\ S)$

definition

$add-learned-cls :: 'v\ clause \Rightarrow ('v, nat, 'v\ clause)\ twl-state-abs \Rightarrow$
 $('v, nat, 'v\ clause)\ twl-state-abs$

where

$add-learned-cls\ C\ S =$
 $TWL-State\ (trail\ S)\ (init-clss\ S)\ (\{\#watch\ S\ C\ \# \} + learned-clss\ S)\ (backtrack-lvl\ S)$
 $(conflicting\ S)$

definition

$remove-cls :: 'v\ clause \Rightarrow ('v, nat, 'v\ clause)\ twl-state-abs \Rightarrow$
 $('v, nat, 'v\ clause)\ twl-state-abs$

where

$remove-cls\ C\ S =$
 $TWL-State\ (trail\ S)\ (filter-mset\ (\lambda D. raw-clause\ D \neq C)\ (init-clss\ S))$
 $(filter-mset\ (\lambda D. raw-clause\ D \neq C)\ (learned-clss\ S))\ (backtrack-lvl\ S)$
 $(conflicting\ S)$

definition $init-state :: 'v\ clauses \Rightarrow ('v, nat, 'v\ clause)\ twl-state-abs$ **where**

$init-state\ N = fold\ add-init-cls\ (linearize\ N)\ (TWL-State\ []\ \{\#\}\ \{\#\}\ 0\ None)$

lemma *unchanged-fold-add-init-cls:*

$trail\ (fold\ add-init-cls\ Cs\ (TWL-State\ M\ N\ U\ k\ C)) = M$
 $learned-clss\ (fold\ add-init-cls\ Cs\ (TWL-State\ M\ N\ U\ k\ C)) = U$
 $backtrack-lvl\ (fold\ add-init-cls\ Cs\ (TWL-State\ M\ N\ U\ k\ C)) = k$
 $conflicting\ (fold\ add-init-cls\ Cs\ (TWL-State\ M\ N\ U\ k\ C)) = C$
by $(induct\ Cs\ arbitrary: N)\ (auto\ simp: add-init-cls-def)$

lemma *unchanged-init-state[simp]:*

$trail\ (init-state\ N) = []$
 $learned-clss\ (init-state\ N) = \{\#\}$
 $backtrack-lvl\ (init-state\ N) = 0$
 $conflicting\ (init-state\ N) = None$
unfolding $init-state-def$ **by** $(rule\ unchanged-fold-add-init-cls)+$

lemma *clauses-init-fold-add-init:*

$no-dup\ M \implies$
 $image-mset\ raw-clause\ (init-clss\ (fold\ add-init-cls\ Cs\ (TWL-State\ M\ N\ U\ k\ C))) =$
 $mset\ Cs + image-mset\ raw-clause\ N$
by $(induct\ Cs\ arbitrary: N)\ (auto\ simp: add.assoc\ add-init-cls-def\ clause-watch)$

lemma *init-clss-init-state[simp]:* $image-mset\ raw-clause\ (init-clss\ (init-state\ N)) = N$

unfolding $init-state-def$ **by** $(simp\ add: clauses-init-fold-add-init\ linearize)$

definition *restart'* **where**

$restart'\ S = TWL-State\ []\ (init-clss\ S)\ (restart-learned\ S)\ 0\ None$

end

21.4 Instantiation of the previous locale

definition *watch-nat* :: (nat, nat, nat clause) twl-state-abs \Rightarrow nat clause \Rightarrow nat clause twl-clause **where**
watch-nat *S* *C* =
 (let
C' = remdups (sorted-list-of-set (set-mset *C*));
 negation-not-assigned = filter ($\lambda L. -L \notin \text{ lits-of } (\text{trail } S)$) *C'*;
 negation-assigned-sorted-by-trail = filter ($\lambda L. L \in \# C$) (map ($\lambda L. -\text{lit-of } L$) (trail *S*));
W = take 2 (negation-not-assigned @ negation-assigned-sorted-by-trail);
UW = sorted-list-of-multiset (*C* - mset *W*)
 in *TWL-Clause* (mset *W*) (mset *UW*))

lemma *list-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* ← *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* ← map ($\lambda L. - \text{lit-of } L$) *M*. *L* ∈ # *C*] = *l*
shows distinct *l*
unfolding *H*[symmetric]
apply (rule distinct-filter)
using *n-d* **by** (induction *M*) auto

lemma *watch-nat-lists-disjointD*:
assumes
 $l: [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)) . -L \notin \text{ lits-of } (\text{trail } S)] = l$ **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[symmetric] l'[symmetric] lits-of-def)

lemma *watch-nat-list-cases-witness*[consumes 2, case-names nil-nil nil-single nil-other single-nil single-other other]:
fixes
 C :: 'v literal multiset **and**
 C' :: 'v literal list **and**
 S :: (('v, 'b, 'c) marked-lit, 'd, 'e, 'f) twl-state
defines
 xs $\equiv [L \leftarrow \text{remdups } C'. -L \notin \text{ lits-of } (\text{trail } S)]$ **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**
 C': set *C'* = set-mset *C* **and**
 nil-nil: $xs = [] \implies ys = [] \implies P$ **and**
 nil-single:

$\bigwedge a. xs = [] \implies ys = [a] \implies a \in \# C \implies P$ **and**
nil-other: $\bigwedge a b ys'. xs = [] \implies ys = a \# b \# ys' \implies a \neq b \implies P$ **and**
single-nil: $\bigwedge a. xs = [a] \implies ys = [] \implies P$ **and**
single-other: $\bigwedge a b ys'. xs = [a] \implies ys = b \# ys' \implies a \neq b \implies P$ **and**
other: $\bigwedge a b xs'. xs = a \# b \# xs' \implies a \neq b \implies P$
shows P
proof –
note $xs\text{-def}[simp]$ **and** $ys\text{-def}[simp]$
have $dist: distinct [L \leftarrow remdups\ C' . - L \notin lits\text{-of}\ (trail\ S)]$
by *auto*
then have $H: \bigwedge a xs. [L \leftarrow remdups\ C' . - L \notin lits\text{-of}\ (trail\ S)]$
 $\neq a \# a \# xs$
by *force*
show *?thesis*
apply (*cases* $[L \leftarrow remdups\ C' . - L \notin lits\text{-of}\ (trail\ S)]$
rule: list-cases2;
cases $[L \leftarrow map\ (\lambda L. - lit\text{-of}\ L)\ (trail\ S) . L \in \# C]$ *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 C' xs-def ys-def* **apply** (*smt imageE image-eqI list.set-intros(1) lits-of-def*
mem-Collect-eq set-filter set-map uminus-of-uminus-id)
using *single-other C' unfolding xs-def ys-def* **apply** (*smt imageE image-eqI list.set-intros(1)*
lits-of-def mem-Collect-eq set-filter set-map uminus-of-uminus-id)
using *other xs-def ys-def* **by** (*metis H*)
qed

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**
 $S :: (('v, 'b, 'c)\ marked\ lit, 'd, 'e, 'f)\ twl\ state$
defines
 $xs \equiv [L \leftarrow remdups\ (sorted\ list\ of\ set\ (set\ mset\ C)) . - L \notin lits\text{-of}\ (trail\ S)]$ **and**
 $ys \equiv [L \leftarrow map\ (\lambda L. - lit\text{-of}\ L)\ (trail\ S) . L \in \# C]$
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
using *watch-nat-list-cases-witness[OF n-d, of sorted-list-of-set (set-mset C) C P]*
nil-nil nil-single nil-other single-nil single-other other
unfolding $xs\text{-def}[symmetric]$ $ys\text{-def}[symmetric]$ **by** *auto*

lemma *watch-nat-lists-set-union-witness*:
fixes
 $C :: 'v\ literal\ multiset$ **and**

$C' :: 'v \text{ literal list and}$
 $S :: (('v, 'b, 'c) \text{ marked-lit, 'd, 'e, 'f}) \text{ twl-state}$
defines
 $xs \equiv [L \leftarrow \text{remdups } C'. - L \notin \text{lits-of } (\text{trail } S)] \text{ and}$
 $ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) . L \in \# C]$
assumes $n\text{-d: no-dup } (\text{trail } S) \text{ and } C': \text{set } C' = \text{set-mset } C$
shows $\text{set-mset } C = \text{set } xs \cup \text{set } ys$
using $n\text{-d } C' \text{ uminus-lit-swap unfolding } xs\text{-def } ys\text{-def by } (\text{auto simp: lits-of-def})$

lemma *watch-nat-lists-set-union:*

fixes
 $C :: 'v::\text{linorder literal multiset and}$
 $S :: (('v, 'b, 'c) \text{ marked-lit, 'd, 'e, 'f}) \text{ twl-state}$
defines
 $xs \equiv [L \leftarrow \text{remdups } (\text{sorted-list-of-set } (\text{set-mset } C)). - L \notin \text{lits-of } (\text{trail } S)] \text{ and}$
 $ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{trail } S) . L \in \# C]$
assumes $n\text{-d: no-dup } (\text{trail } S)$
shows $\text{set-mset } C = \text{set } xs \cup \text{set } ys$
using $\text{watch-nat-lists-set-union-witness}[\text{of } S (\text{sorted-list-of-set } (\text{set-mset } C)) C, \text{OF } n\text{-d}]$
 $\text{sorted-list-of-set } xs\text{-def } ys\text{-def by blast}$

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 *set-mset-is-single-in-mset-is-single:*

