

Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

June 2, 2016

Contents

1	More Standard Theorems	7
1.1	More about Multisets	7
1.1.1	Basic Setup	7
1.1.2	Lemmas about intersections	7
1.1.3	Lemmas about size	8
1.1.4	Multiset Extension of Multiset Ordering	9
1.1.5	Remove	10
1.1.6	Replicate	11
1.1.7	Multiset and set conversion	11
1.1.8	Removing duplicates	12
1.1.9	Filter	16
1.1.10	Sums	17
1.1.11	Order	17
1.2	Transitions	17
1.2.1	More theorems about Closures	18
1.2.2	Full Transitions	19
1.2.3	Well-Foundedness and Full Transitions	20
1.2.4	More Well-Foundedness	21
1.3	Various Lemmas	23
1.4	More List	25
1.4.1	<i>upt</i>	25
1.4.2	Lexicographic Ordering	28
1.4.3	Remove	28
2	Definition of Entailment	31
2.1	Clausal Logic	31
2.1.1	Literals	31
2.1.2	Clauses	34
2.2	Herbrand Intepretation	36
2.2.1	Herbrand Interpretations	36
2.3	Partial Clausal Logic	38
2.3.1	Clauses	38
2.3.2	Partial Interpretations	38
2.3.3	Subsumptions	54
2.3.4	Removing Duplicates	55
2.3.5	Set of all Simple Clauses	55
2.3.6	Experiment: Expressing the Entailments as Locales	59
2.3.7	Entailment to be extended	59

3	Normalisation	63
3.1	Logics	63
3.1.1	Definition and abstraction	63
3.1.2	properties of the abstraction	64
3.1.3	Subformulas and properties	67
3.1.4	Positions	70
3.2	Semantics over the syntax	73
3.3	Rewrite systems and properties	74
3.3.1	Lifting of rewrite rules	74
3.3.2	Consistency preservation	77
3.3.3	Full Lifting	78
3.4	Transformation testing	78
3.4.1	Definition and first properties	78
3.4.2	Invariant conservation	81
3.5	Rewrite Rules	84
3.5.1	Elimination of the equivalences	84
3.5.2	Eliminate Implication	86
3.5.3	Eliminate all the True and False in the formula	87
3.5.4	PushNeg	93
3.5.5	Push inside	98
3.6	The full transformations	112
3.6.1	Abstract Property characterizing that only some connective are inside the others	112
3.6.2	Conjunctive Normal Form	114
3.6.3	Disjunctive Normal Form	115
3.7	More aggressive simplifications: Removing true and false at the beginning	116
3.7.1	Transformation	116
3.7.2	More invariants	118
3.7.3	The new CNF and DNF transformation	122
3.8	Link with Multiset Version	123
3.8.1	Transformation to Multiset	123
3.8.2	Equisatisfiability of the two Version	123
4	Resolution-based techniques	127
4.1	Resolution	127
4.1.1	Simplification Rules	127
4.1.2	Unconstrained Resolution	129
4.1.3	Inference Rule	129
4.1.4	Lemma about the simplified state	144
4.1.5	Resolution and Invariants	147
4.2	Superposition	167
4.2.1	We can now define the rules of the calculus	174
4.3	Partial Clausal Logic	180
4.3.1	Decided Literals	180
4.3.2	Backtracking	184
4.3.3	Decomposition with respect to the First Decided Literals	185
4.3.4	Negation of Clauses	192
4.3.5	Other	196
4.3.6	Extending Entailments to multisets	197

5	NOT's CDCL and DPLL	199
5.1	Measure	199
5.2	NOT's CDCL	203
5.2.1	Auxiliary Lemmas and Measure	203
5.2.2	Initial definitions	204
5.2.3	DPLL with backjumping	208
5.2.4	CDCL	224
5.2.5	CDCL with restarts	246
5.2.6	Merging backjump and learning	254
5.2.7	Instantiations	265
5.3	DPLL as an instance of NOT	280
5.3.1	DPLL with simple backtrack	280
5.3.2	Adding restarts	285
5.4	Weidenbach's DPLL	285
5.4.1	Rules	285
5.4.2	Invariants	286
5.4.3	Termination	294
5.4.4	Final States	296
5.4.5	Link with NOT's DPLL	298
6	Weidenbach's CDCL	305
6.1	Weidenbach's CDCL with Multisets	305
6.1.1	The State	305
6.1.2	CDCL Rules	313
6.1.3	Structural Invariants	320
6.1.4	CDCL Strong Completeness	346
6.1.5	Higher level strategy	347
6.1.6	Termination	392
6.2	Merging backjump rules	414
6.2.1	Inclusion of the states	414
6.2.2	More lemmas conflict-propagate and backjumping	416
6.2.3	CDCL with Merging	431
6.2.4	CDCL with Merge and Strategy	436
6.3	Link between Weidenbach's and NOT's CDCL	470
6.3.1	Inclusion of the states	470
6.3.2	Additional Lemmas between NOT and W states	474
6.3.3	Inclusion of Weidenbach's CDCL in NOT's CDCL	475
6.3.4	Correctness of <i>cdcl_W-merge-stgy</i>	480
6.3.5	Adding Restarts	482
6.4	Incremental SAT solving	493
7	Implementation of DPLL and CDCL	505
7.1	Simple List-Based Implementation of the DPLL and CDCL	505
7.1.1	Common Rules	505
7.1.2	CDCL specific functions	508
7.1.3	Simple Implementation of DPLL	510
7.1.4	List-based CDCL Implementation	520
7.1.5	CDCL Implementation	523
7.1.6	Abstract Clause Representation	549
7.2	Weidenbach's CDCL with Abstract Clause Representation	550

7.2.1	Instantiation of the Multiset Version	550
7.2.2	Abstract Relation and Relation Theorems	552
7.2.3	The State	557
7.2.4	CDCL Rules	565
7.2.5	Higher level strategy	580
7.3	2-Watched-Literal	584
7.3.1	Essence of 2-WL	585
7.3.2	Two Watched-Literals with invariant	606

Chapter 1

More Standard Theorems

This chapter contains additional lemmas built on top of HOL.

end

```
theory Multiset-More
imports ~~/src/HOL/Library/Multiset-Order
begin
```

1.1 More about Multisets

Isabelle's theory of finite multisets is not as developed as other areas, such as lists and sets. The present theory introduces some missing concepts and lemmas. Some of it is expected to move to Isabelle's library.

1.1.1 Basic Setup

declare

diff-single-trivial [simp]

in-image-mset [iff]

image-mset.compositionality [simp]

mset-leD[*dest*, *intro?*]

Multiset.in-multiset-in-set[simp]

lemma *image-mset-cong2*[*cong*]:

$(\bigwedge x. x \in \# M \implies f x = g x) \implies M = N \implies \text{image-mset } f M = \text{image-mset } g N$

by (*hypsubst*, *rule image-mset-cong*)

lemma *subset-msetE* [*elim!*]:

$[| A \subset \# B; [| A \subseteq \# B; \sim (B \subseteq \# A) |] \implies R |] \implies R$

unfolding *subteq-mset-def subset-mset-def* **by** (*meson mset-less-eqI subset-mset.eq-iff*)

1.1.2 Lemmas about intersections

lemma *mset-inter-single*:

$x \in \# \Sigma \implies \Sigma \# \cap \{\#x\} = \{\#x\}$
 $x \notin \# \Sigma \implies \Sigma \# \cap \{\#x\} = \{\#\}$
apply (*simp add: mset-le-single subset-mset.inf-absorb2*)
by (*simp add: multiset-inter-def*)

1.1.3 Lemmas about size

This sections adds various lemmas about size. Most lemmas have a finite set equivalent.

lemma *size-mset-SucE*: $\text{size } A = \text{Suc } n \implies (\bigwedge a B. A = \{\#a\} + B \implies \text{size } B = n \implies P) \implies P$
by (*cases A*) (*auto simp add: ac-simps*)

lemma *size-Suc-Diff1*:
 $x \in \# \Sigma \implies \text{Suc } (\text{size } (\Sigma - \{\#x\})) = \text{size } \Sigma$
using *arg-cong[OF insert-DiffM, of - - size]* **by** *simp*

lemma *size-Diff-singleton*: $x \in \# \Sigma \implies \text{size } (\Sigma - \{\#x\}) = \text{size } \Sigma - 1$
by (*simp add: size-Suc-Diff1 [symmetric]*)

lemma *size-Diff-singleton-if*: $\text{size } (A - \{\#x\}) = (\text{if } x \in \# A \text{ then } \text{size } A - 1 \text{ else } \text{size } A)$
by (*simp add: size-Diff-singleton*)

lemma *size-Un-Int*:
 $\text{size } A + \text{size } B = \text{size } (A \# \cup B) + \text{size } (A \# \cap B)$
proof –
have *: $A + B = B + (A - B + (A - (A - B)))$
by (*simp add: subset-mset.add-diff-inverse union-commute*)
have $\text{size } B + \text{size } (A - B) = \text{size } A + \text{size } (B - A)$
unfolding *size-union[symmetric] subset-mset.sup-commute sup-subset-mset-def[symmetric]*
by *blast*
then show *?thesis* **unfolding** *multiset-inter-def size-union[symmetric]* *
by (*auto simp add: sup-subset-mset-def*)
qed

lemma *size-Un-disjoint*:
assumes $A \# \cap B = \{\#\}$
shows $\text{size } (A \# \cup B) = \text{size } A + \text{size } B$
using *assms size-Un-Int [of A B]* **by** *simp*

lemma *size-Diff-subset-Int*:
shows $\text{size } (\Sigma - \Sigma') = \text{size } \Sigma - \text{size } (\Sigma \# \cap \Sigma')$
proof –
have *: $\Sigma - \Sigma' = \Sigma - \Sigma \# \cap \Sigma'$ **by** (*auto simp add: multiset-eq-iff*)
show *?thesis* **unfolding** * **using** *size-Diff-submset subset-mset.inf.cobounded1* **by** *blast*
qed

lemma *diff-size-le-size-Diff*: $\text{size } (\Sigma :: \text{multiset}) - \text{size } \Sigma' \leq \text{size } (\Sigma - \Sigma')$
proof –
have $\text{size } \Sigma - \text{size } \Sigma' \leq \text{size } \Sigma - \text{size } (\Sigma \# \cap \Sigma')$
using *size-mset-mono diff-le-mono2 subset-mset.inf-le2* **by** *blast*
also have $\dots = \text{size } (\Sigma - \Sigma')$ **by** (*simp add: size-Diff-subset-Int*)
finally show *?thesis* .
qed

lemma *size-Diff1-less*: $x \in \# \Sigma \implies \text{size } (\Sigma - \{\#x\}) < \text{size } \Sigma$
apply (*rule Suc-less-SucD*)

by (simp add: size-Suc-Diff1)

lemma size-Diff2-less: $x \in \# \Sigma \implies y \in \# \Sigma \implies \text{size } (\Sigma - \{\#x\} - \{\#y\}) < \text{size } \Sigma$
 using nonempty-has-size by (fastforce intro!: diff-Suc-less simp add: size-Diff1-less
 size-Diff-subset-Int mset-inter-single)

lemma size-Diff1-le: $\text{size } (\Sigma - \{\#x\}) \leq \text{size } \Sigma$
 by (cases $x \in \# \Sigma$) (simp-all add: size-Diff1-less less-imp-le)

lemma size-psubset: $(\Sigma :: \text{multiset}) \leq \# \Sigma' \implies \text{size } \Sigma < \text{size } \Sigma' \implies \Sigma < \# \Sigma'$
 using less-irrefl subset-mset-def by blast

1.1.4 Multiset Extension of Multiset Ordering

The $op \# \subset \# \#$ and $op \# \subseteq \# \#$ operators are introduced as the multiset extension of the multiset orderings of $op \# \subset \#$ and $op \# \subseteq \#$.

definition less-mset-mset :: $('a :: \text{order}) \text{ multiset multiset} \Rightarrow 'a \text{ multiset multiset} \Rightarrow \text{bool}$
 (infix $\# < \# \#$ 50)

where

$M' \# < \# \# M \longleftrightarrow (M', M) \in \text{mult } \{(x', x). x' \# < \# x\}$

definition le-mset-mset :: $('a :: \text{order}) \text{ multiset multiset} \Rightarrow 'a \text{ multiset multiset} \Rightarrow \text{bool}$
 (infix $\# \leq \# \#$ 50)

where

$M' \# \leq \# \# M \longleftrightarrow M' \# < \# \# M \vee M' = M$

notation less-mset-mset (infix $\# \subset \# \#$ 50)

notation le-mset-mset (infix $\# \subseteq \# \#$ 50)

lemmas less-mset-mset_{DM} = order.mult_{DM}[OF order-multiset, folded less-mset-mset-def]

lemmas less-mset-mset_{HO} = order.mult_{HO}[OF order-multiset, folded less-mset-mset-def]

interpretation multiset-multiset-order: order

le-mset-mset :: $('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow \text{bool}$

less-mset-mset :: $('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow \text{bool}$

unfolding less-mset-mset-def[abs-def] le-mset-mset-def[abs-def] less-multiset-def[abs-def]

by (rule order.order-mult)+ standard

interpretation multiset-multiset-linorder: linorder

le-mset-mset :: $('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow \text{bool}$

less-mset-mset :: $('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow ('a :: \text{linorder}) \text{ multiset multiset} \Rightarrow \text{bool}$

unfolding less-mset-mset-def[abs-def] le-mset-mset-def[abs-def]

by (rule linorder.linorder-mult[OF linorder-multiset])

lemma wf-less-mset-mset: wf $\{(\Sigma :: ('a :: \text{wellorder}) \text{ multiset multiset}, T). \Sigma \# \subset \# \# T\}$

unfolding less-mset-mset-def by (auto intro: wf-mult wf-less-multiset)

interpretation multiset-multiset-wellorder: wellorder

le-mset-mset :: $('a :: \text{wellorder}) \text{ multiset multiset} \Rightarrow ('a :: \text{wellorder}) \text{ multiset multiset} \Rightarrow \text{bool}$

less-mset-mset :: $('a :: \text{wellorder}) \text{ multiset multiset} \Rightarrow ('a :: \text{wellorder}) \text{ multiset multiset} \Rightarrow \text{bool}$

by unfold-locales (blast intro: wf-less-mset-mset[unfolded wf-def, rule-format])

lemma union-less-mset-mset-mono2: $B \# \subset \# \# D \implies C + B \# \subset \# \# C + (D :: 'a :: \text{order multiset multiset})$

apply (unfold less-mset-mset-def mult-def)

apply (erule trancl-induct)
apply (blast intro: mult1-union)
apply (blast intro: mult1-union trancl-trans)
done

lemma union-less-mset-mset-diff-plus:
 $U \leq \# \Sigma \implies T \# \subset \# \# U \implies \Sigma - U + T \# \subset \# \# \Sigma$
apply (erule subset-mset.diff-add[symmetric])
using union-less-mset-mset-mono2[of $T U \Sigma - U$] **by** simp

lemma ex-gt-imp-less-mset-mset:
 $(\exists y :: 'a :: \text{linorder multiset} \in \# T. (\forall x. x \in \# \Sigma \longrightarrow x \# \subset \# y)) \implies \Sigma \# \subset \# \# T$
using less-mset-mset_{HO} **by** (metis count-greater-zero-iff count-inI less-nat-zero-code
 multiset-linorder.not-less-iff-gr-or-eq)

1.1.5 Remove

lemma set-mset-minus-replicate-mset[simp]:
 $n \geq \text{count } A \ a \implies \text{set-mset } (A - \text{replicate-mset } n \ a) = \text{set-mset } A - \{a\}$
 $n < \text{count } A \ a \implies \text{set-mset } (A - \text{replicate-mset } n \ a) = \text{set-mset } A$
unfolding set-mset-def **by** (auto split: if-split simp: not-in-iff)

abbreviation removeAll-mset :: $'a \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$ **where**
 removeAll-mset $C \ M \equiv M - \text{replicate-mset } (\text{count } M \ C) \ C$

lemma mset-removeAll[simp, code]:
 removeAll-mset $C \ (\text{mset } L) = \text{mset } (\text{removeAll } C \ L)$
by (induction L) (auto simp: ac-simps multiset-eq-iff split: if-split-asm)

lemma removeAll-mset-filter-mset:
 removeAll-mset $C \ M = \text{filter-mset } (op \neq C) \ M$
by (induction M) (auto simp: ac-simps multiset-eq-iff)

abbreviation remove1-mset :: $'a \Rightarrow 'a \text{ multiset} \Rightarrow 'a \text{ multiset}$ **where**
 remove1-mset $C \ M \equiv M - \{\#C\#\}$

lemma remove1-mset-remove1[code]:
 remove1-mset $C \ (\text{mset } L) = \text{mset } (\text{remove1 } C \ L)$
by auto

lemma in-remove1-mset-neg:
assumes $ab: a \neq b$
shows $a \in \# \text{remove1-mset } b \ C \longleftrightarrow a \in \# C$

proof –

have $\text{count } \{\#b\# \} \ a = 0$
using ab **by** simp
then show ?thesis
by (metis (no-types) count-diff diff-zero mem-Collect-eq set-mset-def)

qed

lemma size-mset-removeAll-mset-le-iff:
 $\text{size } (\text{removeAll-mset } x \ M) < \text{size } M \longleftrightarrow x \in \# M$
apply (rule iffI)
apply (force intro: count-inI)
apply (rule mset-less-size)
apply (auto simp: subset-mset-def multiset-eq-iff)

done

lemma *size-mset-remove1-mset-le-iff*:
 $\text{size } (\text{remove1-mset } x \ M) < \text{size } M \longleftrightarrow x \in \# \ M$
apply (rule *iffI*)
using *less-irrefl* **apply** *fastforce*
apply (rule *mset-less-size*)
by (auto elim: *in-countE simp: subset-mset-def multiset-eq-iff*)

lemma *set-mset-remove1-mset[simp]*:
 $\text{set-mset } (\text{remove1-mset } L \ (\text{mset } W)) = \text{set } (\text{remove1 } L \ W)$
by (metis *mset-remove1 set-mset-mset*)

1.1.6 Replicate

lemma *replicate-mset-plus*: $\text{replicate-mset } (a + b) \ C = \text{replicate-mset } a \ C + \text{replicate-mset } b \ C$
by (induct *a*) (auto simp: *ac-simps*)

lemma *mset-replicate-replicate-mset*:
 $\text{mset } (\text{replicate } n \ L) = \text{replicate-mset } n \ L$
by (induction *n*) auto

lemma *set-mset-single-iff-replicate-mset*:
 $\text{set-mset } U = \{a\} \longleftrightarrow (\exists n > 0. \ U = \text{replicate-mset } n \ a) \ (\text{is } ?S \longleftrightarrow ?R)$

proof

assume *?R*

then show *?S* **by** auto

next

assume *?S*

show *?R*

proof (rule *ccontr*)

assume $\neg ?R$

have $\forall n. \ U \neq \text{replicate-mset } n \ a$

using $\langle ?S \rangle \langle \neg ?R \rangle$ **by** (metis *gr-zeroI insert-not-empty set-mset-replicate-mset-subset*)

then obtain *b* **where** $b \in \# \ U$ **and** $b \neq a$

by (metis *count-replicate-mset mem-Collect-eq multiset-eqI neq0-conv set-mset-def*)

then show *False*

using $\langle ?S \rangle$ **by** auto

qed

qed

1.1.7 Multiset and set conversion

lemma *count-mset-set-if*:
 $\text{count } (\text{mset-set } A) \ a = (\text{if } a \in A \wedge \text{finite } A \text{ then } 1 \text{ else } 0)$
by auto

lemma *mset-set-set-mset-empty-mempty[iff]*:
 $\text{mset-set } (\text{set-mset } D) = \{\#\} \longleftrightarrow D = \{\#\}$
by (auto dest: *arg-cong[of - - set-mset]*)

lemma *size-mset-set-card*:
 $\text{finite } S \implies \text{size } (\text{mset-set } S) = \text{card } S$
by (induction *S* rule: *finite-induct*) auto

lemma *count-mset-set-le-one*: $\text{count } (\text{mset-set } A) \ x \leq 1$

by (metis count-mset-set(1) count-mset-set(2) count-mset-set(3) eq-iff le-numeral-extra(1))

lemma mset-set-subseteq-mset-set[iff]:

assumes finite A finite B

shows mset-set A $\subseteq\#$ mset-set B \longleftrightarrow A \subseteq B

by (metis assms contra-subsetD count-mset-set(1,3) count-mset-set-le-one finite-set-mset-mset-set less-eq-nat.simps(1) mset-less-eqI set-mset-mono)

lemma mset-set-set-mset-subseteq[simp]: mset-set (set-mset A) $\subseteq\#$ A

by (metis count-mset-set(1,3) finite-set-mset less-eq-nat.simps(1) less-one mem-Collect-eq mset-less-eqI not-less set-mset-def)

lemma mset-sorted-list-of-set[simp]:

mset (sorted-list-of-set A) = mset-set A

by (metis mset-sorted-list-of-multiset sorted-list-of-mset-set)

lemma mset-take-subseteq: mset (take n xs) $\subseteq\#$ mset xs

apply (induct xs arbitrary: n)

apply simp

by (case-tac n) simp-all

1.1.8 Removing duplicates

definition remdups-mset :: 'v multiset \Rightarrow 'v multiset **where**

remdups-mset S = mset-set (set-mset S)

lemma remdups-mset-in[iff]: a $\in\#$ remdups-mset A \longleftrightarrow a $\in\#$ A

unfolding remdups-mset-def **by** auto

lemma count-remdups-mset-eq-1: a $\in\#$ remdups-mset A \longleftrightarrow count (remdups-mset A) a = 1

unfolding remdups-mset-def **by** (auto simp: count-eq-zero-iff intro: count-inI)

lemma remdups-mset-empty[simp]:

remdups-mset {#} = {#}

unfolding remdups-mset-def **by** auto

lemma remdups-mset-singleton[simp]:

remdups-mset {#a#} = {#a#}

unfolding remdups-mset-def **by** auto

lemma set-mset-remdups[simp]: set-mset (remdups-mset C) = set-mset C

by auto

lemma remdups-mset-eq-empty[iff]:

remdups-mset D = {#} \longleftrightarrow D = {#}

unfolding remdups-mset-def **by** blast

lemma remdups-mset-singleton-sum[simp]:

remdups-mset ({#a#} + A) = (if a $\in\#$ A then remdups-mset A else {#a#} + remdups-mset A)

remdups-mset (A + {#a#}) = (if a $\in\#$ A then remdups-mset A else {#a#} + remdups-mset A)

unfolding remdups-mset-def **by** (simp-all add: insert-absorb)

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

mset (remdups D) = remdups-mset (mset D)

by (induction D) (auto simp add: ac-simps)

definition *distinct-mset* :: 'a multiset \Rightarrow bool **where**
distinct-mset $S \longleftrightarrow (\forall a. a \in \# S \longrightarrow \text{count } S \ a = 1)$

lemma *distinct-mset-empty[simp]*: *distinct-mset* $\{\#\}$
unfolding *distinct-mset-def* **by** *auto*

lemma *distinct-mset-singleton[simp]*: *distinct-mset* $\{\#a\#\}$
unfolding *distinct-mset-def* **by** *auto*

definition *distinct-mset-set* :: 'a multiset set \Rightarrow bool **where**
distinct-mset-set $\Sigma \longleftrightarrow (\forall S \in \Sigma. \text{distinct-mset } S)$

lemma *distinct-mset-set-empty[simp]*:
distinct-mset-set $\{\}$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-singleton[iff]*:
distinct-mset-set $\{A\} \longleftrightarrow \text{distinct-mset } A$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-insert[iff]*:
distinct-mset-set $(\text{insert } S \ \Sigma) \longleftrightarrow (\text{distinct-mset } S \wedge \text{distinct-mset-set } \Sigma)$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-set-union[iff]*:
distinct-mset-set $(\Sigma \cup \Sigma') \longleftrightarrow (\text{distinct-mset-set } \Sigma \wedge \text{distinct-mset-set } \Sigma')$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-union*:
assumes *dist*: *distinct-mset* $(A + B)$
shows *distinct-mset* A

proof –

obtain *aa* :: 'a multiset \Rightarrow 'a **where**
f2: $\forall m. \neg \text{distinct-mset } m \vee (\forall a. (a::'a) \notin \# m \vee \text{count } m \ a = 1)$
 $\forall m. aa \ m \in \# m \wedge \text{count } m \ (aa \ m) \neq 1 \vee \text{distinct-mset } m$
by (*metis* (*full-types*) *distinct-mset-def*)
then have *count* $(A + B) \ (aa \ A) = 1 \vee \text{distinct-mset } A$
using *dist* **by** (*meson* *mset-leD* *mset-le-add-left*)
then show *?thesis*
using *f2* **by** (*metis* (*no-types*) *One-nat-def* *add-is-1* *count-union* *mem-Collect-eq* *order-less-irrefl* *set-mset-def*)

qed

lemma *distinct-mset-minus[simp]*:
distinct-mset $A \Longrightarrow \text{distinct-mset } (A - B)$
by (*metis* *Multiset.diff-le-self* *mset-le-exists-conv* *distinct-mset-union*)

lemma *in-distinct-mset-set-distinct-mset*:
 $a \in \Sigma \Longrightarrow \text{distinct-mset-set } \Sigma \Longrightarrow \text{distinct-mset } a$
unfolding *distinct-mset-set-def* **by** *auto*

lemma *distinct-mset-remdups-mset[simp]*: *distinct-mset* $(\text{remdups-mset } S)$
using *count-remdups-mset-eq-1* **unfolding** *distinct-mset-def* **by** *metis*

lemma *distinct-mset-distinct[simp]*:
distinct-mset $(\text{mset } x) = \text{distinct } x$

unfolding *distinct-mset-def* **by** (auto simp: *distinct-count-atmost-1 not-in-iff*[*symmetric*])

lemma *distinct-mset-mset-set*:

distinct-mset (*mset-set* *A*)

unfolding *distinct-mset-def count-mset-set-if* **by** (auto simp: *not-in-iff*)

lemma *distinct-mset-rempdups-union-mset*:

assumes *distinct-mset* *A* **and** *distinct-mset* *B*

shows $A \# \cup B = \text{remdups-mset } (A + B)$

using *assms nat-le-linear* **unfolding** *remdups-mset-def*

by (force simp add: *multiset-eq-iff max-def count-mset-set-if distinct-mset-def not-in-iff*)

lemma *distinct-mset-set-distinct*:

distinct-mset-set (*mset* ' *set* *Cs*) $\longleftrightarrow (\forall c \in \text{set } Cs. \text{distinct } c)$

unfolding *distinct-mset-set-def* **by** auto

lemma *distinct-mset-add-single*:

distinct-mset ($\{\#a\# \} + L$) $\longleftrightarrow \text{distinct-mset } L \wedge a \notin \# L$

unfolding *distinct-mset-def*

apply (rule *iffI*)

prefer 2 **apply** (auto simp: *not-in-iff*)[]

apply *standard*

apply (*intro allI*)

apply (*rename-tac aa, case-tac a = aa*)

by (auto split: *if-split-asm*)

lemma *distinct-mset-single-add*:

distinct-mset ($L + \{\#a\# \}$) $\longleftrightarrow \text{distinct-mset } L \wedge a \notin \# L$

unfolding *add.commute*[of $L \{\#a\# \}$] *distinct-mset-add-single* **by** *fast*

lemma *distinct-mset-size-eq-card*:

distinct-mset *C* $\implies \text{size } C = \text{card } (\text{set-mset } C)$

by (*induction C*) (auto simp: *distinct-mset-single-add*)