$\text{set-mset } C = \{a\} \implies x \in \# C \implies x = a$
by *fastforce*

lemma *index-uminus-index-map-uminus:*

$-a \in \text{set } L \implies \text{index } L (-a) = \text{index } (\text{map } \text{uminus } L) (a::'a \text{ literal})$
by (*induction L*) *auto*

lemma *index-filter:*

$a \in \text{set } L \implies b \in \text{set } L \implies P a \implies P b \implies$
 $\text{index } L a \leq \text{index } L b \longleftrightarrow \text{index } (\text{filter } P L) a \leq \text{index } (\text{filter } P L) b$
by (*induction L*) *auto*

lemma *wf-watch-witness:*

fixes $C :: 'a \text{ literal multiset and } C':: 'a \text{ literal list and}$
 $S :: (('a, 'b, 'c) \text{ marked-lit, 'd, 'e, 'f}) \text{ twl-state}$
defines
 $\text{ass: negation-not-assigned} \equiv \text{filter } (\lambda L. -L \notin \text{lits-of } (\text{trail } S)) (\text{remdups } C') \text{ and}$
 $\text{tr: negation-assigned-sorted-by-trail} \equiv \text{filter } (\lambda L. L \in \# C) (\text{map } (\lambda L. -\text{lit-of } L) (\text{trail } S))$
defines

```

    W: W  $\equiv$  take 2 (negation-not-assigned @ negation-assigned-sorted-by-trail)
  assumes
    n-d[simp]: no-dup (trail S) and
    C': set C' = set-mset C
  shows wf-twl-cls (trail S) (TWL-Clause (mset W) (C - mset W))
  unfolding wf-twl-cls.simps
proof (intro conjI, goal-cases)
  case 1
  then show ?case using n-d C' W unfolding ass tr
    by (cases rule: watch-nat-list-cases-witness[of S C' C])
      (auto dest: filter-in-list-prop-verifiedD
        simp: distinct-mset-add-single)
  next
  case 2
  then show ?case unfolding W by simp
  next
  case 3
  then show ?case using n-d C'
  proof (cases rule: watch-nat-list-cases-witness[of S C' C])
    case nil-nil
    then have set-mset C = set []  $\cup$  set []
      using C' watch-nat-lists-set-union-witness n-d by metis
    then show ?thesis
      by simp
  next
  case (nil-single a)
  then show ?thesis
    using watch-nat-lists-set-union-witness[of S C' C] C' 3
    by (auto dest!: arg-cong[of - [] set] simp: W ass tr)
  next
  case nil-other
  then show ?thesis
    using 3 by (auto dest!: arg-cong[of - [] set] simp: W ass tr)
  next
  case single-nil
  show ?thesis
    using watch-nat-lists-set-union-witness[of S C' C] C' 3 mset-leD
    by (auto simp: W ass tr single-nil)
  next
  case single-other
  then show ?thesis
    using 3 by (auto dest!: arg-cong[of - [] set] simp: W ass tr)
  next
  case other
  then show ?thesis
    using 3 by (auto dest!: arg-cong[of - [] set] simp: W ass tr)
  qed
next
case 4 note -[simp] = this
{
  fix a :: 'a literal and ys' :: 'a literal list and L :: 'a literal and
    L' :: 'a literal
  assume a1: [L $\leftarrow$ remdups C'. - L  $\notin$  lits-of (trail S)] = [a]
  assume a2: set-mset C = insert L (insert a (set ys'))
  assume a3: L'  $\in$  # C

```

```

assume  $a_4$ :  $a \neq L'$ 
have  $set (L \# a \# ys') = set-mset C$ 
  using  $a_2$  by auto
then have  $L' \notin set [l \leftarrow remdups C'. - l \notin lits-of (trail S)]$ 
  using  $a_4 a_1$  by (metis list.set(1) list.set(2) singleton-iff)
then have  $- L' \in lits-of (trail S)$ 
  using  $a_3 C'$  by simp
} note  $H = this$ 
show ?case
  using  $n-d C'$  apply (cases rule: watch-nat-list-cases-witness[of S C' C])
  apply (auto dest: filter-in-list-prop-verifiedD
    simp: W ass tr lits-of-def C' filter-empty-conv)[4]
  using watch-nat-lists-set-union-witness[of S C' C] C'
  by (auto dest: filter-in-list-prop-verifiedD H simp: W ass tr)
next
case 5
from  $n-d C'$  show ?case
  proof (cases rule: watch-nat-list-cases-witness[of S C' C])
    case nil-nil
    then show ?thesis by (auto simp: W ass tr)
  next
    case nil-single
    then show ?thesis
      using watch-nat-lists-set-union-witness[of S C' C] C' by (auto simp: W ass tr)
  next
    case nil-other
    then show ?thesis
      unfolding watched-decided-most-recently.simps Ball-mset-def
      apply (intro allI impI)
      apply (subst index-uminus-index-map-uminus,
        simp add: index-uminus-index-map-uminus lits-of-def o-def)
      apply (subst index-uminus-index-map-uminus,
        simp add: index-uminus-index-map-uminus lits-of-def o-def)

      apply (subst index-filter[of - -  $\lambda L. L \in \# C$ ])
      by (auto dest: filter-in-list-prop-verifiedD
        simp: uminus-lit-swap lits-of-def o-def W ass tr)
  next
    case single-nil
    then show ?thesis
      using watch-nat-lists-set-union-witness[of S C' C] C' by (auto simp: W ass tr)
  next
    case single-other
    then show ?thesis
      unfolding watched-decided-most-recently.simps Ball-mset-def
      apply (clarify)
      apply (subst index-uminus-index-map-uminus,
        simp add: index-uminus-index-map-uminus lits-of-def o-def)
      apply (subst index-uminus-index-map-uminus,
        simp add: index-uminus-index-map-uminus lits-of-def o-def)

      apply (subst index-filter[of - -  $\lambda L. L \in \# C$ ])
      by (auto dest: filter-in-list-prop-verifiedD
        simp: W ass tr uminus-lit-swap lits-of-def o-def)
  next

```



```

case other
then show ?thesis
  unfolding watched-decided-most-recently.simps
  apply clarify
  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
    W ass tr)
qed
qed

lemma wf-watch-nat: no-dup (trail S) ⇒ wf-twl-cls (trail S) (watch-nat S C)
  using wf-watch-witness[of S sorted-list-of-set (set-mset C) C]
  by (metis List.finite-set mset-sorted-list-of-multiset set-sorted-list-of-multiset
    sorted-list-of-set watch-nat-def)

definition
  rewatch-nat ::
    (nat, nat, nat literal multiset) marked-lit ⇒ (nat, nat, nat clause) twl-state-abs ⇒
    nat clause twl-clause ⇒ nat clause twl-clause
where
  rewatch-nat L S C =
    (if - lit-of L ∈# watched C then
      case filter (λL'. L' ∉# watched C ∧ - L' ∉ lits-of (L # trail S))
        (sorted-list-of-multiset (unwatched C)) of
         $\square \Rightarrow C$ 
      | L' # - ⇒
        TWL-Clause (watched C - {#- lit-of L#} + {#L'#}) (unwatched C - {#L'#} + {#- lit-of
L#})
      else
        C)

lemma clause-rewatch-witness:
  fixes UW :: 'a literal list and
    S :: (('a, 'b, 'c) marked-lit, 'd, 'e, 'f) twl-state and
    L :: ('a, 'b, 'c) marked-lit and C :: 'a literal multiset twl-clause
  defines C' ≡ (if - lit-of L ∈# watched C then
    case filter (λL'. L' ∉# watched C ∧ - L' ∉ lits-of (L # trail S)) UW of
     $\square \Rightarrow C$ 
    | L' # - ⇒
    TWL-Clause (watched C - {#- lit-of L#} + {#L'#}) (unwatched C - {#L'#} + {#- lit-of
L#})
    else
    C)
  assumes
    UW: set UW = set-mset (unwatched C)
  shows raw-clause C' = raw-clause C
  using UW unfolding C'-def by (auto simp: subset-mset.add-diff-assoc2 multiset-eq-iff
    split: list.split dest: filter-in-list-prop-verifiedD)

```

lemma *clause-rewatch-nat*: *raw-clause* (*rewatch-nat* *L S C*) = *raw-clause* *C*
using *clause-rewatch-witness*[*of sorted-list-of-multiset* (*unwatched C*) *C - S*]
by (*auto simp*: *rewatch-nat-def Let-def split*: *list.split split-if-asm*)

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-empty*:
 $a - \{\#b\} = \{\#\} \longleftrightarrow a = \{\#b\} \vee a = \{\#\}$
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* ≤ 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 le-iff-add size-1-singleton-mset
size-eq-0-iff-empty size-mset-2)

lemma *filter-sorted-list-of-multiset-eqD*:
assumes $[x \leftarrow \text{sorted-list-of-multiset } A. p \ x] = x \# xs$ (**is** *?comp = -*)
shows $x \in\# A$
proof -
have $x \in \text{set } ?comp$
using *assms* **by** *simp*
then have $x \in \text{set } (\text{sorted-list-of-multiset } A)$
by *simp*
then show $x \in\# A$
by *simp*
qed

lemma *clause-rewatch-witness'*:
fixes *UWC* :: '*a literal list* **and**
S :: (('a, 'b, 'c) *marked-lit*, 'd, 'e, 'f) *twl-state* **and**
L :: ('a, 'b, 'c) *marked-lit* **and** *C* :: '*a literal multiset twl-clause*
defines $C' \equiv (\text{if } - \text{ lit-of } L \in\# \text{ watched } C \text{ then}$
 $\text{case filter } (\lambda L'. L' \notin\# \text{ watched } C \wedge - L' \notin \text{ lits-of } (L \# \text{ trail } S)) \text{ UWC of}$
 $\quad [] \Rightarrow C$
 $\quad | L' \# - \Rightarrow$
 $\quad \text{TWL-Clause } (\text{watched } C - \{\# - \text{ lit-of } L\} + \{\#L'\}) (\text{unwatched } C - \{\#L'\} + \{\# - \text{ lit-of}$
 $L\# \})$
 $\quad \text{else}$
 $\quad C)$
assumes
UWC: *set UWC* = *set-mset* (*unwatched C*) **and**
wf: *wf-tw-cl*s (*trail S*) *C* **and**
n-d: *no-dup* (*trail S*) **and**
undef: *undefined-lit* (*trail S*) (*lit-of L*)
shows *wf-tw-cl*s (*L* # *trail S*) *C'*

```

proof (cases – lit-of  $L \in \#$  watched  $C$ )
  case False
  then have wf-twl-cls ( $L \#$  trail  $S$ )  $C$ 
    apply (cases  $C$ )
    using wf n-d undef apply (clarify)
    unfolding wf-twl-cls.simps
    apply (intro conjI)
      apply blast
      apply blast
      apply blast
    apply (smt ball-mset-cong bspec-mset insert-iff lits-of-cons nat-neq-iff twl-clause.sel(1)
      uminus-of-uminus-id)
    apply (auto simp: Marked-Propagated-in-iff-in-lits-of)
  done
  then show ?thesis
    using False C'-def by simp
next
  case falsified: True

  let ?unwatched-nonfalsified =
    [ $L' \leftarrow UWC. L' \notin \#$  watched  $C \wedge - L' \notin$  lits-of ( $L \#$  trail  $S$ )]
  obtain  $W UW$  where  $C: C = \textit{TWL-Clause } W UW$ 
    by (cases  $C$ )

  show ?thesis
  proof (cases ?unwatched-nonfalsified)
    case Nil
    show ?thesis
      using falsified Nil
      apply (simp only: wf-twl-cls.simps if-True list.cases C C'-def)
      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
        have  $\bigwedge p l. \textit{filter } p \ UWC \neq [] \vee l \notin \textit{set-mset } UW \vee \neg p \ l$ 
          using  $UWC$  unfolding  $C$  by (metis (no-types) filter-empty-conv twl-clause.sel(2))
        then show ?case
          using 4(2) unfolding Ball-mset-def by (metis (lifting) mem-set-mset-iff twl-clause.sel(1))
      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'-def
  using falsified Cons
  apply (simp only: wf-twl-clcls.simps if-True list.cases C)
  apply (intro conjI)
  proof goal-cases
    case 1
    have distinct-mset (watched (TWL-Clause W UW))
      using wf unfolding C by auto
    moreover have L' ∉ # watched (TWL-Clause W UW) - {#- lit-of L#}
      using 1(2) not-gr0 by (fastforce dest: filter-in-list-prop-verifiedD)
    ultimately show ?case
      by (auto simp: distinct-mset-single-add)
  next
    case 2
    then show ?case using wf C by (metis insert-DiffM2 size-single size-union twl-clause.sel(1)
      wf-twl-clcls.simps)
  next
    case 3
    then show ?case
      using wf C UWC by (force simp: mset-minus-single-eq-mempty dest: subset-singletonD)
  next
    case 4
    have H: ∀ L ∈ # W. - L ∈ lits-of (trail S) →
      (∀ L' ∈ # UW. count W L' = 0 → - L' ∈ lits-of (trail S))
      using wf by (auto simp: C)
    have W: size W ≤ 2 and W-UW: size W < 2 → set-mset UW ⊆ 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]
      apply (force dest: filter-in-list-prop-verifiedD)
      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)
      apply (force dest: filter-in-list-prop-verifiedD)
      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: ∀ x. x ∈ # W → - x ∈ lits-of (trail S) → (∀ x. x ∈ # UW → count W x = 0
      → - x ∈ lits-of (trail S))
      using wf by (auto simp: C)
    show ?case
      unfolding C watched-decided-most-recently.simps Ball-mset-def

```

```

proof (intro allI impI conjI, goal-cases)
  case (1  $xW$   $x$ )
  show ?case
    proof (cases - lit-of L = xW)
      case True
      then show ?thesis
        by (cases xW = x) (auto simp: uminus-lit-swap)
    next
      case False note  $LxW = \text{this}$ 
      have  $f9: L' \in \text{set } [l \leftarrow UWC . l \notin \# \text{watched } (TWL\text{-Clause } W \ UW)$ 
         $\wedge - l \notin \text{lits-of } (L \# \text{trail } S)]$ 
      using 1(2) 5 by auto
      moreover then have  $f11: - xW \in \text{lits-of } (\text{trail } S)$ 
      using 1(3)  $LxW$  unfolding lits-of-cons by (metis (no-types) insert-iff
        uminus-of-uminus-id)
      moreover then have  $xW \notin \# W$ 
      using  $f9$  1(2)  $H$  by (auto simp: C UWC)
      ultimately have False
      using 1 by auto
      then show ?thesis
        by fast
    qed
  qed
qed
qed
qed

```

```

lemma wf-rewatch-nat':
  assumes
     $wf: wf\text{-twl-cl}\ (trail\ S)\ C$  and
     $n\text{-d}: no\text{-dup}\ (trail\ S)$  and
     $undef: undefined\text{-lit}\ (trail\ S)\ (lit\text{-of}\ L)$ 
  shows  $wf\text{-twl-cl}\ (L \# trail\ S)\ (rewatch\text{-nat}\ L\ S\ C)$ 
  using clause-rewatch-witness'[of sorted-list-of-multiset (unwatched C) C S L]
  assms by (auto simp: rewatch-nat-def)

```

```

interpretation twl: abstract-twl watch-nat rewatch-nat sorted-list-of-multiset learned-clss
  apply unfold-locales
  apply (rule clause-watch-nat; simp)
  apply (rule wf-watch-nat; simp)
  apply (rule clause-rewatch-nat)
  apply (rule wf-rewatch-nat'; simp)
  apply (rule mset-sorted-list-of-multiset)
  apply (rule subset-mset.order-refl)
done

```

21.5 Interpretation for $cdcl_W.cdcl_W$

```

context abstract-twl
begin

```

21.5.1 Direct Interpretation

```

interpretation rough-cdcl: stateW trail raw-init-clss raw-learned-clss backtrack-lvl conflicting

```

```

cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
update-conflicting init-state restart'
apply unfold-locales
apply (simp-all add: add-init-cls-def add-learned-cls-def clause-rewatch clause-watch
  cons-trail-def remove-cls-def restart'-def tl-trail-def)
apply (rule image-mset-subseteq-mono[OF restart-learned])
done

```

interpretation *rough-cdcl*: $cdcl_W$ trail raw-init-clss raw-learned-clss backtrack-lvl conflicting
 cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
 update-conflicting init-state restart'
by unfold-locales

21.5.2 Opaque Type with Invariant

declare *rough-cdcl.state-simp*[simp del]

definition *cons-trail-tw*l :: ('v, nat, 'v literal multiset) marked-lit \Rightarrow 'v wf-tw \Rightarrow 'v wf-tw \Rightarrow
where
*cons-trail-tw*l L S \equiv twl-of-rough-state (cons-trail L (rough-state-of-tw \Rightarrow l S))

lemma *wf-tw \Rightarrow l-state-cons-trail*:
 undefined-lit (trail S) (lit-of L) \implies wf-tw \Rightarrow l-state S \implies wf-tw \Rightarrow l-state (cons-trail L S)
unfolding wf-tw \Rightarrow l-state-def **by** (auto simp: cons-trail-def wf-rewatch defined-lit-map)

lemma *rough-state-of-tw \Rightarrow l-cons-trail*:
 undefined-lit (trail-tw \Rightarrow l S) (lit-of L) \implies
 rough-state-of-tw \Rightarrow l (cons-trail-tw \Rightarrow l L S) = cons-trail L (rough-state-of-tw \Rightarrow l S)
using rough-state-of-tw \Rightarrow l twl-of-rough-state-inverse wf-tw \Rightarrow l-state-cons-trail
unfolding cons-trail-tw \Rightarrow l-def **by** blast

abbreviation *add-init-cls-tw*l **where**
*add-init-cls-tw*l C S \equiv twl-of-rough-state (add-init-cls C (rough-state-of-tw \Rightarrow l S))

lemma *wf-tw \Rightarrow l-add-init-cls*: wf-tw \Rightarrow l-state S \implies wf-tw \Rightarrow l-state (add-init-cls L S)
unfolding wf-tw \Rightarrow l-state-def **by** (auto simp: wf-watch add-init-cls-def split: split-if-asm)

lemma *rough-state-of-tw \Rightarrow l-add-init-cls*:
 rough-state-of-tw \Rightarrow l (add-init-cls-tw \Rightarrow l L S) = add-init-cls L (rough-state-of-tw \Rightarrow l S)
using rough-state-of-tw \Rightarrow l twl-of-rough-state-inverse wf-tw \Rightarrow l-add-init-cls **by** blast

abbreviation *add-learned-cls-tw*l **where**
*add-learned-cls-tw*l C S \equiv twl-of-rough-state (add-learned-cls C (rough-state-of-tw \Rightarrow l S))

lemma *wf-tw \Rightarrow l-add-learned-cls*: wf-tw \Rightarrow l-state S \implies wf-tw \Rightarrow l-state (add-learned-cls L S)
unfolding wf-tw \Rightarrow l-state-def **by** (auto simp: wf-watch add-learned-cls-def split: split-if-asm)

lemma *rough-state-of-tw \Rightarrow l-add-learned-cls*:
 rough-state-of-tw \Rightarrow l (add-learned-cls-tw \Rightarrow l L S) = add-learned-cls L (rough-state-of-tw \Rightarrow l S)
using rough-state-of-tw \Rightarrow l twl-of-rough-state-inverse wf-tw \Rightarrow l-add-learned-cls **by** blast

abbreviation *remove-cls-tw*l **where**
*remove-cls-tw*l C S \equiv twl-of-rough-state (remove-cls C (rough-state-of-tw \Rightarrow l S))

lemma *wf-tw \Rightarrow l-remove-cls*: wf-tw \Rightarrow l-state S \implies wf-tw \Rightarrow l-state (remove-cls L S)
unfolding wf-tw \Rightarrow l-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 None)
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*

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*

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-cls_{NOT}*)

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: rough-state-of-twl-tl-trail*)

using *rough-cdcl.clauses-add-cls_{NOT} 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-tw
init-state-tw
restart-tw
apply *unfold-locales*
by (*simp-all* *add*: *rough-state-of-tw-cons-trail* *rough-state-of-tw-tl-trail*
rough-state-of-tw-add-init-cl *rough-state-of-tw-add-learned-cl* *rough-state-of-tw-remove-cl*
rough-state-of-tw-update-backtrack-lvl *rough-state-of-tw-update-conflicting*
rough-state-of-tw-init-state *rough-state-of-tw-restart-tw*
rough-cdcl.learned-clss-restart-state)

interpretation *cdcl_W-tw*: *cdcl_W*

trail-tw
init-clss-tw
learned-clss-tw
backtrack-lvl-tw
conflicting-tw
cons-trail-tw
tl-trail-tw
add-init-cl-tw
add-learned-cl-tw
remove-cl-tw
update-backtrack-lvl-tw
update-conflicting-tw
init-state-tw
restart-tw
by *unfold-locales*

sublocale *cdcl_W*

trail-tw
init-clss-tw
learned-clss-tw
backtrack-lvl-tw
conflicting-tw
cons-trail-tw
tl-trail-tw
add-init-cl-tw
add-learned-cl-tw
remove-cl-tw
update-backtrack-lvl-tw
update-conflicting-tw
init-state-tw
restart-tw
by (*rule* *cdcl_W-tw.cdcl_W-axioms*)

abbreviation *state-eq-tw* (**infix** \sim_{TWL} 51) **where**

state-eq-tw S $S' \equiv$ *rough-cdcl.state-eq* (*rough-state-of-tw* S) (*rough-state-of-tw* S')

notation *cdcl_W-tw.state-eq* (**infix** \sim 51)

declare *cdcl_W-tw.state-simp*[*simp del*]
*cdcl_W-tw.NOT.state-simp*_{NOT}[*simp del*]

To avoid ambiguities:

no-notation *state-eq-tw* (**infix** \sim 51)

definition *propagate-tw* **where**

propagate-tw S $S' \longleftrightarrow$

$(\exists L C. (L, C) \in \text{candidates-propagate-twl } S$
 $\wedge S' \sim \text{cons-trail-twl } (\text{Propagated } L \ C) \ S$
 $\wedge \text{conflicting-twl } S = \text{None})$

lemma *propagate-twl-iff-propagate:*

assumes *inv*: $\text{cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv } S$

shows $\text{cdcl}_W\text{-twl.propagate } S \ T \longleftrightarrow \text{propagate-twl } S \ T$ (**is** $?P \longleftrightarrow ?T$)

proof

assume $?P$

then obtain $C \ L$ **where**

conflicting (*rough-state-of-twl* S) = *None* **and**

CL-Clauses: $C + \{\#L\# \} \in \# \text{cdcl}_W\text{-twl.clauses } S$ **and**

tr-CNot: $\text{trail-twl } S \models_{\text{as}} \text{CNot } C$ **and**

undef-lot: *undefined-lit* (*trail-twl* S) L **and**

$T \sim \text{cons-trail-twl } (\text{Propagated } L \ (C + \{\#L\# \})) \ S$

unfolding $\text{cdcl}_W\text{-twl.propagate.simps}$ **by** *blast*

have *distinct-mset* ($C + \{\#L\# \}$)

using *inv* *CL-Clauses* **unfolding** $\text{cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv-def}$