Another characterisation of *distinct-mset*

lemma *distinct-mset-count-less-1*:

distinct-mset *S* $\longleftrightarrow (\forall a. \text{count } S a \leq 1)$

using *eq-iff nat-le-linear* **unfolding** *distinct-mset-def* **by** *fastforce*

lemma *distinct-mset-add*:

distinct-mset ($L + L'$) $\longleftrightarrow \text{distinct-mset } L \wedge \text{distinct-mset } L' \wedge L \# \cap L' = \{\#\}$ (is ?*A* \longleftrightarrow ?*B*)

proof (rule *iffI*)

assume ?*A*

have *L*: *distinct-mset* *L*

using $\langle \text{distinct-mset } (L + L') \rangle$ *distinct-mset-union* **by** *blast*

moreover **have** *L'*: *distinct-mset* *L'*

using $\langle \text{distinct-mset } (L + L') \rangle$ *distinct-mset-union* **unfolding** *add.commute*[of *L L'*] **by** *blast*

moreover **have** $L \# \cap L' = \{\#\}$

using $L \ L' \langle ?A \rangle$ **unfolding** *multiset-inter-def multiset-eq-iff distinct-mset-count-less-1*

by (*metis* *Nat.diff-le-self add-diff-cancel-left' count-diff count-empty diff-is-0-eq eq-iff le-neq-implies-less less-one*)

ultimately show ?*B* **by** *fast*

next

assume ?*B*

show ?*A*

unfolding *distinct-mset-count-less-1*

```

proof (intro allI)
  fix a
  have count (L + L') a ≤ count L a + count L' a
    by auto
  moreover have count L a + count L' a ≤ 1
    using (?B) by (metis One-nat-def add.commute add-decreasing2 count-diff diff-add-zero
      distinct-mset-count-less-1 le-SucE multiset-inter-count plus-multiset.rep-eq
      subset-mset.inf.idem)
  ultimately show count (L + L') a ≤ 1
    by arith
qed
qed

lemma distinct-mset-set-mset-ident[simp]: distinct-mset M  $\implies$  mset-set (set-mset M) = M
  apply (auto simp: multiset-eq-iff)
  apply (rename-tac x)
  apply (case-tac count M x = 0)
  apply (simp add: not-in-iff[symmetric])
  apply (case-tac count M x = 1)
  apply (simp add: count-inI)
  unfolding distinct-mset-count-less-1 by (meson le-neq-implies-less less-one)

lemma distinct-finite-set-mset-subseteq-iff[iff]:
  assumes dist: distinct-mset M and fin: finite N
  shows set-mset M  $\subseteq$  N  $\longleftrightarrow$  M  $\subseteq$  # mset-set N
proof
  assume set-mset M  $\subseteq$  N
  then show M  $\subseteq$  # mset-set N
    by (metis dist distinct-mset-set-mset-ident fin finite-subset mset-set-subseteq-mset-set)
next
  assume M  $\subseteq$  # mset-set N
  then show set-mset M  $\subseteq$  N
    by (metis contra-subsetD empty-iff finite-set-mset-mset-set infinite-set-mset-mset-set
      set-mset-mono subsetI)
qed

lemma distinct-mem-diff-mset:
  assumes dist: distinct-mset M and mem: x ∈ set-mset (M - N)
  shows x ∉ set-mset N
proof -
  have count M x = 1
    using dist mem by (meson distinct-mset-def in-diffD)
  then show ?thesis
    using mem by (metis count-greater-eq-one-iff in-diff-count not-less)
qed

lemma distinct-set-mset-eq:
  assumes
    dist-m: distinct-mset M and
    dist-n: distinct-mset N and
    set-eq: set-mset M = set-mset N
  shows M = N
proof -
  have mset-set (set-mset M) = mset-set (set-mset N)
    using set-eq by simp
  thus ?thesis

```

using *dist-m dist-n* by *auto*
qed

lemma *distinct-mset-union-mset*:

assumes
 distinct-mset D and
 distinct-mset C
shows *distinct-mset (D # \cup C)*
using *assms unfolding distinct-mset-count-less-1* by *force*

lemma *distinct-mset-inter-mset*:

assumes
 distinct-mset D and
 distinct-mset C
shows *distinct-mset (D # \cap C)*
using *assms unfolding distinct-mset-count-less-1*
by (*meson dual-order.trans subset-mset.inf-le2 subseteq-mset-def*)

lemma *distinct-mset-remove1-All*:

distinct-mset C \implies remove1-mset L C = removeAll-mset L C
by (*auto simp: multiset-eq-iff distinct-mset-count-less-1*)

lemma *distinct-mset-size-2*: *distinct-mset {#a, b#} \longleftrightarrow a \neq b*
unfolding *distinct-mset-def* by *auto*

1.1.9 Filter

lemma *mset-filter-compl*: *mset (filter p xs) + mset (filter (Not \circ p) xs) = mset xs*
apply (*induct xs*)
by *simp*
 (*metis (no-types) add-diff-cancel-left' comp-apply filter.simps(2) mset.simps(2)*
 mset-compl-union)

lemma *image-mset-subseteq-mono*: *A \subseteq # B \implies image-mset f A \subseteq # image-mset f B*
by (*metis image-mset-union subset-mset.le-iff-add*)

lemma *image-filter-ne-mset[simp]*:

image-mset f {#x \in # M. f x \neq y#} = removeAll-mset y (image-mset f M)
by (*induct M, auto, meson count-le-replicate-mset-le order-refl subset-mset.add-diff-assoc2*)

lemma *comprehension-mset-False[simp]*:

{# L \in # A. False#} = {#}
by (*auto simp: multiset-eq-iff*)

Near duplicate of *filter-eq-replicate-mset*: *{# y \in # ?D. y = ?x#} = replicate-mset (count ?D ?x) ?x*.

lemma *filter-mset-eq*:

filter-mset (op = L) A = replicate-mset (count A L) L
by (*auto simp: multiset-eq-iff*)

lemma *filter-mset-union-mset*:

filter-mset P (A # \cup B) = filter-mset P A # \cup filter-mset P B
by (*auto simp: multiset-eq-iff*)

lemma *filter-mset-mset-set*:

finite A \implies filter-mset P (mset-set A) = mset-set {a \in A. P a}

by (auto simp: multiset-eq-iff count-mset-set-if)

See *filter-cong* for the set version. Mark as *[fundef-cong]* too?

lemma *filter-mset-cong*:
 assumes $[simp]: M = M'$ and $[simp]: \bigwedge a. a \in \# M \implies P a = Q a$
 shows $filter\text{-}mset\ P\ M = filter\text{-}mset\ Q\ M$
proof –
 have $M - filter\text{-}mset\ Q\ M = filter\text{-}mset\ (\lambda a. \neg Q a)\ M$
 by (subst multiset-partition[of - Q]) simp
 then show ?thesis
 by (auto simp: filter-mset-eq-conv)
qed

1.1.10 Sums

lemma *msetsum-distrib* $[simp]$:
 fixes $C\ D :: 'a \Rightarrow 'b :: \{comm\text{-}monoid\text{-}add\}$
 shows $(\sum x \in \# A. C\ x + D\ x) = (\sum x \in \# A. C\ x) + (\sum x \in \# A. D\ x)$
 by (induction A) (auto simp: ac-simps)

lemma *msetsum-union-disjoint*:
 assumes $A \# \cap B = \{\#\}$
 shows $(\sum La \in \# A \# \cup B. f\ La) = (\sum La \in \# A. f\ La) + (\sum La \in \# B. f\ La)$
 by (metis assms diff-zero empty-sup image-mset-union msetsum.union multiset-inter-commute
 multiset-union-diff-commute sup-subset-mset-def zero-diff)

1.1.11 Order

Instantiating multiset order as a linear order.

TODO: remove when multiset is of sort ord again

instantiation *multiset* :: $(linorder)\ linorder$
begin

definition *less-multiset* :: $'a :: linorder\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ **where**
 $M' < M \longleftrightarrow M' \# \subset \# M$

definition *less-eq-multiset* :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ **where**
 $(M' :: 'a\ multiset) \leq M \longleftrightarrow M' \# \subseteq \# M$

instance
 by standard (auto simp add: less-eq-multiset-def less-multiset-def multiset-order.less-le-not-le
 add.commute multiset-order.add-right-mono)
end
end

1.2 Transitions

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

theory *Wellfounded-More*
imports *Main*

begin

1.2.1 More theorems about Closures

This is the equivalent of the theorem *rtranclp-mono* 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*:

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

apply (*rule iffI*)

prefer 2 **apply** *blast*

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

Equivalent of the theorem *rtranclp-idemp*

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

by *simp*

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

This theorem already exists as theroem *Nitpick.rtranclp-unfold* (and sledgehammer uses it), but it makes sense to duplicate it, because it is unclear how stable the lemmas in the `~~/src/HOL/Nitpick.thy` theory are.

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

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

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

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

Near duplicate of theorem *tranclpD*:

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

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

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

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

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*)

```

case base
then show ?case by auto
next
case (step z' z'') note z = this(2) and IH = this(3)[OF 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*: $(\lambda a b. P a b \wedge Q a b)^{**} S T \implies P^{**} S T$
by (*induction rule: rtranclp-induct*) *auto*

1.2.2 Full Transitions

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

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

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

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

We define output notations only for printing:

notation (**output**) *full1* $(-^{+\downarrow})$

notation (**output**) *full* $(-^{\downarrow})$

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

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

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

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

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

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

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

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

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

lemma *full1-trancpl-relation-full*:
 $\text{full1 } R^{++} a b \longleftrightarrow \text{full1 } R a b$
 by (metis converse-trancplE full1-def reflclp-trancpl rtrancplD rtrancpl-idemp rtrancpl-reflclp
 trancpl.r-into-trancpl trancpl-into-rtrancpl)

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

lemma *rtrancpl-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 rtrancpl.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 mouna
 { 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-trancpl-relation-full)
 then have ?thesis
 using f1 by blast }
 then have ?thesis
 by auto }
 then show ?thesis
 by fastforce
 qed

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

1.2.3 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-def by auto
 have [simp]: $\bigwedge i. F (\text{Suc } i) = (\text{SOME } b. R (F i) b)$
 using F-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  $R (F i) (SOME\ b.\ R (F i)\ b)$ 
    using H by (simp add: someI-ex)
  then have  $\forall j < Suc\ i.\ R (F j) (F (Suc\ j))$ 
    using H Suc by (simp add: less-Suc-eq)
  then show ?case by fast
qed
}
then have  $\forall j.\ R (F j) (F (Suc\ j))$  by blast
then show False
  using wf unfolding wfP-def wf-iff-no-infinite-down-chain by blast
qed

```

```

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

```

1.2.4 More Well-Foundedness

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

- link between *wf* and infinite chains: theorems *wf-iff-no-infinite-down-chain* and *wf-no-infinite-down-chain*

```

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

```

```

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

```

```

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

```

```

lemma wf-wf-if-measure':
  assumes wf r and H:  $\bigwedge x\ y.\ P\ x \implies g\ x\ y \implies (f\ y, f\ x) \in r$ 
  shows  $wf\ \{(y, x).\ P\ x \wedge g\ x\ y\}$ 
proof –
  have  $wf\ \{(b, a).\ (f\ b, f\ a) \in r\}$  using assms(1) wf-if-measure-f by auto
  then have  $wf\ \{(b, a).\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ a) \in r\}$ 
    using wf-subset[of - {(b, a). P a ∧ g a b ∧ (f b, f a) ∈ r}] by auto
  moreover have  $\{(b, a).\ P\ a \wedge g\ a\ b \wedge (f\ b, f\ a) \in r\} \subseteq \{(b, a).\ (f\ b, f\ a) \in r\}$  by auto

```

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

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

proof –

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

qed

lemma *wfP-if-measure2*: fixes $f :: 'a \Rightarrow nat$

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

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

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

apply(*erule wf-subset*)

apply *auto*

done

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

assumes

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

finite: *finite* A and

wf: *wf* R and

trans: *trans* R

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

proof (*rule wfP-if-measure2*)

fix $T\ S$

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

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

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

have $?f\ T \subseteq ?f\ S$

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

moreover have $T \in ?f\ S$

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

moreover have $T \notin ?f\ T$

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

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

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

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

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

qed

lemma *wf-fst-wf-pair*:

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

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

proof –

have *wf* $(\{(M', M). R\ M' M\} <*\text{lex}*> \{\})$

using *assms* by *auto*

then show *?thesis*

by (*rule wf-subset*) *auto*

qed

lemma *wf-snd-wf-pair*:

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

```

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

```

```

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

```

```

end
theory List-More
imports Main ../lib/Multiset-More
begin

```

```

Sledgehammer parameters
sledgehammer-params[debug]

```

1.3 Various Lemmas

Close to the theorem *nat-less-induct* ($(\bigwedge n. \forall m < n. ?P m \implies ?P n) \implies ?P ?n$), but with a separation between the zero and non-zero case.

```

thm nat-less-induct

```

```

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

```

This is only proved in simple cases by auto. In assumptions, nothing happens, and the theorem *if-split-asm* can blow up goals (because of other if-expressions either in the context or as simplification rules).

```

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

```

Bounded function have not yet been defined in Isabelle.

```

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

```

```

abbreviation unbounded :: ( $'a \Rightarrow 'b::\text{ord}$ )  $\Rightarrow$  bool where
   $\text{unbounded } f \equiv \neg \text{bounded } f$ 

```

```

lemma not-bounded-nat-exists-larger:
  fixes  $f :: \text{nat} \Rightarrow \text{nat}$ 
  assumes unbound: unbounded  $f$ 
  shows  $\exists n. f\ n > m \wedge n > n_0$ 
proof (rule ccontr)
  assume  $H: \neg ?thesis$ 
  have finite  $\{f\ n \mid n. n \leq n_0\}$ 
    by auto
  have  $\bigwedge n. f\ n \leq \text{Max } (\{f\ n \mid n. n \leq n_0\} \cup \{m\})$ 
    apply (case-tac n  $\leq n_0$ )
    apply (metis (mono-tags, lifting) Max-ge Un-insert-right  $\langle \text{finite } \{f\ n \mid n. n \leq n_0\} \rangle$ 
      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

```

A function is bounded iff its product with a non-zero constant is bounded. The non-zero condition is needed only for the reverse implication (see for example $k = 0$ and $f = (\lambda i. i)$ for a counter-example).

```

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

```

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

```

lemma bounded-finite-linorder:

```



```

fixes  $f :: 'a \Rightarrow 'a :: \{finite, linorder\}$ 
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

```

1.4 More List

1.4.1 *upt*

The simplification rules are not very handy, because theorem *upt.simps (2)* (i.e. $[?i..<Suc\ ?j] = (if\ ?i \leq\ ?j\ then\ [?i..<?j]\ @\ [?j]\ else\ [])$) leads to a case distinction, that we do not want if the condition is not in the context.

```

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

```

```

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

```

```

declare upt.simps(2)[simp del]

```

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

```

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

```

```

lemma append-cons-eq-upt:
  assumes  $A @ B = [m..<n]$ 
  shows  $A = [m ..<m+length\ A]$  and  $B = [m + length\ A..<n]$ 

```

```

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

```

The converse of theorem *append-cons-eq-upt* does not hold, for example if @ term "B:: nat list" is empty and A is $[0::'a]$:

```

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

```

oops

A more restrictive version holds:

```

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

```

```

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

```

```

next
  assume ?P and ?B
  then show ?A using append-eq-conv-conj by fastforce
qed

lemma append-cons-eq-upt-length-i:
  assumes  $A @ i \# B = [m..<n]$ 
  shows  $A = [m..<i]$ 
proof -
  have  $A = [m..<m + \text{length } A]$  using assms append-cons-eq-upt by auto
  have  $(A @ i \# B) ! (\text{length } A) = i$  by auto
  moreover have  $n - m = \text{length } (A @ i \# B)$ 
    using assms length-upt by presburger
  then have  $[m..<n] ! (\text{length } A) = m + \text{length } A$  by simp
  ultimately have  $i = m + \text{length } A$  using assms by auto
  then show ?thesis using  $\langle A = [m..<m + \text{length } A] \rangle$  by auto
qed

lemma append-cons-eq-upt-length:
  assumes  $A @ i \# B = [m..<n]$ 
  shows  $\text{length } A = i - m$ 
  using assms
proof (induction A arbitrary: m)
  case Nil
  then show ?case by (metis append-Nil diff-is-0-eq list.size(3) order-refl upt-eq-Cons-conv)
next
  case (Cons a A)
  then have  $A : A @ i \# B = [m + 1..<n]$  by (metis append-Cons upt-eq-Cons-conv)
  then have  $m < i$  by (metis Cons.prem1 append-cons-eq-upt-length-i upt-eq-Cons-conv)
  with Cons.IH[OF A] show ?case by auto
qed

lemma append-cons-eq-upt-length-i-end:
  assumes  $A @ i \# B = [m..<n]$ 
  shows  $B = [\text{Suc } i..<n]$ 
proof -
  have  $B = [\text{Suc } m + \text{length } A..<n]$  using assms append-cons-eq-upt[of  $A @ [i] B m n$ ] by auto
  have  $(A @ i \# B) ! (\text{length } A) = i$  by auto
  moreover have  $n - m = \text{length } (A @ i \# B)$ 
    using assms length-upt by auto
  then have  $[m..<n] ! (\text{length } A) = m + \text{length } A$  by simp
  ultimately have  $i = m + \text{length } A$  using assms by auto
  then show ?thesis using  $\langle B = [\text{Suc } m + \text{length } A..<n] \rangle$  by auto
qed

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

lemma upt-decomp-lt:

```

```

assumes  $H$ :  $xs @ i \# ys @ j \# zs = [m ..< n]$ 
shows  $i < j$ 
proof -
  have  $xs$ :  $xs = [m ..< i]$  and  $ys$ :  $ys = [Suc\ i ..< j]$  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

```

The following two lemmas are useful as simp rules for case-distinction. The case $length\ l = 0$ is already simplified by default.

```

lemma length-list-Suc-0:
   $length\ W = Suc\ 0 \longleftrightarrow (\exists L. W = [L])$ 
apply (cases  $W$ )
apply simp
apply (rename-tac  $a\ W'$ , case-tac  $W'$ )
apply auto
done

```

```

lemma length-list-2:  $length\ S = 2 \longleftrightarrow (\exists a\ b. S = [a, b])$ 
apply (cases  $S$ )
apply simp
apply (rename-tac  $a\ S'$ )
apply (case-tac  $S'$ )
by simp-all

```

```

lemma finite-bounded-list:
  fixes  $b :: nat$ 
  shows finite  $\{xs. length\ xs < s \wedge (\forall i < length\ xs. xs\ !\ i < b)\}$  (is finite (?S s))
proof (induction  $s$ )
  case 0
    then show ?case by auto
  next
    case (Suc  $s$ ) note IH = this(1)
    have  $H$ : ?S (Suc  $s$ )  $\subseteq$  ?S  $s \cup \{x \# xs \mid x\ xs. x < b \wedge length\ xs < s \wedge (\forall i < length\ xs. xs\ !\ i < b)\}$ 
       $\cup \{\}\}$ 
      (is -  $\subseteq$  -  $\cup$  ?C  $\cup$  -)
    proof
      fix  $xs$ 
      assume  $xs \in ?S\ (Suc\ s)$ 
      then have  $B$ :  $\forall i < length\ xs. xs\ !\ i < b$  and  $len$ :  $length\ xs < Suc\ s$ 
        by auto
      consider
        ( $st$ )  $length\ xs < s$  |
        ( $s$ )  $length\ xs = 0$  and  $s = 0$  |
        ( $s'$ )  $s'$  where  $length\ xs = Suc\ s'$ 
      using  $len$  by (cases  $s$ ) (auto simp add: Nat.less-Suc-eq)
    then show  $xs \in ?S\ s \cup ?C \cup \{\}\}$ 
      proof cases
        case  $st$ 
          then show ?thesis using  $B$  by auto
        next
          case  $s$ 
            then show ?thesis using  $B$  by auto
        next

```

```

      case s' note len-xs = this(1)
      then obtain x xs' where xs: xs = x # xs' by (cases xs) auto
      then show ?thesis using len-xs B len s' unfolding xs by auto
    qed
  qed
  have ?C ⊆ (case-prod Cons) ‘ ({x. x < b} × ?S s)
  by auto
  moreover have finite ({x. x < b} × ?S s)
  using IH by (auto simp: finite-cartesian-product-iff)
  ultimately have finite ?C by (simp add: finite-surj)
  then have finite (?S s ∪ ?C ∪ {[]})
  using IH by auto
  then show ?case using H by (auto intro: finite-subset)
qed

```

1.4.2 Lexicographic Ordering

lemma *lexn-Suc*:

```

(x # xs, y # ys) ∈ lexn r (Suc n) ⟷
(length xs = n ∧ length ys = n) ∧ ((x, y) ∈ r ∨ (x = y ∧ (xs, ys) ∈ lexn r n))
by (auto simp: map-prod-def image-iff lex-prod-def)

```

lemma *lexn-n*:

```

n > 0 ⟹ (x # xs, y # ys) ∈ lexn r n ⟷
(length xs = n-1 ∧ length ys = n-1) ∧ ((x, y) ∈ r ∨ (x = y ∧ (xs, ys) ∈ lexn r (n - 1)))
apply (cases n)
apply simp
by (auto simp: map-prod-def image-iff lex-prod-def)

```

There is some subtle point in the proof here. *1* is converted to *Suc 0*, but *2* is not: meaning that *1* is automatically simplified by default using the default simplification rule *lexn.simps*. However, the latter needs additional simplification rule (see the proof of the theorem above).

lemma *lexn2-conv*:

```

([a, b], [c, d]) ∈ lexn r 2 ⟷ (a, c) ∈ r ∨ (a = c ∧ (b, d) ∈ r)
by (auto simp: lexn-n simp del: lexn.simps(2))

```

lemma *lexn3-conv*:

```

([a, b, c], [a', b', c']) ∈ lexn r 3 ⟷
(a, a') ∈ r ∨ (a = a' ∧ (b, b') ∈ r) ∨ (a = a' ∧ b = b' ∧ (c, c') ∈ r)
by (auto simp: lexn-n simp del: lexn.simps(2))

```

1.4.3 Remove

More lemmas about remove

lemma *remove1-Nil*:

```

remove1 (- L) W = [] ⟷ (W = [] ∨ W = [-L])
by (cases W) auto

```

lemma *remove1-mset-single-add*:

```

a ≠ b ⟹ remove1-mset a ({#b#} + C) = {#b#} + remove1-mset a C
remove1-mset a ({#a#} + C) = C
by (auto simp: multiset-eq-iff)

```

Remove under condition

This function removes the first element such that the condition f holds. It generalises *remove1*.

```
fun remove1-cond where  
remove1-cond  $f$  [] = [] |  
remove1-cond  $f$  ( $C' \# L$ ) = (if  $f$   $C'$  then  $L$  else  $C' \#$  remove1-cond  $f$   $L$ )
```

```
lemma remove1  $x$   $xs$  = remove1-cond (( $op =$ )  $x$ )  $xs$   
by (induction  $xs$ ) auto
```

```
lemma mset-map-mset-remove1-cond:  
mset (map mset (remove1-cond ( $\lambda L$ . mset  $L =$  mset  $a$ )  $C$ )) =  
remove1-mset (mset  $a$ ) (mset (map mset  $C$ ))  
by (induction  $C$ ) (auto simp: ac-simps remove1-mset-single-add)
```

We can also generalise *removeAll*, which is close to *filter*:

```
fun removeAll-cond where  
removeAll-cond  $f$  [] = [] |  
removeAll-cond  $f$  ( $C' \# L$ ) =  
(if  $f$   $C'$  then removeAll-cond  $f$   $L$  else  $C' \#$  removeAll-cond  $f$   $L$ )
```

```
lemma removeAll  $x$   $xs$  = removeAll-cond (( $op =$ )  $x$ )  $xs$   
by (induction  $xs$ ) auto
```

```
lemma removeAll-cond  $P$   $xs$  = filter ( $\lambda x$ .  $\neg P$   $x$ )  $xs$   
by (induction  $xs$ ) auto
```

```
lemma mset-map-mset-removeAll-cond:  
mset (map mset (removeAll-cond ( $\lambda b$ . mset  $b =$  mset  $a$ )  $C$ ))  
= removeAll-mset (mset  $a$ ) (mset (map mset  $C$ ))  
by (induction  $C$ ) (auto simp: ac-simps mset-less-eqI multiset-diff-union-assoc)
```

The definition and the correctness theorem are from the multiset theory `~~/src/HOL/Library/Multiset.thy`, but a name is necessary to refer to them:

```
abbreviation union-mset-list where  
union-mset-list  $xs$   $ys$   $\equiv$  case-prod append (fold ( $\lambda x$  ( $ys$ ,  $zs$ ). (remove1  $x$   $ys$ ,  $x \#$   $zs$ ))  $xs$  ( $ys$ , []))
```

```
lemma union-mset-list:  
mset  $xs \# \cup$  mset  $ys$  = mset (union-mset-list  $xs$   $ys$ )  
proof –  
have  $\bigwedge zs$ . mset (case-prod append (fold ( $\lambda x$  ( $ys$ ,  $zs$ ). (remove1  $x$   $ys$ ,  $x \#$   $zs$ ))  $xs$  ( $ys$ ,  $zs$ ))) =  
  (mset  $xs \# \cup$  mset  $ys$ ) + mset  $zs$   
  by (induct  $xs$  arbitrary:  $ys$ ) (simp-all add: multiset-eq-iff)  
then show ?thesis by simp  
qed
```

Filter

```
lemma distinct-filter-eq-if:  
distinct  $C \implies$  length (filter ( $op =$   $L$ )  $C$ ) = (if  $L \in$  set  $C$  then 1 else 0)  
by (induction  $C$ ) auto
```

end

Chapter 2

Definition of Entailment

This chapter defines various form of entailment.

end

2.1 Clausal Logic

```
theory Clausal-Logic
imports ../lib/Multiset-More
begin
```

Resolution operates of clauses, which are disjunctions of literals. The material formalized here corresponds roughly to Sections 2.1 (“Formulas and Clauses”) of Bachmair and Ganzinger, excluding the formula and term syntax.