$\text{cdcl}_W\text{-twl.distinct-cdcl}_W\text{-state-def}$ $\text{cdcl}_W\text{-twl.clauses-def}$ *distinct-mset-set-def*

by (*metis* (*no-types*, *lifting*) *add-gr-0* *mem-set-mset-iff* *plus-multiset.rep-eq*)

then have *C-L-L*: *mset-set* (*set-mset* ($C + \{\#L\# \}$) - $\{L\}$) = C

by (*metis* *Un-insert-right* *add-diff-cancel-left'* *add-diff-cancel-right'*

distinct-mset-set-mset-ident *finite-set-mset* *insert-absorb2* *mset-set.insert-remove*

set-mset-single *set-mset-union*)

have $(L, C + \{\#L\# \}) \in \text{candidates-propagate-twl } S$

apply (*rule* *wf-candidates-propagate-complete*)

using *rough-state-of-twl* **apply** *auto*[]

using *CL-Clauses* **unfolding** $\text{cdcl}_W\text{-twl.clauses-def}$ **apply** *auto*[]

apply *simp*

using *C-L-L* *tr-CNot* **apply** *simp*

using *undef-lot* **apply** *blast*

done

show $?T$ **unfolding** *propagate-twl-def*

apply (*rule* *exI*[*of* - L], *rule* *exI*[*of* - $C + \{\#L\# \}$])

apply (*auto* *simp*: $\langle (L, C + \{\#L\# \}) \in \text{candidates-propagate-twl } S \rangle$

$\langle \text{conflicting } (\text{rough-state-of-twl } S) = \text{None} \rangle$)

using $\langle T \sim \text{cons-trail-twl } (\text{Propagated } L \ (C + \{\#L\# \})) \ S \rangle$ $\text{cdcl}_W\text{-twl.state-eq-backtrack-lvl}$

$\text{cdcl}_W\text{-twl.state-eq-conflicting}$ $\text{cdcl}_W\text{-twl.state-eq-init-clss}$

$\text{cdcl}_W\text{-twl.state-eq-learned-clss}$ $\text{cdcl}_W\text{-twl.state-eq-trail}$ *rough-cdcl.state-eq-def* **by** *blast*

next

assume $?T$

then obtain $L \ C$ **where**

LC: $(L, C) \in \text{candidates-propagate-twl } S$ **and**

T: $T \sim \text{cons-trail-twl } (\text{Propagated } L \ C) \ S$ **and**

confl: *conflicting* (*rough-state-of-twl* S) = *None*

unfolding *propagate-twl-def* **by** *auto*

have [*simp*]: $C - \{\#L\# \} + \{\#L\# \} = C$

using *LC* **unfolding** *candidates-propagate-def*

by *clarify* (*metis* *add commute* *add-diff-cancel-right'* *count-diff* *insert-DiffM*

multi-member-last *not-gr0* *zero-diff*)

have $C \in \# \text{raw-clauses-twl } S$

using *LC* **unfolding** *candidates-propagate-def* *rough-cdcl.clauses-def* **by** *auto*

then have *distinct-mset* C

using *inv* **unfolding** $\text{cdcl}_W\text{-twl.cdcl}_W\text{-all-struct-inv-def}$ $\text{cdcl}_W\text{-twl.distinct-cdcl}_W\text{-state-def}$

$\text{cdcl}_W\text{-twl.clauses-def}$ *distinct-mset-set-def* *rough-cdcl.clauses-def* **by** *auto*

then have $C\text{-}L\text{-}L$: $\text{mset-set } (\text{set-mset } C - \{L\}) = C - \{\#L\# \}$
by ($\text{metis } \langle C - \{\#L\# \} + \{\#L\# \} = C \rangle$ add-left-imp-eq $\text{diff-single-trivial}$
 $\text{distinct-mset-set-mset-ident}$ finite-set-mset mem-set-mset-iff mset-set.remove
 $\text{multi-self-add-other-not-self}$ union-commute)

show $?P$
apply ($\text{rule } \text{cdcl}_W\text{-twl.propagate.intros}[of - \text{trail-twl } S \text{ init-clss-twl } S$
 $\text{learned-clss-twl } S \text{ backtrack-lvl-twl } S \text{ } C - \{\#L\# \} \text{ } L])$
using confl **apply** $\text{auto}[]$
using LC **unfolding** $\text{candidates-propagate-def}$ **apply** ($\text{auto simp: } \text{cdcl}_W\text{-twl.clauses-def}[]$)
using $\text{wf-candidates-propagate-sound}[OF - LC]$ $\text{rough-state-of-twl}$ **apply** ($\text{simp add: } C\text{-}L\text{-}L$)
using $\text{wf-candidates-propagate-sound}[OF - LC]$ $\text{rough-state-of-twl}$ **apply** simp
using T **unfolding** $\text{cdcl}_W\text{-twl.state-eq-def}$ $\text{rough-cdcl.state-eq-def}$ **by** auto

qed
no-notation $CDCL\text{-Two-Watched-Literals.twl.state-eq-twl}$ (**infix** \sim_{TWL} 51)

definition conflict-twl **where**
 $\text{conflict-twl } S \text{ } S' \longleftrightarrow$
 $(\exists C. C \in \text{candidates-conflict-twl } S$
 $\wedge S' \sim \text{update-conflicting-twl } (\text{Some } C) \text{ } S$
 $\wedge \text{conflicting-twl } S = \text{None})$

lemma $\text{conflict-twl-iff-conflict}$:
shows $\text{cdcl}_W\text{-twl.conflict } S \text{ } T \longleftrightarrow \text{conflict-twl } S \text{ } T$ (**is** $?C \longleftrightarrow ?T$)

proof
assume $?C$
then obtain $M \text{ } N \text{ } U \text{ } k \text{ } C$ **where**
 S : $\text{rough-cdcl.state } (\text{rough-state-of-twl } S) = (M, N, U, k, \text{None})$ **and**
 C : $C \in \# \text{cdcl}_W\text{-twl.clauses } S$ **and**
 $M\text{-}C$: $M \models_{as} C\text{Not } C$ **and**
 T : $T \sim \text{update-conflicting-twl } (\text{Some } C) \text{ } S$
by auto
have $C \in \text{candidates-conflict-twl } S$
apply ($\text{rule } \text{wf-candidates-conflict-complete}$)
apply simp
using C **apply** ($\text{auto simp: } \text{cdcl}_W\text{-twl.clauses-def}[]$)
using $M\text{-}C \text{ } S$ **by** auto
moreover have $T \sim \text{twl-of-rough-state } (\text{update-conflicting } (\text{Some } C) \text{ } (\text{rough-state-of-twl } S))$
using T **unfolding** $\text{rough-cdcl.state-eq-def}$ $\text{cdcl}_W\text{-twl.state-eq-def}$ **by** auto
ultimately show $?T$
using S **unfolding** conflict-twl-def **by** auto

next
assume $?T$
then obtain C **where**
 C : $C \in \text{candidates-conflict-twl } S$ **and**
 T : $T \sim \text{update-conflicting-twl } (\text{Some } C) \text{ } S$ **and**
 confl : $\text{conflicting-twl } S = \text{None}$
unfolding conflict-twl-def **by** auto
have $C \in \# \text{cdcl}_W\text{-twl.clauses } S$
using C **unfolding** $\text{candidates-conflict-def}$ $\text{cdcl}_W\text{-twl.clauses-def}$ **by** auto
moreover have $\text{trail-twl } S \models_{as} C\text{Not } C$
using $\text{wf-candidates-conflict-sound}[OF - C]$ **by** auto
ultimately show $?C$ **apply** –
apply ($\text{rule } \text{cdcl}_W\text{-twl.conflict.conflict-rule}[of - - - - C]$)
using $\text{confl } T$ **unfolding** $\text{rough-cdcl.state-eq-def}$ $\text{cdcl}_W\text{-twl.state-eq-def}$ **by** auto

qed

inductive $cdcl_W\text{-}twl :: 'v \text{ wf-}twl \Rightarrow 'v \text{ wf-}twl \Rightarrow \text{bool}$ **for** $S :: 'v \text{ wf-}twl$ **where**
propagate: $\text{propagate-}twl\ S\ S' \Longrightarrow cdcl_W\text{-}twl\ S\ S' \mid$
conflict: $\text{conflict-}twl\ S\ S' \Longrightarrow cdcl_W\text{-}twl\ S\ S' \mid$
other: $cdcl_W\text{-}twl.cdcl_W\text{-}o\ S\ S' \Longrightarrow cdcl_W\text{-}twl\ S\ S' \mid$
rf: $cdcl_W\text{-}twl.cdcl_W\text{-}rf\ S\ S' \Longrightarrow cdcl_W\text{-}twl\ S\ S'$

lemma $cdcl_W\text{-}twl\text{-}iff\text{-}cdcl_W$:
assumes $cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $cdcl_W\text{-}twl\ S\ T \longleftrightarrow cdcl_W\text{-}twl.cdcl_W\ S\ T$
by (*simp add: assms $cdcl_W\text{-}twl.cdcl_W.simps\ cdcl_W\text{-}twl.simps\ conflict\text{-}twl\text{-}iff\text{-}conflict\ propagate\text{-}twl\text{-}iff\text{-}propagate$*)

lemma $rtrancpl\text{-}cdcl_W\text{-}twl\text{-}all\text{-}struct\text{-}inv\text{-}inv$:
assumes $cdcl_W\text{-}twl^{**}\ S\ T$ **and** $cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$
using *assms* **by** (*induction rule: $rtrancpl\text{-}induct$*)
(simp-all add: $cdcl_W\text{-}twl\text{-}iff\text{-}cdcl_W\ cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$)

lemma $rtrancpl\text{-}cdcl_W\text{-}twl\text{-}iff\text{-}rtrancpl\text{-}cdcl_W$:
assumes $cdcl_W\text{-}twl.cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
shows $cdcl_W\text{-}twl^{**}\ S\ T \longleftrightarrow cdcl_W\text{-}twl.cdcl_W^{**}\ S\ T$ (**is** $?T \longleftrightarrow ?W$)

proof

assume $?W$
then show $?T$
proof (*induction rule: $rtrancpl\text{-}induct$*)
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 $cdcl_W\text{-}twl\ T\ U$
using *assms* $st\ cdcl\ cdcl_W\text{-}twl.rtrancpl\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv\ cdcl_W\text{-}twl\text{-}iff\text{-}cdcl_W$
by *blast*
then show $?case$ **using** IH **by** *auto*
qed

next

assume $?T$
then show $?W$
proof (*induction rule: $rtrancpl\text{-}induct$*)
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 $cdcl_W\text{-}twl.cdcl_W\ T\ U$
using *assms* $st\ cdcl\ rtrancpl\text{-}cdcl_W\text{-}twl\text{-}all\text{-}struct\text{-}inv\text{-}inv\ cdcl_W\text{-}twl\text{-}iff\text{-}cdcl_W$
by *blast*
then show $?case$ **using** IH **by** *auto*
qed

qed

interpretation $cdcl_{NOT}\text{-}twl$: *backjumping-ops*

$\lambda S.$ *convert-trail-from- W ($trail\text{-}twl\ S$)*

abstract- $twl.raw\text{-}clauses\text{-}twl$

$\lambda L\ (S:: 'v \text{ wf-}twl).$

cons-trail- twl

(convert-marked-lit-from-NOT L) (S:: 'v wf-twl)
 tl-trail-twl
 add-learned-cls-twl
 remove-cls-twl
 λC - - (S:: 'v wf-twl) -. $C \in \text{candidates-conflict-twl } S$
 by unfold-locales

lemma reduce-trail-to_{NOT}-skip-beginning-twl:
assumes trail-twl $S = \text{convert-trail-from-NOT } (F' @ F)$
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 $S = \text{convert-trail-from-NOT } (F' @ \text{Marked } K () \# F) \implies$
 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) \geq 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

interpretation cdcl_{NOT}-twl: dpll-with-backjumping-ops

λS . convert-trail-from-W (trail-twl S)
 abstract-twl.raw-clauses-twl
 λL S.
 cons-trail-twl
 (convert-marked-lit-from-NOT L) S
 tl-trail-twl
 add-learned-cls-twl
 remove-cls-twl
 λL S. lit-of L \in fst ' candidates-propagate-twl S
 λS . no-dup (trail-twl S)
 λC - - S -. $C \in \text{candidates-conflict-twl } S$

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-twl (cdcl_W-twl.reduce-trail-to_{NOT} F S)))
let ?T = (cons-trail-twl (Propagated L {#}) (cdcl_W-twl.reduce-trail-to_{NOT} F S))
have tr-F-S: map lit-of (trail-twl (cdcl_W-twl.reduce-trail-to_{NOT} F S)) =
 map lit-of (convert-trail-from-NOT F)
apply (subst trail-twl-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-twl S)
  using 1(1) by blast
have wf-twl-state (rough-state-of-twl (cdclW-twl.reduce-trail-toNOT F S))
  using wf-twl-state-rough-state-of-twl by blast
moreover have undef': undefined-lit (trail-twl (cdclW-twl.reduce-trail-toNOT F S)) L
  using undef arg-cong[OF tr-F-S, of map atm-of] unfolding defined-lit-map image-set
  by (simp add: o-def)
ultimately have wf-twl-state ?T'
  by (simp-all add: wf-twl-state-cons-trail)
then have init-clss-twl ?T = init-clss-twl (cdclW-twl.reduce-trail-toNOT F S)
  using 1(6) by (simp add: undef')
then have [simp]: init-clss-twl ?T = init-clss-twl S
  by (simp add: cdclW-twl.reduce-trail-toNOT-reduce-trail-convert)

have learned-clss-twl ?T = learned-clss-twl (cdclW-twl.reduce-trail-toNOT 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-twl-def)
moreover have learned-clss-twl (cdclW-twl.reduce-trail-toNOT F S)
  = learned-clss-twl S
  by (simp add: cdclW-twl.reduce-trail-toNOT-reduce-trail-convert)
ultimately have [simp]: learned-clss-twl ?T = learned-clss-twl S
  by simp
have tr-L-F-S: map lit-of (trail-twl ?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-twl S
  apply (rule wf-candidates-twl-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-twl.backjump S
  (cons-trail-twl (convert-marked-lit-from-NOT (Propagated L ()))
   (cdclW-twl.reduce-trail-toNOT F S))
  apply (rule cdclNOT-twl.backjump.intros[of S F' K F - L C, OF 1(3) - 1(4-6) - 1(8-9)])
  unfolding cdclW-twl-NOT.state-eqNOT-def apply (metis convert-marked-lit-from-NOT.simps(1))
  using 1(7) 1(3) apply presburger
  using C-conflict-cand by simp
then show ?case
  by blast
qed

interpretation cdclNOT-twl: dp11-with-backjumping
  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

```

```

λL S. lit-of L ∈ fst ' candidates-propagate-tw1 S
λS. no-dup (trail-tw1 S)
λC - - (S:: 'v wf-tw1) -. C ∈ candidates-conflict-tw1 S
apply unfold-locales
using cdclNOT-tw1.dpll-bj-no-dup by (simp add: o-def)
end

end
theory Prop-Superposition
imports Partial-Clausal-Logic ../lib/Herbrand-Interpretation
begin
sledgehammer-params[verbose]
no-notation Herbrand-Interpretation.true-cls (infix  $\models$  50)
notation Herbrand-Interpretation.true-cls (infix  $\models_h$  50)

no-notation Herbrand-Interpretation.true-clss (infix  $\models_s$  50)
notation Herbrand-Interpretation.true-clss (infix  $\models_{hs}$  50)

lemma herbrand-interp-iff-partial-interp-cls:
  S  $\models_h$  C  $\longleftrightarrow$  {Pos P|P. P∈S} ∪ {Neg P|P. P∉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∈S} ∪ {Neg P|P. P∉S})
  unfolding consistent-interp-def by auto

lemma herbrand-total-over-set:
  total-over-set ({Pos P|P. P∈S} ∪ {Neg P|P. P∉S}) T
  unfolding total-over-set-def by auto

lemma herbrand-total-over-m:
  total-over-m ({Pos P|P. P∈S} ∪ {Neg P|P. P∉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∈S} ∪ {Neg P|P. P∉S}  $\models_s$  C
  unfolding true-clss-def Ball-def herbrand-interp-iff-partial-interp-cls
  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 ∈ N. D #⊂# C}

notation (latex output)
  clss-lt ( $\prec^{\sup}$   $\prec^{\sup}$ )

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  $N :: 'a \text{ clause set}$ 
begin

```

We do not create an equivalent of δ , but we directly defined N_C by inlining the definition.

```

function
  production :: 'a clause  $\Rightarrow$  'a interp
where
  production  $C =$ 
    { $A. C \in N \wedge C \neq \{\#\} \wedge \text{Max } (\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1$ 
       $\wedge \neg (\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D) \models_h C \wedge S C = \{\#\}$ }
  by auto
termination by (relation {( $D, C$ ).  $D \# \subset \# C$ }) (auto simp: wf-less-multiset)

```

```

declare production.simps[simp del]

```

```

definition interp :: 'a clause  $\Rightarrow$  'a interp where
  interp  $C = (\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D)$ 

```

```

lemma production-unfold:
  production  $C = \{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max } (\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge \neg$ 
   $\text{interp } C \models_h C \wedge S C = \{\#\}\}$ 
  unfolding interp-def by (rule production.simps)

```

```

abbreviation productive  $A \equiv (\text{production } A \neq \{\})$ 

```

```

abbreviation produces :: 'a clause  $\Rightarrow$  'a  $\Rightarrow$  bool where
  produces  $C A \equiv \text{production } C = \{A\}$ 

```

```

lemma producesD:
  produces  $C A \Longrightarrow C \in N \wedge C \neq \{\#\} \wedge \text{Pos } A = \text{Max } (\text{set-mset } C) \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge \neg$ 
   $\text{interp } C \models_h C \wedge S C = \{\#\}$ 
  unfolding production-unfold by auto

```

```

lemma produces  $C A \Longrightarrow \text{Pos } A \in \# C$ 
  by (simp add: Max-in-lits producesD)

```

```

lemma interp'-def-in-set:
  interp  $C = (\bigcup D \in \{D \in N. D \# \subset \# C\}. \text{production } D)$ 
  unfolding interp-def apply auto
  unfolding production-unfold apply auto
  done

```

```

lemma production-iff-produces:
  produces  $D A \longleftrightarrow A \in \text{production } D$ 
  unfolding production-unfold by auto

```

```

definition Interp :: 'a clause  $\Rightarrow$  'a interp where
  Interp  $C = \text{interp } C \cup \text{production } C$ 

```