2.1.1 Literals

Literals consist of a polarity (positive or negative) and an atom, of type *'a*.

```
datatype 'a literal =
  is-pos: Pos (atm-of: 'a)
| Neg (atm-of: 'a)
```

```
abbreviation is-neg :: 'a literal  $\Rightarrow$  bool where is-neg L  $\equiv \neg$  is-pos L
```

```
lemma Pos-atm-of-iff[simp]: Pos (atm-of L) = L  $\longleftrightarrow$  is-pos L
by auto (metis literal.disc(1))
```

```
lemma Neg-atm-of-iff[simp]: Neg (atm-of L) = L  $\longleftrightarrow$  is-neg L
by auto (metis literal.disc(2))
```

```
lemma ex-lit-cases:  $(\exists L. P L) \longleftrightarrow (\exists A. P (Pos A) \vee P (Neg A))$ 
by (metis literal.exhaust)
```

```
instantiation literal :: (type) uminus
begin
```

```
definition uminus-literal :: 'a literal  $\Rightarrow$  'a literal where
  uminus L = (if is-pos L then Neg else Pos) (atm-of L)
```

```
instance ..
```

end

lemma

uminus-Pos[simp]: $- \text{Pos } A = \text{Neg } A$ and
uminus-Neg[simp]: $- \text{Neg } A = \text{Pos } A$
unfolding *uminus-literal-def* **by** *simp-all*

lemma *atm-of-uminus[simp]:*

atm-of $(-L) = \text{atm-of } L$
by (*case-tac L, auto*)

lemma *uminus-of-uminus-id[simp]:*

$- (- (x:: 'v \text{ literal})) = x$
by (*simp add: uminus-literal-def*)

lemma *uminus-not-id[simp]:*

$x \neq - (x:: 'v \text{ literal})$
by (*case-tac x, auto*)

lemma *uminus-not-id'[simp]:*

$- x \neq (x:: 'v \text{ literal})$
by (*case-tac x, auto*)

lemma *uminus-eq-inj[iff]:*

$-(a:: 'v \text{ literal}) = -b \iff a = b$
by (*case-tac a; case-tac b*) *auto+*

lemma *uminus-lit-swap:*

$(a:: 'a \text{ literal}) = -b \iff -a = b$
by *auto*

instantiation *literal :: (preorder) preorder*

begin

definition *less-literal :: 'a literal \Rightarrow 'a literal \Rightarrow bool where*

less-literal L M $\iff \text{atm-of } L < \text{atm-of } M \vee \text{atm-of } L \leq \text{atm-of } M \wedge \text{is-neg } L < \text{is-neg } M$

definition *less-eq-literal :: 'a literal \Rightarrow 'a literal \Rightarrow bool where*

less-eq-literal L M $\iff \text{atm-of } L < \text{atm-of } M \vee \text{atm-of } L \leq \text{atm-of } M \wedge \text{is-neg } L \leq \text{is-neg } M$

instance

apply *intro-classes*

unfolding *less-literal-def less-eq-literal-def* **by** (*auto intro: order-trans simp: less-le-not-le*)

end

instantiation *literal :: (order) order*

begin

instance

apply *intro-classes*

unfolding *less-eq-literal-def* **by** (*auto intro: literal.expand*)

end

lemma *pos-less-neg[simp]: $\text{Pos } A < \text{Neg } A$*


```

unfolding less-literal-def by simp

lemma pos-less-pos-iff[simp]:  $Pos\ A < Pos\ B \longleftrightarrow A < B$ 
unfolding less-literal-def by simp

lemma pos-less-neg-iff[simp]:  $Pos\ A < Neg\ B \longleftrightarrow A \leq B$ 
unfolding less-literal-def by (auto simp: less-le-not-le)

lemma neg-less-pos-iff[simp]:  $Neg\ A < Pos\ B \longleftrightarrow A < B$ 
unfolding less-literal-def by simp

lemma neg-less-neg-iff[simp]:  $Neg\ A < Neg\ B \longleftrightarrow A < B$ 
unfolding less-literal-def by simp

lemma pos-le-neg[simp]:  $Pos\ A \leq Neg\ A$ 
unfolding less-eq-literal-def by simp

lemma pos-le-pos-iff[simp]:  $Pos\ A \leq Pos\ B \longleftrightarrow A \leq B$ 
unfolding less-eq-literal-def by (auto simp: less-le-not-le)

lemma pos-le-neg-iff[simp]:  $Pos\ A \leq Neg\ B \longleftrightarrow A \leq B$ 
unfolding less-eq-literal-def by (auto simp: less-imp-le)

lemma neg-le-pos-iff[simp]:  $Neg\ A \leq Pos\ B \longleftrightarrow A < B$ 
unfolding less-eq-literal-def by simp

lemma neg-le-neg-iff[simp]:  $Neg\ A \leq Neg\ B \longleftrightarrow A \leq B$ 
unfolding less-eq-literal-def by (auto simp: less-imp-le)

lemma leq-imp-less-eq-atm-of:  $L \leq M \implies atm-of\ L \leq atm-of\ M$ 
by (metis less-eq-literal-def less-le-not-le)

instantiation literal :: (linorder) linorder
begin

instance
  apply intro-classes
  unfolding less-eq-literal-def less-literal-def by auto

end

instantiation literal :: (wellorder) wellorder
begin

instance
proof intro-classes
  fix  $P :: 'a\ literal \Rightarrow bool$  and  $L :: 'a\ literal$ 
  assume  $ih: \bigwedge L. (\bigwedge M. M < L \implies P\ M) \implies P\ L$ 
  have  $\bigwedge x. (\bigwedge y. y < x \implies P\ (Pos\ y) \wedge P\ (Neg\ y)) \implies P\ (Pos\ x) \wedge P\ (Neg\ x)$ 
    by (rule conjI[OF ih ih])
    (auto simp: less-literal-def atm-of-def split: literal.splits intro: ih)
  hence  $\bigwedge A. P\ (Pos\ A) \wedge P\ (Neg\ A)$ 
    by (rule less-induct) blast
  thus  $P\ L$ 
    by (cases  $L$ ) simp+
qed

```

end

2.1.2 Clauses

Clauses are (finite) multisets of literals.

type-synonym 'a clause = 'a literal multiset

abbreviation poss :: 'a multiset \Rightarrow 'a clause **where** poss AA $\equiv \{\#Pos\ A.\ A \in \# AA\}$

abbreviation negs :: 'a multiset \Rightarrow 'a clause **where** negs AA $\equiv \{\#Neg\ A.\ A \in \# AA\}$

lemma image-replicate-mset[simp]: $\{\#f\ A.\ A \in \# replicate_mset\ n\ A\} = replicate_mset\ n\ (f\ A)$
by (induct n) (simp, subst replicate-mset-Suc, simp)

lemma Max-in-lits: $C \neq \{\#\} \implies Max\ (set_mset\ C) \in \# C$
by (rule Max-in[OF finite-set-mset, unfolded set-mset-eq-empty-iff])

lemma Max-atm-of-set-mset-commute: $C \neq \{\#\} \implies Max\ (atm_of\ 'set_mset\ C) = atm_of\ (Max\ (set_mset\ C))$
by (rule mono-Max-commute[symmetric])
 (auto simp: mono-def atm-of-def less-eq-literal-def less-literal-def)

lemma Max-pos-neg-less-multiset:
assumes max: $Max\ (set_mset\ C) = Pos\ A$ **and** neg: $Neg\ A \in \# D$
shows $C \# \subseteq \# D$

proof –

have $Max\ (set_mset\ C) < Neg\ A$

using max **by** simp

thus ?thesis

using neg **by** (metis (no-types) ex-gt-imp-less-multiset Max-less-iff[OF finite-set-mset]
 all-not-in-conv)

qed

lemma pos-Max-imp-neg-notin: $Max\ (set_mset\ C) = Pos\ A \implies Neg\ A \notin \# C$
using Max-pos-neg-less-multiset[unfolded multiset-linorder.not-le[symmetric]] **by** blast

lemma less-eq-Max-lit: $C \neq \{\#\} \implies C \# \subseteq \# D \implies Max\ (set_mset\ C) \leq Max\ (set_mset\ D)$

proof (unfold le-multiset_{HO})

assume ne: $C \neq \{\#\}$ **and** ex-gt: $\forall x. count\ D\ x < count\ C\ x \longrightarrow (\exists y > x. count\ C\ y < count\ D\ y)$

from ne **have** $Max\ (set_mset\ C) \in \# C$

by (fast intro: Max-in-lits)

hence $\exists l. l \in \# D \wedge \neg l < Max\ (set_mset\ C)$

using ex-gt **by** (metis count-greater-zero-iff count-inI less-not-sym)

hence $\neg Max\ (set_mset\ D) < Max\ (set_mset\ C)$

by (metis Max.coboundedI[OF finite-set-mset] le-less-trans)

thus ?thesis

by simp

qed

definition atms-of :: 'a clause \Rightarrow 'a set **where**
 atms-of C = atm-of 'set-mset C

lemma atms-of-empty[simp]: $atms_of\ \{\#\} = \{\}$
unfolding atms-of-def **by** simp

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

lemma *atms-of-union-mset[simp]*:
 $\text{atms-of } (A \# \cup B) = \text{atms-of } A \cup \text{atms-of } B$
unfolding *atms-of-def* **by** (*auto simp: max-def split: if-split-asm*)

lemma *finite-atms-of[iff]*: *finite* ($\text{atms-of } C$)
unfolding *atms-of-def* **by** *simp*

lemma *atm-of-lit-in-atms-of*: $L \in \# C \implies \text{atm-of } L \in \text{atms-of } C$
unfolding *atms-of-def* **by** *simp*

lemma *atms-of-plus[simp]*: $\text{atms-of } (C + D) = \text{atms-of } C \cup \text{atms-of } D$
unfolding *atms-of-def image-def* **by** *auto*

lemma *pos-lit-in-atms-of*: $\text{Pos } A \in \# C \implies A \in \text{atms-of } C$
unfolding *atms-of-def* **by** (*metis image-iff literal.sel(1)*)

lemma *neg-lit-in-atms-of*: $\text{Neg } A \in \# C \implies A \in \text{atms-of } C$
unfolding *atms-of-def* **by** (*metis image-iff literal.sel(2)*)

lemma *atm-imp-pos-or-neg-lit*: $A \in \text{atms-of } C \implies \text{Pos } A \in \# C \vee \text{Neg } A \in \# C$
unfolding *atms-of-def image-def mem-Collect-eq*
by (*metis Neg-atm-of-iff Pos-atm-of-iff*)

lemma *atm-iff-pos-or-neg-lit*: $A \in \text{atms-of } L \longleftrightarrow \text{Pos } A \in \# L \vee \text{Neg } A \in \# L$
by (*auto intro: pos-lit-in-atms-of neg-lit-in-atms-of dest: atm-imp-pos-or-neg-lit*)

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

lemma *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*:
 $\text{atm-of } L \in \text{atm-of } I \longleftrightarrow (L \in I \vee -L \in I)$
by (*auto intro: rev-image-eqI simp: atm-of-eq-atm-of*)

lemma *lits-subseteq-imp-atms-subseteq*: $\text{set-mset } C \subseteq \text{set-mset } D \implies \text{atms-of } C \subseteq \text{atms-of } D$
unfolding *atms-of-def* **by** *blast*

lemma *atms-empty-iff-empty[iff]*: $\text{atms-of } C = \{\} \longleftrightarrow C = \{\#\}$
unfolding *atms-of-def image-def Collect-empty-eq*
by (*metis all-not-in-conv set-mset-eq-empty-iff*)

lemma
 $\text{atms-of-poss[simp]}$: $\text{atms-of } (\text{poss } AA) = \text{set-mset } AA$ **and**
 $\text{atms-of-negg[simp]}$: $\text{atms-of } (\text{negs } AA) = \text{set-mset } AA$
unfolding *atms-of-def image-def* **by** *auto*

lemma *less-eq-Max-atms-of*: $C \neq \{\#\} \implies C \# \subseteq \# D \implies \text{Max } (\text{atms-of } C) \leq \text{Max } (\text{atms-of } D)$
unfolding *atms-of-def*
by (*metis Max-atm-of-set-mset-commute le-multiset-empty-right leq-imp-less-eq-atm-of less-eq-Max-lit*)

lemma *le-multiset-Max-in-imp-Max*:
 $\text{Max } (\text{atms-of } D) = A \implies C \# \subseteq \# D \implies A \in \text{atms-of } C \implies \text{Max } (\text{atms-of } C) = A$

by (*metis* *Max.coboundedI*[*OF finite-atms-of*] *atms-of-def empty-iff eq-iff image-subsetI*
less-eq-Max-atms-of set-mset-empty subset-Compl-self-eq)

lemma *atm-of-Max-lit[simp]*: $C \neq \{\#\} \implies \text{atm-of } (\text{Max } (\text{set-mset } C)) = \text{Max } (\text{atms-of } C)$
unfolding *atms-of-def Max-atm-of-set-mset-commute ..*

lemma *Max-lit-eq-pos-or-neg-Max-atm*:

$C \neq \{\#\} \implies \text{Max } (\text{set-mset } C) = \text{Pos } (\text{Max } (\text{atms-of } C)) \vee \text{Max } (\text{set-mset } C) = \text{Neg } (\text{Max } (\text{atms-of } C))$

by (*metis* *Neg-atm-of-iff Pos-atm-of-iff atm-of-Max-lit*)

lemma *atms-less-imp-lit-less-pos*: $(\bigwedge B. B \in \text{atms-of } C \implies B < A) \implies L \in \# C \implies L < \text{Pos } A$
unfolding *atms-of-def less-literal-def* **by** *force*

lemma *atms-less-eq-imp-lit-less-eq-neg*: $(\bigwedge B. B \in \text{atms-of } C \implies B \leq A) \implies L \in \# C \implies L \leq \text{Neg } A$
unfolding *less-eq-literal-def* **by** (*simp add: atm-of-lit-in-atms-of*)

end

2.2 Herbrand Intepretation

theory *Herbrand-Interpretation*

imports *Clausal-Logic*

begin

Resolution operates of clauses, which are disjunctions of literals. The material formalized here corresponds roughly to Sections 2.2 (“Herbrand Interpretations”) of Bachmair and Ganzinger, excluding the formula and term syntax.