```

lemma
  assumes produces  $C P$ 
  shows Interp  $C \models_h C$ 
  unfolding Interp-def assms using producesD[OF assms]

```


by (metis Max-in-lits Un-insert-right insertI1 pos-literal-in-imp-true-clb)

definition *INTERP* :: 'a interp **where**
INTERP = ($\bigcup D \in N.$ production *D*)

lemma *interp-subseteq-Interp[simp]*: *interp C* \subseteq *Interp C*
unfolding *Interp-def* **by** *simp*

lemma *Interp-as-UNION*: *Interp C* = ($\bigcup D \in \{D. D \# \subseteq \# C\}.$ production *D*)
unfolding *Interp-def* *interp-def* *le-multiset-def* **by** *fast*

lemma *productive-not-empty*: *productive C* $\implies C \neq \{\#\}$
unfolding *production-unfold* **by** *auto*

lemma *productive-imp-produces-Max-literal*: *productive C* \implies *produces C* (*atm-of* (*Max* (*set-mset C*)))
unfolding *production-unfold* **by** (*auto simp del: atm-of-Max-lit*)

lemma *productive-imp-produces-Max-atom*: *productive C* \implies *produces C* (*Max* (*atms-of C*))
unfolding *atms-of-def* *Max-atm-of-set-mset-commute[OF productive-not-empty]*
by (*rule productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-literal*: *produces C A* $\implies A = \text{atm-of } (\text{Max } (\text{set-mset } C))$
by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-atom*: *produces C A* $\implies A = \text{Max } (\text{atms-of } C)$
by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-atom*)

lemma *produces-imp-Pos-in-lits*: *produces C A* $\implies \text{Pos } A \in \# C$
by (*auto intro: Max-in-lits dest!: producesD*)

lemma *productive-in-N*: *productive C* $\implies C \in N$
unfolding *production-unfold* **by** *auto*

lemma *produces-imp-atms-leq*: *produces C A* $\implies B \in \text{atms-of } C \implies B \leq A$
by (*metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom singleton-inject*)

lemma *produces-imp-neg-notin-lits*: *produces C A* $\implies \neg \text{Neg } A \in \# C$
by (*rule pos-Max-imp-neg-notin*) (*auto dest: producesD*)

lemma *less-eq-imp-interp-subseteq-interp*: *C* $\# \subseteq \# D \implies \text{interp } C \subseteq \text{interp } D$
unfolding *interp-def* **by** *auto* (*metis multiset-order.order.strict-trans2*)

lemma *less-eq-imp-interp-subseteq-Interp*: *C* $\# \subseteq \# D \implies \text{interp } C \subseteq \text{Interp } D$
unfolding *Interp-def* **using** *less-eq-imp-interp-subseteq-interp* **by** *blast*

lemma *less-imp-production-subseteq-interp*: *C* $\# \subset \# D \implies \text{production } C \subseteq \text{interp } D$
unfolding *interp-def* **by** *fast*

lemma *less-eq-imp-production-subseteq-Interp*: *C* $\# \subseteq \# D \implies \text{production } C \subseteq \text{Interp } D$
unfolding *Interp-def* **using** *less-imp-production-subseteq-interp*
by (*metis multiset-order.le-imp-less-or-eq le-supI1 sup-ge2*)

lemma *less-imp-Interp-subseteq-interp*: *C* $\# \subset \# D \implies \text{Interp } C \subseteq \text{interp } D$

unfolding *Interp-def*
by (*auto simp: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)

lemma *less-eq-imp-Interp-subseteq-Interp*: $C \# \subseteq \# D \implies \text{Interp } C \subseteq \text{Interp } D$
using *less-imp-Interp-subseteq-interp*
unfolding *Interp-def* **by** (*metis multiset-order.le-imp-less-or-eq le-supI2 subset-refl sup-commute*)

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin \text{Interp } C \implies A \in \text{interp } D \implies C \# \subset \# D$
using *less-eq-imp-interp-subseteq-Interp multiset-linorder.not-less* **by** *blast*

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin \text{interp } C \implies A \in \text{interp } D \implies C \# \subset \# D$
using *less-eq-imp-interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin \text{Interp } C \implies A \in \text{Interp } D \implies C \# \subset \# D$
using *less-eq-imp-Interp-subseteq-Interp multiset-linorder.not-less* **by** *blast*

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin \text{interp } C \implies A \in \text{Interp } D \implies C \# \subseteq \# D$
using *less-imp-Interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

lemma *interp-subseteq-INTERP*: $\text{interp } C \subseteq \text{INTERP}$
unfolding *interp-def INTERP-def* **by** (*auto simp: production-unfold*)

lemma *production-subseteq-INTERP*: $\text{production } C \subseteq \text{INTERP}$
unfolding *INTERP-def* **using** *production-unfold* **by** *blast*

lemma *Interp-subseteq-INTERP*: $\text{Interp } C \subseteq \text{INTERP}$
unfolding *Interp-def* **by** (*auto intro!: interp-subseteq-INTERP production-subseteq-INTERP*)

This lemma corresponds to theorem 2.7.6 page 66 of CW.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: $\text{Neg } A \in \# C$ **and** *d*: *produces* $D A$
shows $A \in \text{interp } C$

proof –

from *d* **have** $\text{Max } (\text{set-mset } D) = \text{Pos } A$
using *production-unfold* **by** *blast*
hence $D \# \subset \# \{\# \text{Neg } A \#\}$
by (*auto intro: Max-pos-neg-less-multiset*)
moreover have $\{\# \text{Neg } A \#\} \# \subseteq \# C$
by (*rule less-eq-imp-le-multiset*) (*rule mset-le-single[OF a-in-c[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*: $\text{Neg } A \in \# C \implies A \notin \text{Interp } D \implies C \# \subseteq \# D \implies \neg \text{produces } D'' A$
by (*auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-Interp*)

lemma *in-production-imp-produces*: $A \in \text{production } C \implies \text{produces } C A$
by (*metis insert-absorb productive-imp-produces-Max-atom singleton-insert-inj-eq'*)

lemma *not-produces-imp-notin-production*: $\neg \text{produces } C A \implies A \notin \text{production } C$
by (*metis in-production-imp-produces*)

lemma *not-produces-imp-notin-interp*: $(\bigwedge D. \neg \text{produces } D A) \implies A \notin \text{interp } C$
unfolding *interp-def* **by** (*fast intro!: in-production-imp-produces*)

The results below corresponds to Lemma 3.4.

Nitpicking: If $D = D'$ and D is productive, $I^D \subseteq I_{D'}$ does not hold.

lemma *true-Interp-imp-general:*

```

assumes
  c-le-d:  $C \# \subseteq \# D$  and
  d-lt-d':  $D \# \subset \# D'$  and
  c-at-d:  $\text{Interp } D \models_h C$  and
  subs:  $\text{interp } D' \subseteq (\bigcup C \in CC. \text{production } C)$ 
shows  $(\bigcup C \in CC. \text{production } C) \models_h C$ 
proof (cases  $\exists A. \text{Pos } A \in \# C \wedge A \in \text{Interp } D$ )
case True
then obtain  $A$  where  $a\text{-in-}c: \text{Pos } A \in \# C$  and  $a\text{-at-}d: A \in \text{Interp } D$ 
by blast
from  $a\text{-at-}d$  have  $A \in \text{interp } D'$ 
using  $d\text{-lt-}d'$  less-imp-Interp-subseteq-interp by blast
thus ?thesis
using  $\text{subs } a\text{-in-}c$  by (blast dest: contra-subsetD)
next
case False
then obtain  $A$  where  $a\text{-in-}c: \text{Neg } A \in \# C$  and  $A \notin \text{Interp } D$ 
using  $c\text{-at-}d$  unfolding true-cls-def by blast
hence  $\bigwedge D''. \neg \text{produces } D'' A$ 
using  $c\text{-le-}d$  neg-notin-Interp-not-produce by simp
thus ?thesis
using  $a\text{-in-}c$  subs not-produces-imp-notin-production by auto
qed

```

lemma *true-Interp-imp-interp:* $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{interp } D' \models_h C$
using *interp-def true-Interp-imp-general* **by** *simp*

lemma *true-Interp-imp-Interp:* $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{Interp } D' \models_h C$
using *Interp-as-UNION interp-subseteq-Interp true-Interp-imp-general* **by** *simp*

lemma *true-Interp-imp-INTERP:* $C \# \subseteq \# D \implies \text{Interp } D \models_h C \implies \text{INTERP} \models_h C$
using *INTERP-def interp-subseteq-INTERP*
true-Interp-imp-general[OF - less-multiset-right-total]
by *simp*

lemma *true-interp-imp-general:*

```

assumes
  c-le-d:  $C \# \subseteq \# D$  and
  d-lt-d':  $D \# \subset \# D'$  and
  c-at-d:  $\text{interp } D \models_h C$  and
  subs:  $\text{interp } D' \subseteq (\bigcup C \in CC. \text{production } C)$ 
shows  $(\bigcup C \in CC. \text{production } C) \models_h C$ 
proof (cases  $\exists A. \text{Pos } A \in \# C \wedge A \in \text{interp } D$ )
case True
then obtain  $A$  where  $a\text{-in-}c: \text{Pos } A \in \# C$  and  $a\text{-at-}d: A \in \text{interp } D$ 
by blast
from  $a\text{-at-}d$  have  $A \in \text{interp } D'$ 
using  $d\text{-lt-}d'$  less-eq-imp-interp-subseteq-interp[OF multiset-order.less-imp-le] by blast
thus ?thesis
using  $\text{subs } a\text{-in-}c$  by (blast dest: contra-subsetD)
next
case False

```

then obtain A **where** $a\text{-in-}c$: $Neg\ A \in\# C$ **and** $A \notin \text{interp } D$
using $c\text{-at-}d$ **unfolding** true-cls-def **by** blast
hence $\bigwedge D''. \neg \text{produces } D''\ A$
using $c\text{-le-}d$ **by** $(\text{auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-interp})$
thus $?thesis$
using $a\text{-in-}c\ \text{subs not-produces-imp-notin-production}$ **by** auto
qed

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

lemma $\text{true-interp-imp-interp}$: $C \# \subseteq\# D \implies D \# \subset\# D' \implies \text{interp } D \models_h C \implies \text{interp } D' \models_h C$
using $\text{interp-def true-interp-imp-general}$ **by** simp

lemma $\text{true-interp-imp-Interp}$: $C \# \subseteq\# D \implies D \# \subset\# D' \implies \text{interp } D \models_h C \implies \text{Interp } D' \models_h C$
using $\text{Interp-as-UNION interp-subseteq-Interp[of } D']\ \text{true-interp-imp-general}$ **by** simp

lemma $\text{true-interp-imp-INTERP}$: $C \# \subseteq\# D \implies \text{interp } D \models_h C \implies \text{INTERP} \models_h C$
using $\text{INTERP-def interp-subseteq-INTERP}$
 $\text{true-interp-imp-general[OF - less-multiset-right-total]}$
by simp

lemma $\text{productive-imp-false-interp}$: $\text{productive } C \implies \neg \text{interp } C \models_h C$
unfolding production-unfold **by** auto

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

lemma $\text{cls-gt-double-pos-no-production}$:
assumes D : $\{\#Pos\ P, Pos\ P\#\} \# \subset\# C$
shows $\neg \text{produces } C\ P$
proof $-$
let $?D = \{\#Pos\ P, Pos\ P\#\}$
note $D' = D[\text{unfolded less-multiset}_{HO}]$
consider
 $(P)\ \text{count } C\ (Pos\ P) \geq 2$
 $| (Q)\ Q\ \text{where } Q > Pos\ P\ \text{and } Q \in\# C$
using $HOL.spec[OF HOL.conjunct2[OF D'], of Pos\ P]$ **by** auto
thus $?thesis$
proof cases
case Q
have $Q \in \text{set-mset } C$
using $Q(2)$ **by** $(\text{auto split: split-if-asm})$
then have $\text{Max } (\text{set-mset } C) > Pos\ P$
using $Q(1)\ \text{Max-gr-iff}$ **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 $\text{production } D \neq \{P\}$
proof $-$

```

note  $D' = D[\text{unfolded less-multiset}_{HO}]$ 
consider
  (P)  $Neg\ P \in \# D$ 
| (Q) Q where  $Q > Neg\ P$  and  $count\ D\ Q > count\ (C + \{\#Neg\ P\# \})\ Q$ 
  using  $HOL.spec[OF\ HOL.conjunct2[OF\ D],\ of\ Neg\ P]$  by fastforce
thus ?thesis
proof cases
  case Q
  have  $Q \in set-mset\ D$ 
    using  $Q(2)$  by (auto split: split-if-asm)
  then have  $Max\ (set-mset\ D) > Neg\ P$ 
    using  $Q(1)\ Max-gr-iff$  by blast
  hence  $Max\ (set-mset\ D) > Pos\ P$ 
    using  $less-trans[of\ Pos\ P\ Neg\ P\ Max\ (set-mset\ D)]$  by auto
  thus ?thesis
    unfolding production-unfold by auto
next
  case P
  hence  $Max\ (set-mset\ D) > Pos\ P$ 
    by (meson Max-ge finite-set-mset le-less-trans linorder-not-le mem-set-mset-iff
      pos-less-neg)
  thus ?thesis
    unfolding production-unfold by auto
qed
qed

```

```

lemma in-interp-is-produced:
  assumes  $P \in INTERP$ 
  shows  $\exists D. D + \{\#Pos\ P\# \} \in N \wedge produces\ (D + \{\#Pos\ P\# \})\ P$ 
  using assms unfolding INTERP-def UN-iff production-iff-produces Ball-def
  by (metis ground-resolution-with-selection.produces-imp-Pos-in-lits insert-DiffM2
    ground-resolution-with-selection-axioms not-produces-imp-notin-production)

```

end
end

abbreviation $MMax\ M \equiv Max\ (set-mset\ M)$

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 $C\ N = (class-lt\ N\ C \models_p C)$

lemma *less-multiset[iff]*: $M < N \longleftrightarrow M \# \subset \# N$
unfolding *less-multiset-def* **by** *auto*

lemma *less-eq-multiset[iff]*: $M \leq N \longleftrightarrow M \# \subseteq \# N$

unfolding *less-eq-multiset-def* **by** *auto*

lemma *herbrand-true-clss-true-clss-clss-herbrand-true-clss*:

assumes

AB: $A \models_{hs} B$ **and**

BC: $B \models_p C$

shows $A \models_h C$

proof –

let $?I = \{Pos\ P \mid P. P \in A\} \cup \{Neg\ P \mid P. P \notin A\}$

have $B: ?I \models_s B$ **using** *AB*

by (*auto simp add: herbrand-interp-iff-partial-interp-clss*)

have $IH: \bigwedge I. total-over-set\ I\ (atms-of\ C) \implies total-over-m\ I\ B \implies consistent-interp\ I \implies I \models_s B \implies I \models C$ **using** *BC*

by (*auto simp add: true-clss-clss-def*)

show *?thesis*

unfolding *herbrand-interp-iff-partial-interp-clss*

by (*auto intro: IH[of ?I] simp add: herbrand-total-over-set herbrand-total-over-m herbrand-consistent-interp B*)

qed

lemma *abstract-red-subset-mset-abstract-red*:

assumes

abstr: *abstract-red* $C\ N$ **and**

c-lt-d: $C \subseteq\# D$

shows *abstract-red* $D\ N$

proof –

have $\{D \in N. D \# \subset \# C\} \subseteq \{D' \in N. D' \# \subset \# D\}$

using *c-lt-d less-eq-imp-le-multiset* **by** *fastforce*

thus *?thesis*

using *abstr* **unfolding** *abstract-red-def clss-lt-def*

by (*metis (no-types, lifting) c-lt-d subset-mset.diff-add true-clss-clss-mono-r' true-clss-clss-subset*)

qed

lemma *true-clss-clss-extended*:

assumes

$A \models_p B$ **and**

tot: *total-over-m* $I\ (A)$ **and**

cons: *consistent-interp* I **and**

$I-A: I \models_s A$

shows $I \models B$

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  $\forall a. a \notin \text{atms-of-ms } A \vee \text{Pos } a \in I \vee \text{Neg } a \in I$ 
using tot by (simp add: total-over-m-def total-over-set-def)
hence  $aa (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\}) (I \cup \{\text{Pos } a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})$ 
 $\notin \text{atms-of-ms } A \cup \text{atms-of-ms } \{B\} \vee \text{Pos } (aa (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\})$ 
 $(I \cup \{\text{Pos } a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})) \in I$ 
 $\cup \{\text{Pos } a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}$ 
 $\vee \text{Neg } (aa (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\})$ 
 $(I \cup \{\text{Pos } a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})) \in I$ 
 $\cup \{\text{Pos } a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}$ 
by auto
hence  $\text{total-over-set } (I \cup \{\text{Pos } a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}) (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\})$ 
using f2 by (meson total-over-set-def)
thus ?thesis
by (simp add: total-over-m-def)
qed
moreover have  $?I \models_s A$ 
using I-A by auto
ultimately have  $?I \models B$ 
using  $\langle A \models_p B \rangle$  unfolding true-clss-cls-def by auto
thus ?thesis
oops
lemma
assumes
 $CP: \neg \text{clss-lt } N (\{\#C\# \} + \{\#E\# \}) \models_p \{\#C\# \} + \{\#Neg P\# \}$  and
 $\text{clss-lt } N (\{\#C\# \} + \{\#E\# \}) \models_p \{\#E\# \} + \{\#Pos P\# \} \vee \text{clss-lt } N (\{\#C\# \} + \{\#E\# \}) \models_p$ 
 $\{\#C\# \} + \{\#Neg P\# \}$ 
shows  $\text{clss-lt } N (\{\#C\# \} + \{\#E\# \}) \models_p \{\#E\# \} + \{\#Pos P\# \}$ 
oops

locale ground-ordered-resolution-with-redundancy =
 $\text{ground-resolution-with-selection} +$ 
fixes  $\text{redundant} :: 'a::\text{wellorder clause} \Rightarrow 'a \text{ clauses} \Rightarrow \text{bool}$ 
assumes
 $\text{redundant-iff-abstract: redundant } A \ N \longleftrightarrow \text{abstract-red } A \ N$ 
begin
definition  $\text{saturated} :: 'a \text{ clauses} \Rightarrow \text{bool}$  where
 $\text{saturated } N \longleftrightarrow (\forall A \ B \ C. A \in N \longrightarrow B \in N \longrightarrow \neg \text{redundant } A \ N \longrightarrow \neg \text{redundant } B \ N$ 
 $\longrightarrow \text{superposition-rules } A \ B \ C \longrightarrow \text{redundant } C \ N \vee C \in N)$ 
lemma
assumes
 $\text{saturated: saturated } N$  and
 $\text{finite: finite } N$  and
 $\text{empty: } \{\#\} \notin N$ 
shows  $\text{INTERP } N \models_{hs} N$ 
proof (rule ccontr)
let  $?N_{\mathcal{I}} = \text{INTERP } N$ 
assume  $\neg ?thesis$ 
hence  $\text{not-empty: } \{E \in N. \neg ?N_{\mathcal{I}} \models_h E\} \neq \{\}$ 
unfolding true-clss-def Ball-def by auto
def  $D \equiv \text{Min } \{E \in N. \neg ?N_{\mathcal{I}} \models_h E\}$ 
have  $[\text{simp}]: D \in N$ 
unfolding D-def

```

```

  by (metis (mono-tags, lifting) Min-in not-empty finite mem-Collect-eq rev-finite-subset subsetI)
have not-d-interp:  $\neg ?N_{\mathcal{I}} \models_h D$ 
  unfolding D-def
  by (metis (mono-tags, lifting) Min-in finite mem-Collect-eq not-empty rev-finite-subset subsetI)
have cls-not-D:  $\bigwedge E. E \in N \implies E \neq D \implies \neg ?N_{\mathcal{I}} \models_h E \implies D \leq E$ 
  using finite D-def by (auto simp del: less-eq-multiset)
obtain C L where D:  $D = C + \{\#L\# \}$  and LSD:  $L \in \# S D \vee (S D = \{\# \} \wedge \text{Max } (\text{set-mset } D) = L)$ 
proof (cases  $S D = \{\# \}$ )
  case False
  then obtain L where  $L \in \# S D$ 
    using Max-in-lits by blast
  moreover
    hence  $L \in \# D$ 
      using S-selects-subseteq[of D] by auto
    hence  $D = (D - \{\#L\# \}) + \{\#L\# \}$ 
      by auto
    ultimately show ?thesis using that by blast
next
let ?L = MMax D
case True
moreover
  have  $?L \in \# D$ 
    by (metis (no-types, lifting) Max-in-lits  $\langle D \in N \rangle$  empty)
  hence  $D = (D - \{\#?L\# \}) + \{\#?L\# \}$ 
    by auto
  ultimately show ?thesis using that by blast
qed
have red:  $\neg \text{redundant } D N$ 
proof (rule ccontr)
  assume red[simplified]:  $\sim \sim \text{redundant } D N$ 
  have  $\forall E < D. E \in N \longrightarrow ?N_{\mathcal{I}} \models_h E$ 
    using cls-not-D not-le by fastforce
  hence  $?N_{\mathcal{I}} \models_{hs} \text{clss-lt } N D$ 
    unfolding clss-lt-def true-clss-def Ball-def by blast
  thus False
    using red not-d-interp unfolding abstract-red-def redundant-iff-abstract
    using herbrand-true-clss-true-clss-cls-herbrand-true-clss by fast
qed
consider
  (L) P where  $L = \text{Pos } P$  and  $S D = \{\# \}$  and  $\text{Max } (\text{set-mset } D) = \text{Pos } P$ 
| (Lneg) P where  $L = \text{Neg } P$ 
  using LSD S-selects-neg-lits[of D L] by (cases L) auto
thus False
proof cases
  case L note P = this(1) and S = this(2) and max = this(3)
  have count D L > 1
    proof (rule ccontr)
      assume  $\sim ?thesis$ 
      hence count:  $\text{count } D L = 1$ 
        unfolding D by auto
      have  $\neg ?N_{\mathcal{I}} \models_h D$ 
        using not-d-interp true-interp-imp-INTERP ground-resolution-with-selection-axioms
        by blast