2.2.1 Herbrand Interpretations

A Herbrand interpretation is a set of ground atoms that are to be considered true.

type-synonym *'a interp* = *'a set*

definition *true-lit* :: *'a interp* \Rightarrow *'a literal* \Rightarrow *bool* (**infix** \models_l 50) **where**
 $I \models_l L \longleftrightarrow (\text{if is-pos } L \text{ then } (\lambda P. P) \text{ else Not } (\text{atm-of } L \in I))$

lemma *true-lit-simps[simp]*:

$I \models_l \text{Pos } A \longleftrightarrow A \in I$

$I \models_l \text{Neg } A \longleftrightarrow A \notin I$

unfolding *true-lit-def* **by** *auto*

lemma *true-lit-iff[iff]*: $I \models_l L \longleftrightarrow (\exists A. L = \text{Pos } A \wedge A \in I \vee L = \text{Neg } A \wedge A \notin I)$
by (*cases L*) *simp+*

definition *true-cls* :: *'a interp* \Rightarrow *'a clause* \Rightarrow *bool* (**infix** \models 50) **where**
 $I \models C \longleftrightarrow (\exists L. L \in \# C \wedge I \models_l L)$

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

unfolding *true-cls-def* **by** *simp*

lemma *true-cls-singleton[iff]*: $I \models \{\#L\# \} \longleftrightarrow I \models_l L$

unfolding *true-cls-def* **by** *simp*

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*: $\text{set-mset } C \subseteq \text{set-mset } D \implies I \models C \implies I \models D$
unfolding *true-cls-def* *subset-eq* **by** *metis*

lemma
assumes $I \subseteq J$
shows
false-to-true-imp-ex-pos: $\neg I \models C \implies J \models C \implies \exists A \in J. \text{Pos } A \in \# C$ **and**
true-to-false-imp-ex-neg: $I \models C \implies \neg J \models C \implies \exists A \in J. \text{Neg } A \in \# C$
using *assms* **unfolding** *subset-iff* *true-cls-def*
by (*metis literal.collapse true-lit-simps*)**+**

lemma *true-cls-replicate-mset*[*iff*]: $I \models \text{replicate-mset } n \ L \longleftrightarrow n \neq 0 \wedge I \models_l L$
by (*induct n*) *auto*

lemma *pos-literal-in-imp-true-cls*[*intro*]: $\text{Pos } A \in \# C \implies A \in I \implies I \models C$
by (*metis true-cls-def true-lit-simps*(1))

lemma *neg-literal-notin-imp-true-cls*[*intro*]: $\text{Neg } A \in \# C \implies A \notin I \implies I \models C$
by (*metis true-cls-def true-lit-simps*(2))

lemma *pos-neg-in-imp-true*: $\text{Pos } A \in \# C \implies \text{Neg } A \in \# C \implies I \models C$
unfolding *true-cls-def* **by** (*metis true-lit-simps*)

definition *true-clss* :: 'a *interp* \Rightarrow 'a *clause set* \Rightarrow *bool* (**infix** \models_s 50) **where**
 $I \models_s CC \longleftrightarrow (\forall C \in CC. I \models C)$

lemma *true-clss-empty*[*iff*]: $I \models_s \{\}$
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-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-mono*: $DD \subseteq CC \implies I \models_s CC \implies I \models_s DD$
unfolding *true-clss-def* **by** *blast*

abbreviation *satisfiable* :: 'a *clause set* \Rightarrow *bool* **where**
satisfiable $CC \equiv \exists I. I \models_s CC$

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

lemma *true-cls-mset-empty*[*iff*]: $I \models_m \{\#\}$
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*

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** *auto*

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

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

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

end

2.3 Partial Clausal Logic

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

We define here entailment by a set of literals. This is *not* an Herbrand interpretation and has different properties. One key difference is that such a set can be inconsistent (i.e. containing both L and $-L$).

Satisfiability is defined by the existence of a total and consistent model.

2.3.1 Clauses

Clauses are (finite) multisets of literals.

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

2.3.2 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*]

Consistency

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

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

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

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

lemma *consistent-interp-change-insert*:

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

Atoms

We define here various lifting of *atm-of* (applied to a single literal) to set and multisets of literals.

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

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

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

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

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

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

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

atms-of-s $C = \text{atm-of } ' C$

lemma *atms-of-ms-empty-set[simp]*:

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

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

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

lemma *atms-of-ms-mono*:

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

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

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

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

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

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

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

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

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

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

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

proof

assume *?P*

then show *?Q* **unfolding** *atms-of-s-def* **by** (*metis image-iff literal.exhaust-sel*)

next

assume *?Q*

then show *?P* **unfolding** *atms-of-s-def* **by** *force*

qed

lemma *atm-of-in-atm-of-set-in-uminus*:

$\text{atm-of } L' \in \text{atm-of } 'B \implies L' \in B \vee - L' \in B$

using *atms-of-s-def* **by** (*cases L'*) *fastforce+*

Totality

definition *total-over-set* :: *'a interp* \Rightarrow *'a set* \Rightarrow *bool* **where**

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

definition *total-over-m* :: *'a literal set* \Rightarrow *'a clause set* \Rightarrow *bool* **where**

total-over-m I ψ s = *total-over-set I* (*atms-of-ms ψ s*)

lemma *total-over-set-empty*[*simp*]:

total-over-set I $\{\}$

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

lemma *total-over-m-empty*[*simp*]:

total-over-m I $\{\}$

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

lemma *total-over-set-single*[*iff*]:

$\text{total-over-set } I \{L\} \longleftrightarrow (\text{Pos } L \in I \vee \text{Neg } L \in I)$

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

lemma *total-over-set-insert*[*iff*]:

$\text{total-over-set } I (\text{insert } L \text{ } Ls) \longleftrightarrow ((\text{Pos } L \in I \vee \text{Neg } L \in I) \wedge \text{total-over-set } I Ls)$

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

lemma *total-over-set-union*[*iff*]:

$\text{total-over-set } I (Ls \cup Ls') \longleftrightarrow (\text{total-over-set } I Ls \wedge \text{total-over-set } I Ls')$

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 $\text{total-over-m } I \{C + D\} \longleftrightarrow (\text{total-over-m } I \{C\} \wedge \text{total-over-m } I \{D\})$

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

lemma *total-over-m-union*[*iff*]:

$\text{total-over-m } I (A \cup B) \longleftrightarrow (\text{total-over-m } I A \wedge \text{total-over-m } I B)$

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

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

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

proof –

let $?I' = \{Pos\ v \mid v. v \in atms-of-ms\ B \wedge v \notin atms-of-ms\ A\}$
have $\forall x \in ?I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A$ **by** *auto*
moreover have *total-over-m (I \cup ?I') (A \cup B)*
using *total unfolding total-over-m-def total-over-set-def* **by** *auto*
ultimately show *?thesis* **by** *blast*

qed

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

proof –

let $?I' = \{Pos\ v \mid v. v \in atms-of-ms\ B \wedge v \notin atms-of-ms\ A \wedge Pos\ v \notin I \wedge Neg\ v \notin I\}$
have $\forall x \in ?I'. atm-of\ x \in atms-of-ms\ B \wedge atm-of\ x \notin atms-of-ms\ A$ **by** *auto*
moreover have *total-over-m (I \cup ?I') (A \cup B)*
using *total unfolding total-over-m-def total-over-set-def* **by** *auto*
moreover have *consistent-interp (I \cup ?I')*
using *cons unfolding consistent-interp-def* **by** *(intro allI) (rename-tac L, case-tac L, auto)*
ultimately show *?thesis* **by** *blast*

qed

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

lemma *total-over-set-literal-defined*:
assumes $\{\#A\# \} + D \in \psi s$
and *total-over-set I (atms-of-ms ψs)*
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 $L: L \notin \# \psi \ -L \notin \# \psi$
shows *total-over-m I { ψ }*
unfolding *total-over-m-def total-over-set-def*

proof

fix l
assume $l: l \in atms-of-ms\ \{\psi\}$
then have $Pos\ l \in I \vee Neg\ l \in I \vee l = atm-of\ L$
using *assms unfolding total-over-m-def total-over-set-def* **by** *auto*
moreover have $atm-of\ L \notin atms-of-ms\ \{\psi\}$

```

proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then have  $atm\text{-}of\ L \in atm\text{-}of\ \psi$  by auto
  then have  $Pos\ (atm\text{-}of\ L) \in\# \psi \vee Neg\ (atm\text{-}of\ L) \in\# \psi$ 
    using  $atm\text{-}imp\text{-}pos\text{-}or\text{-}neg\text{-}lit$  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  $Pos\ l \in I \vee Neg\ l \in I$  using l by metis
qed

```

```

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

```

```

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

```

Interpretations

definition true-cls :: 'a interp \Rightarrow 'a clause \Rightarrow bool (infix \models 50) **where**
 $I \models C \longleftrightarrow (\exists L \in\# C. I \models_l L)$

```

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

```

```

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

```

```

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

```

```

lemma true-cls-mono-set-mset: set-mset C  $\subseteq$  set-mset D  $\Longrightarrow I \models C \Longrightarrow I \models D$ 
  unfolding true-cls-def subset-eq Bex-def by metis

```

```

lemma true-cls-mono-leD[dest]: A  $\subseteq\# B \Longrightarrow I \models A \Longrightarrow I \models B$ 
  unfolding true-cls-def by auto

```

```

lemma
  assumes I  $\models \psi$ 
  shows
    true-cls-union-increase[simp]: I  $\cup$  I'  $\models \psi$  and
    true-cls-union-increase'[simp]: I'  $\cup$  I  $\models \psi$ 
  using assms unfolding true-cls-def by auto

```

```

lemma true-cls-mono-set-mset-l:
  assumes A  $\models \psi$ 
  and A  $\subseteq B$ 
  shows B  $\models \psi$ 
  using assms unfolding true-cls-def by auto

```

```

lemma true-cls-replicate-mset[iff]: I  $\models replicate\text{-}mset\ n\ L \longleftrightarrow n \neq 0 \wedge I \models_l L$ 

```

by (induct n) auto

lemma *true-cls-empty-entails*[iff]: $\neg \{\} \models N$
 by (auto simp add: true-cls-def)

lemma *true-cls-not-in-remove*:
 assumes $L \notin \# \chi$ and $I \cup \{L\} \models \chi$
 shows $I \models \chi$
 using *assms* **unfolding** *true-cls-def* **by** *auto*

definition *true-clss* :: '*a* interp \Rightarrow '*a* clauses \Rightarrow bool (infix \models_s 50) **where**
 $I \models_s CC \longleftrightarrow (\forall C \in CC. I \models C)$

lemma *true-clss-empty*[simp]: $I \models_s \{\}$
unfolding *true-clss-def* **by** *blast*

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

lemma *true-clss-empty-entails-empty*[iff]: $\{\} \models_s N \longleftrightarrow N = \{\}$
unfolding *true-clss-def* **by** (auto simp add: true-cls-def)

lemma *true-cls-insert-l* [simp]:
 $M \models A \implies \text{insert } L \ M \models A$
unfolding *true-cls-def* **by** *auto*

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

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

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

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

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

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

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

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

lemma *notin-vars-union-true-cls-true-cls*:

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

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

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

using *I-CC in-implies-atm-of-on-atms-of-ms unfolding true-clss-def Ball-def true-cls-def Bex-def true-lit-def*
by *blast*

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

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

moreover have *total-over-m* $?I \text{ } CC$

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

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

lemma *satisfiable-carac[iff]*:
 $(\exists I. \text{consistent-interp } I \wedge I \models_s \varphi) \longleftrightarrow \text{satisfiable } \varphi$ (**is** $(\exists I. ?Q I) \longleftrightarrow ?S$)
proof
 assume *?S*
 then show $\exists I. ?Q I$ **unfolding** *satisfiable-def* by *auto*
next
 assume $\exists I. ?Q I$
 then obtain *I* **where** *cons*: *consistent-interp I* **and** *I*: $I \models_s \varphi$ **by** *metis*
 let *?I' = {Pos v | v. v \notin atms-of-s I \wedge v \in atms-of-ms φ }*
 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') φ*
 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*

Entailment for Multisets of Clauses

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

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

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

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

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

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

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

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

theorem *true-cls-remove-unused*:

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

qed

lemma *multiset-not-empty*:

assumes $M \neq \{\#\}$
and $x \in\# M$
shows $\exists A. x = \text{Pos } A \vee x = \text{Neg } A$
using *assms literal.exhaust-sel* **by** *blast*

lemma *atms-of-ms-empty*:

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

lemma *consistent-interp-disjoint*:

assumes *consI*: *consistent-interp I*
and *disj*: $\text{atms-of-s } A \cap \text{atms-of-s } I = \{\}$
and *consA*: *consistent-interp A*
shows *consistent-interp* $(A \cup I)$

proof (*rule ccontr*)

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

```

  using disj unfolding atms-of-s-def by (auto simp add: rev-image-eqI)
ultimately show False
  using consA consI unfolding consistent-interp-def by (metis (full-types) Un-iff
    literal.exhaust-sel uminus-Neg uminus-Pos)
qed

```

```

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

```

```

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

```

```

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

```

Tautologies

We define tautologies as clauses entailed by every total model and show later that is equivalent to containing a literal and its negation.