```


hence produces $N \ D \ P$
 using *not-empty empty finite* $\langle D \in N \rangle$ *count L*
true-interp-imp-INTERP unfolding production-iff-produces unfolding production-unfold
 by (*auto simp add: max not-empty*)
 hence $INTERP \ N \models_h \ D$
 unfolding D
 by (*metis pos-literal-in-imp-true-cls produces-imp-Pos-in-lits*
production-subseteq-INTERP singletonI subsetCE)
 thus *False*
 using *not-d-interp* by *blast*
 qed
 then obtain C' where $C':D = C' + \{\#Pos \ P\# \} + \{\#Pos \ P\# \}$
 unfolding D by (*metis P add.left-neutral add-less-cancel-right count-single count-union*
multi-member-split)
 have *sup: superposition-rules D D (D - {\#L\#})*
 unfolding $C' \ L$ by (*auto simp add: superposition-rules.simps*)
 have $C' + \{\#Pos \ P\# \} \# \subset \# \ C' + \{\#Pos \ P\# \} + \{\#Pos \ P\# \}$
 by *auto*
 moreover have $\neg ?N_{\mathcal{I}} \models_h (D - \{\#L\# \})$
 using *not-d-interp unfolding C' L by auto*
 ultimately have $C' + \{\#Pos \ P\# \} \notin N$
 by (*metis (no-types, lifting) C' P add-diff-cancel-right' cls-not-D less-multiset*
multi-self-add-other-not-self not-le)
 have $D - \{\#L\# \} \# \subset \# \ D$
 unfolding $C' \ L$ by *auto*
 have $c'-p-p: C' + \{\#Pos \ P\# \} + \{\#Pos \ P\# \} - \{\#Pos \ P\# \} = C' + \{\#Pos \ P\# \}$
 by *auto*
 have *redundant (C' + {\#Pos P\#}) N*
 using *saturated red sup* $\langle D \in N \rangle \langle C' + \{\#Pos \ P\# \} \notin N \rangle$ *unfolding saturated-def C' L c'-p-p*
 by *blast*
 moreover have $C' + \{\#Pos \ P\# \} \subseteq \# \ C' + \{\#Pos \ P\# \} + \{\#Pos \ P\# \}$
 by *auto*
 ultimately show *False*
 using *red unfolding C' redundant-iff-abstract by (blast dest:*
abstract-red-subset-mset-abstract-red)
 next
 case *Lneg* note $L = this(1)$
 have $P \in ?N_{\mathcal{I}}$
 using *not-d-interp unfolding D true-cls-def L by (auto split: split-if-asm)*
 then obtain E where
DPN: E + {\#Pos P\#} \in N and
prod: production N (E + {\#Pos P\#}) = \{P\}
 using *in-interp-is-produced by blast*
 have *sup-EC: superposition-rules (E + {\#Pos P\#}) (C + {\#Neg P\#}) (E + C)*
 using *superposition-l by fast*
 hence *superposition N (N \cup \{E+C\})*
 using *DPN* $\langle D \in N \rangle$ *unfolding D L by (auto simp add: superposition.simps)*
 have
PMax: Pos P = MMax (E + {\#Pos P\#}) and
count (E + {\#Pos P\#}) (Pos P) \leq 1 and
S (E + {\#Pos P\#}) = \{\#\} and
 $\neg interp \ N \ (E + \{\#Pos \ P\# \}) \models_h E + \{\#Pos \ P\# \}$
 using *prod unfolding production-unfold by auto*
 have $Neg \ P \notin \# \ E$
 using *prod produces-imp-neg-notin-lits by force*

hence $\bigwedge y. y \in \# (E + \{\#Pos P\})$
 $\implies count (E + \{\#Pos P\}) (Neg P) < count (C + \{\#Neg P\}) (Neg P)$
by (*auto split: split-if-asm*)
moreover have $\bigwedge y. y \in \# (E + \{\#Pos P\}) \implies y < Neg P$
using *PMax by (metis DPN Max-less-iff empty finite-set-mset mem-set-mset-iff pos-less-neg set-mset-eq-empty-iff)*
moreover have $E + \{\#Pos P\} \neq C + \{\#Neg P\}$
using *prod produces-imp-neg-notin-lits by force*
ultimately have $E + \{\#Pos P\} \# \subset \# C + \{\#Neg P\}$
unfolding *less-multiset_{HO} by (metis add.left-neutral add-lessD1)*
have *ce-lt-d: C + E # \subset \# D*
unfolding *D L*
by (*metis (mono-tags, lifting) Max-pos-neg-less-multiset One-nat-def PMax count-single less-multiset-plus-right-nonempty mult-less-trans single-not-empty union-less-mono2 zero-less-Suc*)
have $?N_{\mathcal{I}} \models_h E + \{\#Pos P\}$
using $\langle P \in ?N_{\mathcal{I}} \rangle$ **by** *blast*
have $?N_{\mathcal{I}} \models_h C+E \vee C+E \notin N$
using *ce-lt-d cls-not-D unfolding D-def by fastforce*
have $Pos P \notin \# C+E$
using $D \langle P \in ground-resolution-with-selection.INTERP S N \rangle$
 $\langle count (E + \{\#Pos P\}) (Pos P) \leq 1 \rangle$ **multi-member-skip not-d-interp by** *auto*
hence $\bigwedge y. y \in \# C+E$
 $\implies count (C+E) (Pos P) < count (E + \{\#Pos P\}) (Pos P)$
by (*auto split: split-if-asm*)

have $\neg redundant (C + E) N$
proof (*rule ccontr*)
assume *red'[simplified]: \neg ?thesis*
have *abs: clss-lt N (C + E) \models_p C + E*
using *redundant-iff-abstract red' unfolding abstract-red-def by auto*
have *clss-lt N (C + E) \models_p E + \{\#Pos P\} \vee clss-lt N (C + E) \models_p C + \{\#Neg P\}*
proof *clarify*
assume *CP: \neg clss-lt N (C + E) \models_p C + \{\#Neg P\}*
{ fix I
assume
total-over-m I (clss-lt N (C + E) \cup \{E + \{\#Pos P\}\}) and
consistent-interp I and
I \models_s clss-lt N (C + E)
hence $I \models C + E$
using *abs sorry*
moreover have $\neg I \models C + \{\#Neg P\}$
using *CP unfolding true-clss-cls-def*
sorry
ultimately have $I \models E + \{\#Pos P\}$ **by** *auto*
}
then show $clss-lt N (C + E) \models_p E + \{\#Pos P\}$
unfolding *true-clss-cls-def by auto*
qed
moreover have $clss-lt N (C + E) \subseteq clss-lt N (C + \{\#Neg P\})$
using *ce-lt-d mult-less-trans unfolding clss-lt-def D L by force*
ultimately have $redundant (C + \{\#Neg P\}) N \vee clss-lt N (C + E) \models_p E + \{\#Pos P\}$
unfolding *redundant-iff-abstract abstract-red-def using true-clss-cls-subset by blast*
show *False sorry*
qed

```

    moreover have  $\neg$  redundant  $(E + \{\#Pos\ P\# \})\ N$ 
      sorry
    ultimately have  $CEN: C + E \in N$ 
      using  $\langle D \in N \rangle \langle E + \{\#Pos\ P\# \} \in N \rangle$  saturated sup-EC red unfolding saturated-def  $D\ L$ 
      by (metis union-commute)
    have  $CED: C + E \neq D$ 
      using  $D\ ce-lt-d$  by auto
    have interp:  $\neg INTERP\ N \models_h C + E$ 
      sorry
    show False
      using cls-not-D[OF  $CEN\ CED\ interp$ ] ce-lt-d unfolding INTERP-def less-eq-multiset-def by
auto
qed
qed

end

lemma tautology-is-redundant:
  assumes tautology  $C$ 
  shows abstract-red  $C\ N$ 
  using assms unfolding abstract-red-def true-clss-cl-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-cl-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

```

22 Implementation for 2 Watched-Literals

```

theory CDCL-Two-Watched-Literals-Implementation
imports CDCL-Two-Watched-Literals DPLL-CDCL-W-Implementation
begin

type-synonym conc-twl-state =
  ((nat, nat, nat literal list) marked-lit, nat literal list twl-clause list, nat, nat literal list)
  twl-state

fun convert :: ('a, 'b, 'c list) marked-lit  $\Rightarrow$  ('a, 'b, 'c multiset) marked-lit where
  convert (Propagated L C) = Propagated L (mset C) |
  convert (Marked K i) = Marked K i

abbreviation convert-tr :: ('a, 'b, 'c list) marked-lits  $\Rightarrow$  ('a, 'b, 'c multiset) marked-lits
  where
  convert-tr  $\equiv$  map convert

abbreviation convertC :: 'a literal list option  $\Rightarrow$  'a clause option where
  convertC  $\equiv$  map-option mset

fun raw-clause-l :: 'v list twl-clause  $\Rightarrow$  'v multiset twl-clause where
  raw-clause-l (TWL-Clause UW W) = TWL-Clause (mset W) (mset UW)

abbreviation convert-clss :: 'v literal list twl-clause list  $\Rightarrow$  'v clause twl-clause multiset
  where
  convert-clss S  $\equiv$  mset (map raw-clause-l S)

fun raw-state-of-conc :: conc-twl-state  $\Rightarrow$  (nat, nat, nat clause) twl-state-abs where
  raw-state-of-conc (TWL-State M N U k C) =
    TWL-State (convert-tr M) (convert-clss N) (convert-clss U) k (map-option mset C)

lemma
  raw-state-of-conc (tl-trail S) = tl-trail (raw-state-of-conc S)
  unfolding tl-trail-def by (induction S) (auto simp: map-tl)

```

definition *watch-nat* :: *conc-twl-state* \Rightarrow *nat literal list* \Rightarrow *nat literal list twl-clause* **where**
watch-nat *S* *C* =
 (let
C' = *remdups* *C*;
negation-not-assigned = *filter* ($\lambda L. -L \notin \text{lits-of } (\text{trail } S)$) *C'*;
negation-assigned-sorted-by-trail = *filter* ($\lambda L. L \in \text{set } C$) (*map* ($\lambda L. -\text{lit-of } L$) (*trail* *S*));
W = *take* 2 (*negation-not-assigned* @ *negation-assigned-sorted-by-trail*);
UW = *foldl* ($\lambda a \ l. \text{remove1 } l \ a$) *C* *W*
 in *TWL-Clause* *W* *UW*)

definition
rewatch-nat ::
 (*nat*, *nat*, *nat literal list*) *marked-lit* \Rightarrow *conc-twl-state* \Rightarrow
nat literal list twl-clause \Rightarrow *nat literal list twl-clause*

where

rewatch-nat *L* *S* *C* =
 (if $-\text{lit-of } L \in \text{set } (\text{watched } C)$ then
 case *filter* ($\lambda L'. L' \notin \text{set } (\text{watched } C) \wedge -L' \notin \text{lits-of } (L \# \text{trail } S)$)
 (*unwatched* *C*) of
 [] \Rightarrow *C*
 | *L' # -* \Rightarrow
TWL-Clause (*L' # remove1* ($-\text{lit-of } L$) (*watched* *C*))
 ($-\text{lit-of } L \# \text{remove1 } L'$ (*unwatched* *C*))
 else
C)

definition *raw-candidates-conflict* :: *conc-twl-state* \Rightarrow *nat literal list list* **where**
raw-candidates-conflict *S* =
map ($\lambda T. \text{case } T \text{ of } \text{TWL-Clause } W \ UW \Rightarrow W @ UW$)
 (*filter* ($\lambda C. \text{set } (\text{watched } C) \subseteq (\text{uminus 'lits-of } (\text{trail } S))$)
 (*init-clss* *S* @ *learned-clss* *S*))

definition *do-conflict-step* :: *conc-twl-state* \Rightarrow *conc-twl-state option* **where**
do-conflict-step *S* =
 (case *conflicting* *S* of
 Some - \Rightarrow None
 | None \Rightarrow
 (case *raw-candidates-conflict* *S* of
 [] \Rightarrow None
 | *a # -* \Rightarrow Some (*update-conflicting* (Some *a*) *S*)))

end