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

```

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

```

```

lemma tautology-minus[simp]:
  assumes L  $\in\# A$  and  $\neg L \in\# A$ 
  shows tautology A
  by (metis assms literal.exhaust tautology-Pos-Neg uminus-Neg uminus-Pos)

```

```

lemma tautology-exists-Pos-Neg:
  assumes tautology  $\psi$ 
  shows  $\exists p. \text{Pos } p \in\# \psi \wedge \text{Neg } p \in\# \psi$ 
proof (rule ccontr)
  assume p:  $\neg (\exists p. \text{Pos } p \in\# \psi \wedge \text{Neg } p \in\# \psi)$ 
  let ?I =  $\{-L \mid L. L \in\# \psi\}$ 
  have total-over-set ?I (atms-of  $\psi$ )
    unfolding total-over-set-def using atm-imp-pos-or-neg-lit by force
  moreover have  $\neg ?I \models \psi$ 
    unfolding true-cls-def true-lit-def Bex-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:

```


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

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

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

lemma *minus-interp-tautology:*
assumes $\{-L \mid L. L \in \# \chi\} \models \chi$
shows *tautology* χ
proof –
obtain L **where** $L \in \# \chi \wedge -L \in \# \chi$
using *assms* **unfolding** *true-cls-def* **by** *auto*
then show *?thesis* **using** *tautology-decomp literal.exhaust uminus-Neg uminus-Pos* **by** *metis*
qed

lemma *remove-literal-in-model-tautology:*
assumes $I \cup \{\text{Pos } P\} \models \varphi$
and $I \cup \{\text{Neg } P\} \models \varphi$
shows $I \models \varphi \vee \text{tautology } \varphi$
using *assms* **unfolding** *true-cls-def* **by** *auto*

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

Entailment for clauses and propositions

We also need entailment of clauses by other clauses.

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

lemma true-clss-clss-union-and[iff]:
   $A \models_{ps} C \cup D \longleftrightarrow (A \models_{ps} C \wedge A \models_{ps} D)$ 
proof
{
  fix  $A \ C \ D :: 'a \ \text{clauses}$ 
  assume  $A: A \models_{ps} C \cup D$ 
  have  $A \models_{ps} C$ 
    unfolding true-clss-clss-def true-clss-clss-def insert-def total-over-m-insert
    proof (intro allI impI)
      fix  $I$ 
      assume
         $\text{tot}AC: \text{total-over-m } I \ (A \cup C) \ \text{and}$ 
         $\text{cons}: \text{consistent-interp } I \ \text{and}$ 
         $I: I \models_s A$ 
      then have  $\text{tot}: \text{total-over-m } I \ A \ \text{and } \text{tot}': \text{total-over-m } I \ C \ \text{by } \text{auto}$ 
      obtain  $I'$  where
         $\text{tot}': \text{total-over-m } (I \cup I') \ (A \cup C \cup D) \ \text{and}$ 
         $\text{cons}': \text{consistent-interp } (I \cup I') \ \text{and}$ 
         $H: \forall x \in I'. \text{atm-of } x \in \text{atms-of-ms } D \wedge \text{atm-of } x \notin \text{atms-of-ms } (A \cup C)$ 
        using total-over-m-consistent-extension[OF - cons, of A C] tot tot' by blast
      moreover have  $I \cup I' \models_s A \ \text{using } I \ \text{by } \text{simp}$ 
      ultimately have  $I \cup I' \models_s C \cup D \ \text{using } A \ \text{unfolding } \text{true-clss-clss-def} \ \text{by } \text{auto}$ 
      then have  $I \cup I' \models_s C \cup D \ \text{by } \text{auto}$ 
      then show  $I \models_s C \ \text{using } \text{notin-vars-union-true-clss-true-clss}[of I'] \ H \ \text{by } \text{auto}$ 
    qed
  } note  $H = \text{this}$ 
  assume  $A \models_{ps} C \cup D$ 
  then show  $A \models_{ps} C \wedge A \models_{ps} D \ \text{using } H[of A] \ \text{Un-commute}[of C D] \ \text{by } \text{metis}$ 
next
  assume  $A \models_{ps} C \wedge A \models_{ps} D$ 
  then show  $A \models_{ps} C \cup D$ 
    unfolding true-clss-clss-def by auto
qed

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

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

```

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

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

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

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

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

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

lemma *true-clss-clss-generalise-true-clss-clss*:
 $A \cup C \models_{ps} D \implies B \models_{ps} C \implies A \cup B \models_{ps} D$
proof –
assume $a1: A \cup C \models_{ps} D$
assume $B \models_{ps} C$
then have $f2: \bigwedge M. M \cup B \models_{ps} C$
by (*meson true-clss-clss-union-l-r*)
have $\bigwedge M. C \cup (M \cup A) \models_{ps} D$
using $a1$ **by** (*simp add: Un-commute sup-left-commute*)
then show *?thesis*
using $f2$ **by** (*metis (no-types) Un-commute true-clss-clss-left-right true-clss-clss-union-and*)
qed

lemma *true-clss-clss-or-true-clss-clss-or-not-true-clss-clss-or*:
assumes $D: N \models_p D + \{\#- L\# \}$
and $C: N \models_p C + \{\#L\# \}$
shows $N \models_p D + C$
unfolding *true-clss-clss-def*
proof (*intro allI impI*)
fix I
assume
 $tot: total-over-m\ I\ (N \cup \{D + C\})$ **and**
 $consistent-interp\ I$ **and**
 $I \models_s N$
{
assume $L: L \in I \vee -L \in I$
then have $total-over-m\ I\ \{D + \{\#- L\# \}\}$
using tot **by** (*cases L*) *auto*
then have $I \models D + \{\#- L\# \}$ **using** $D \langle I \models_s N \rangle tot \langle consistent-interp\ I \rangle$
unfolding *true-clss-clss-def* **by** *auto*

```

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

```

lemma *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  $D \langle I \models_s N \rangle \text{ tot} \langle \text{consistent-interp } I \rangle$  unfolding true-clss-cls-def by auto
    moreover
      have total-over-m  $I \{C + \{\#L\#\}\}$ 
        using  $L \text{ tot}$  by (cases  $L$ ) auto
      then have  $I \models C + \{\#L\#\}$ 
        using  $C \langle I \models_s N \rangle \text{ tot} \langle \text{consistent-interp } I \rangle$  unfolding true-clss-cls-def by auto
      ultimately have  $I \models D \# \cup C$  using  $\langle \text{consistent-interp } I \rangle$  unfolding consistent-interp-def
        by auto
    }
  }

```

```

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

```

2.3.3 Subsumptions

lemma *subsumption-total-over-m*:

```

assumes  $A \subseteq \# B$ 
shows total-over-m  $I \{B\} \implies \text{total-over-m } I \{A\}$ 
using assms unfolding subset-mset-def total-over-m-def total-over-set-def
by (auto simp add: mset-le-exists-conv)

```

lemma *atms-of-replicate-mset-replicate-mset-uminus[simp]*:

```

 $\text{atms-of } (D - \text{replicate-mset } (\text{count } D \ L) \ L - \text{replicate-mset } (\text{count } D \ (-L)) \ (-L))$ 
 $= \text{atms-of } D - \{\text{atm-of } L\}$ 
by (fastforce simp: atm-of-eq-atm-of atms-of-def)

```

lemma *subsumption-chained*:

```

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

```

proof (*induct card* $\{Pos \ v \mid v. v \in \text{atms-of } D \wedge v \notin \text{atms-of } C\}$ *arbitrary*: D
rule: nat-less-induct-case)

```

case 0 note  $n = \text{this}(1)$  and  $H = \text{this}(2)$  and  $\text{incl} = \text{this}(3)$ 
then have  $\text{atms-of } D \subseteq \text{atms-of } C$  by auto
then have  $\forall I. \text{total-over-m } I \{C\} \longrightarrow \text{total-over-m } I \{D\}$ 
  unfolding total-over-m-def total-over-set-def by auto
moreover have  $\forall I. I \models C \longrightarrow I \models D$  using  $\text{incl}$  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 ∪ {L}) {D}
    unfolding total-over-m-def total-over-set-def using atms-of-D by auto

  assume IDL: I ⊨ ?D'
  then have I ∪ {L} ⊨ D unfolding true-cls-def by force
  then have I ∪ {L} ⊨ φ using H tot by auto

  moreover
    have tot': total-over-m (I ∪ {-L}) {D}
      using tot unfolding total-over-m-def total-over-set-def by auto
    have I ∪ {-L} ⊨ D using IDL unfolding true-cls-def by force
    then have I ∪ {-L} ⊨ φ using H tot' by auto
  ultimately have I ⊨ φ ∨ tautology φ
    using L remove-literal-in-model-tautology by force
} note H' = this

have L ∉# C and -L ∉# C using L atm-iff-pos-or-neg-lit by force+
then have C-in-D': C ⊆# ?D' using ⟨C ⊆# D⟩ by (auto simp: subseq-mset-def not-in-iff)
have card {Pos v | v. v ∈ atms-of ?D' ∧ v ∉ atms-of C} <
  card {Pos v | v. v ∈ atms-of D ∧ v ∉ atms-of C}
  using L by (auto intro!: psubset-card-mono)
then show ?case
  using IH C-in-D' H' unfolding card[symmetric] by blast
qed

```

2.3.4 Removing Duplicates

```

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

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

lemma true-cls-remdups-mset[iff]: I ⊨ remdups-mset C ⟷ I ⊨ C
  unfolding true-cls-def by auto

lemma true-clss-cls-remdups-mset[iff]: A ⊨p remdups-mset C ⟷ A ⊨p C
  unfolding true-clss-cls-def total-over-m-def by auto

```

2.3.5 Set of all Simple Clauses

A simple clause with respect to a set of atoms is such that

1. its atoms are included in the considered set of atoms;
2. it is not a tautology;
3. it does not contains duplicate literals.

It corresponds to the clauses that cannot be simplified away in a calculus without considering the other 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*) =
 $(\text{op} + \{\#\text{Pos } l\}) \text{ ' } (\text{simple-clss } \text{atms})$
 $\cup (\text{op} + \{\#\text{Neg } l\}) \text{ ' } (\text{simple-clss } \text{atms})$
 $\cup \text{simple-clss } \text{atms}(\text{is } ?I = ?U)$

proof (*standard*; *standard*)

fix C

assume $C \in ?I$

then have

atms: $\text{atms-of } C \subseteq \text{insert } l \text{ atms}$ **and**

taut: $\neg \text{tautology } C$ **and**

dist: *distinct-mset* C

unfolding *simple-clss-def* **by** *auto*

have $H: \bigwedge x. x \in \# C \implies \text{atm-of } x \in \text{insert } l \text{ atms}$

using *atm-of-lit-in-atms-of atms* **by** *blast*

consider

(*Add*) $L \text{ where } L \in \# C \text{ and } L = \text{Neg } l \vee L = \text{Pos } l$

| (*No*) $\text{Pos } l \notin \# C \text{ Neg } l \notin \# C$

by *auto*

then show $C \in ?U$

proof *cases*

case *Add*

then have $LCL: L \notin \# C - \{\#L\}$

using *dist* **unfolding** *distinct-mset-def* **by** (*auto simp: not-in-iff*)

have $LC: -L \notin \# C$

using *taut Add* **by** *auto*

obtain $aa :: 'a$ **where**

$f4: (aa \in \text{atms-of } (\text{remove1-mset } L C) \longrightarrow aa \in \text{atms}) \longrightarrow \text{atms-of } (\text{remove1-mset } L C) \subseteq \text{atms}$
by (*meson subset-iff*)

obtain $ll :: 'a$ *literal* **where**

$aa \notin \text{atm-of ' set-mset } (\text{remove1-mset } L C) \vee aa = \text{atm-of } ll \wedge ll \in \# \text{remove1-mset } L C$

by *blast*

then have $\text{atms-of } (C - \{\#L\}) \subseteq \text{atms}$

using $f4$ *Add LCL LC H* **unfolding** *atms-of-def* **by** (*metis H in-diffD insertE*

literal.exhaust-sel uminus-Neg uminus-Pos)

moreover have $\neg \text{tautology } (C - \{\#L\})$

using *taut* **by** (*metis Add(1) insert-DiffM tautology-add-single*)

moreover have *distinct-mset* $(C - \{\#L\})$

using *dist* **by** *auto*

ultimately have $(C - \{\#L\}) \in \text{simple-clss } \text{atms}$

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 $C \in \text{simple-clss } \text{atms}$

using *taut atms dist* **unfolding** *simple-clss-def*


```

    by (auto simp: atm-iff-pos-or-neg-lit split: if-split-asm dest!: H)
  then show ?thesis by blast
qed
next
fix C
assume C ∈ ?U
then consider
  (Add) L C' where C = {#L#} + C' and C' ∈ simple-clss atms and
    L = Pos l ∨ L = Neg l
  | (No) C ∈ simple-clss atms
by auto
then show C ∈ ?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' ⊆ atms and
    taut: ¬tautology C' and
    dist: distinct-mset C'
    unfolding simple-clss-def by auto
  have atms-of C ⊆ insert l atms
    using atms C' L by auto
  moreover have ¬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 ∈ simple-clss atms
  shows atms-of x ⊆ atms ∧ ¬tautology x ∧ distinct-mset x
  using assms unfolding simple-clss-def by auto

```

```

lemma cls-in-simple-clss:
  shows {#} ∈ 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) ≤ (3::nat) ^ (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:
   $\bigwedge C'. \{\#Pos\ l\#\} + C' \notin \text{simple-clss } C$ 
   $\bigwedge C'. \{\#Neg\ l\#\} + C' \notin \text{simple-clss } C$ 
  using l unfolding simple-clss-def by auto
have H:  $\bigwedge C' D. \{\#Pos\ l\#\} + C' = \{\#Neg\ l\#\} + D \implies D \in \text{simple-clss } C \implies \text{False}$ 
  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 +  $\{\#Pos\ l\#\}$ ) ' (simple-clss C)
let ?N = (op +  $\{\#Neg\ l\#\}$ ) ' (simple-clss C)
let ?O = simple-clss C
have card (?P  $\cup$  ?N  $\cup$  ?O) = card (?P  $\cup$  ?N) + card ?O
  apply (subst card-Un-disjoint)
  using l fin by (auto simp: simple-clss-finite notin)
moreover have card (?P  $\cup$  ?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  $(3::nat) \wedge \text{card (insert l C)} = 3 \wedge (\text{card } C) + 3 \wedge (\text{card } C) + 3 \wedge (\text{card } C)$ 
  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:  $\text{atms} \subseteq \text{atms}'$ 
  shows  $\text{simple-clss } \text{atms} \subseteq \text{simple-clss } \text{atms}'$ 
  using assms unfolding simple-clss-def by auto

```

```

lemma distinct-mset-not-tautology-implies-in-simple-clss:
  assumes distinct-mset  $\chi$  and  $\neg \text{tautology } \chi$ 
  shows  $\chi \in \text{simple-clss (atms-of } \chi)$ 
  using assms unfolding simple-clss-def by auto

```

```

lemma simplified-in-simple-clss:
  assumes distinct-mset-set  $\psi$  and  $\forall \chi \in \psi. \neg \text{tautology } \chi$ 
  shows  $\psi \subseteq \text{simple-clss (atms-of-ms } \psi)$ 
  using assms unfolding simple-clss-def
  by (auto simp: distinct-mset-set-def atms-of-ms-def)

```

2.3.6 Experiment: Expressing the Entailments as Locales

```

locale entail =
  fixes entail :: 'a set  $\Rightarrow$  'b  $\Rightarrow$  bool (infix  $\models_e$  50)
  assumes entail-insert[simp]:  $I \neq \{\} \implies \text{insert } L \ I \models_e x \longleftrightarrow \{L\} \models_e x \vee I \models_e x$ 
  assumes entail-union[simp]:  $I \models_e A \implies I \cup I' \models_e A$ 
begin

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

lemma entails-empty[simp]:
   $I \models_{es} \{\}$ 
  unfolding entails-def by auto

lemma entails-single[iff]:
   $I \models_{es} \{a\} \longleftrightarrow I \models_e a$ 
  unfolding entails-def by auto

lemma entails-insert-l[simp]:
   $M \models_{es} A \implies \text{insert } L \ M \models_{es} A$ 
  unfolding entails-def by (metis Un-commute entail-union insert-is-Un)

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

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

lemma entails-insert-mono:  $DD \subseteq CC \implies I \models_{es} CC \implies I \models_{es} DD$ 
  unfolding entails-def by blast

lemma entails-union-increase[simp]:
  assumes  $I \models_{es} \psi$ 
  shows  $I \cup I' \models_{es} \psi$ 
  using assms unfolding entails-def by auto

lemma true-clss-commute-l:
   $I \cup I' \models_{es} \psi \longleftrightarrow I' \cup I \models_{es} \psi$ 
  by (simp add: Un-commute)

lemma entails-remove[simp]:  $I \models_{es} N \implies I \models_{es} \text{Set.remove } a \ N$ 
  by (simp add: entails-def)

lemma entails-remove-minus[simp]:  $I \models_{es} N \implies I \models_{es} N - A$ 
  by (simp add: entails-def)

end

interpretation true-cl: entail true-cl
  by standard (auto simp add: true-cl-def)

```

2.3.7 Entailment to be extended

In some cases we want a more general version of entailment to have for example $\{\} \models \{\#L, -L\# \}$. This is useful when the model we are building might not be total (the literal L might

have been definitely removed from the set of clauses), but we still want to have a property of entailment considering that theses removed literals are not important.

We can given a model I consider all the natural extensions: C is entailed by an extended I , if for all total extension of I , this model entails C .

definition *true-clss-ext* :: 'a literal set \Rightarrow 'a literal multiset set \Rightarrow bool (**infix** \models_{sext} 49)

where

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

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

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

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

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

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

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

unfolding *true-clss-ext-def*

proof (*intro allI impI*)

fix J

assume

$I \subseteq J$ **and**

cons: *consistent-interp* J **and**

tot: *total-over-m* J $(N - \{C\})$

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

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* $\in N - \{C\}$)

apply *auto* \square

using *atms-of-s-def atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*

by (*fastforce simp: atms-of-def*)

ultimately have $?J \models_s N$

using *assms* **unfolding** *true-clss-ext-def* **by** *blast*

then have $?J \models_s N - \{C\}$ **by** *auto*

have $\{v \in ?J. atm\text{-}of \ v \in atms\text{-}of\text{-}ms \ (N - \{C\})\} \subseteq J$

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

by (*auto intro!: rev-image-eqI*)

then show $J \models_s N - \{C\}$

using *true-clss-remove-unused* [*OF* $\langle ?J \models_s N - \{C\} \rangle$] **unfolding** *true-clss-def*

by (*meson true-clss-mono-set-mset-l*)

qed

lemma *consistent-true-clss-ext-satisfiable*:

assumes *consistent-interp* I **and** $I \models_{\text{sext}} A$

shows *satisfiable* A

by (*metis Un-empty-left assms satisfiable-carac subset-Un-eq sup.left-idem*

total-over-m-consistent-extension total-over-m-empty true-clss-ext-def)

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

assumes $\neg \text{consistent-interp } I$

shows $I \models_{\text{sext}} A$

by (*meson assms consistent-interp-subset true-clss-ext-def*)

```
end
theory Prop-Logic
imports Main
begin
```


Chapter 3

Normalisation

We define here the normalisation from formula towards conjunctive and disjunctive normal form, including normalisation towards multiset of multisets to represent CNF.

3.1 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.

3.1.1 Definition and abstraction

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

datatype *'v propo* =
 FT | *FF* | *FVar* *'v* | *FNot* *'v propo* | *FAnd* *'v propo* *'v propo* | *FOR* *'v propo* *'v propo*
 | *FImp* *'v propo* *'v propo* | *FEq* *'v propo* *'v propo*

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

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

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

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

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

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

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

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

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

```
fun conn :: 'v connective  $\Rightarrow$  'v propo list  $\Rightarrow$  'v propo where
conn CT [] = FT |
conn CF [] = FF |
conn (CVar v) [] = FVar v |
conn CNot [ $\varphi$ ] = FNot  $\varphi$  |
conn CAnd ( $\varphi$  # [ $\psi$ ]) = FAnd  $\varphi$   $\psi$  |
conn COr ( $\varphi$  # [ $\psi$ ]) = FOr  $\varphi$   $\psi$  |
conn CImp ( $\varphi$  # [ $\psi$ ]) = FImp  $\varphi$   $\psi$  |
conn CEq ( $\varphi$  # [ $\psi$ ]) = FEq  $\varphi$   $\psi$  |
conn - - = FF
```

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

```
lemma connective-cases-arity[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]:  $(c = CT \vee c = CF \vee c = CVar v) \implies \text{wf-conn } c []$  |
wf-conn-unary[simp]:  $c = CNot \implies \text{wf-conn } c [\psi]$  |
wf-conn-binary[simp]:  $c \in \text{binary-connectives} \implies \text{wf-conn } c (\psi \# \psi' \# [])$ 
thm wf-conn.induct
```

```
lemma wf-conn-induct[consumes 1, case-names CT CF CVar CNot COr CAnd CImp CEq]:
assumes wf-conn c x and
 $\bigwedge v. c = CT \implies P []$  and
 $\bigwedge v. c = CF \implies P []$  and
 $\bigwedge v. c = CVar v \implies P []$  and
 $\bigwedge \psi. c = CNot \implies P [\psi]$  and
 $\bigwedge \psi \psi'. c = COr \implies P [\psi, \psi']$  and
 $\bigwedge \psi \psi'. c = CAnd \implies P [\psi, \psi']$  and
 $\bigwedge \psi \psi'. c = CImp \implies P [\psi, \psi']$  and
 $\bigwedge \psi \psi'. c = CEq \implies P [\psi, \psi']$ 
shows P x
using assms by induction (auto simp: binary-connectives-def)
```

3.1.2 properties of the abstraction

First we can define simplification rules.

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



```

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

```

```

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

```

```

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

```

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

```

lemma list-length2-decomp: length l = 2  $\implies$  ( $\exists$  a b. l = a # b # [])
apply (induct l, auto)
by (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 ψ :: 'v propo
  case wf-conn-unary
  then show ?P [ψ] using conn binary-connectives-def
  using connective.distinct by blast

```

```

next
  fix  $\psi \psi' :: 'v \text{ propo}$ 
  show  $?P [\psi, \psi']$  by auto
qed
qed

```

```

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

```

Decomposing the Not into an element is moreover very useful.

```

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

```

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

```

lemma wf-conn-no-arity-change:
   $\text{length } l = \text{length } l' \implies \text{wf-conn } c\ l \longleftrightarrow \text{wf-conn } c\ l'$ 
proof -
  {
    fix  $l\ l'$ 
    have  $\text{length } l = \text{length } l' \implies \text{wf-conn } c\ l \implies \text{wf-conn } c\ l'$ 
      apply (cases  $c\ l$  rule: wf-conn.induct, auto)
      by (metis wf-conn-bin-list-length)
  }
  then show  $\text{length } l = \text{length } l' \implies \text{wf-conn } c\ l = \text{wf-conn } c\ l'$  by metis
qed

```

```

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

```

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

```

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

```

```

lemma conn-inj:
  fixes  $c\ ca :: 'v \text{ connective}$  and  $l\ \psi s :: 'v \text{ propo list}$ 
  assumes  $\text{corr}: \text{wf-conn } ca\ l$ 
  and  $\text{corr}': \text{wf-conn } c\ \psi s$ 

```

```

and eq: conn ca l = conn c  $\psi$ s
shows ca = c  $\wedge$   $\psi$ s = l
using corr
proof (cases ca l rule: wf-conn.cases)
case (wf-conn-nullary v)
then show ca = c  $\wedge$   $\psi$ s = l using assms
by (metis conn.simps(1) conn.simps(2) conn.simps(3) wf-conn-list(1-3))
next
case (wf-conn-unary  $\psi'$ )
then have *: FNot  $\psi'$  = conn c  $\psi$ s 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  $\psi$ s = l using * conn-inj-not(2) corr' wf-conn-unary(1) by force
ultimately show ca = c  $\wedge$   $\psi$ s = l by auto
next
case (wf-conn-binary  $\psi'$   $\psi''$ )
then show ca = c  $\wedge$   $\psi$ s = 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

```

3.1.3 Subformulas and properties

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

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

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

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:
shows *b*: FNot $\varphi \preceq \psi \Rightarrow \varphi \preceq \psi$
apply (induct rule: subformula.induct)
using *subformula-into-subformula* wf-conn-unary subformula-refl list.set-intros(1) subformula-refl
by (fastforce intro: subformula-into-subformula)+

lemma *subformula-in-binary-conn*:
assumes *conn*: $c \in \text{binary-connectives}$
shows $f \preceq \text{conn } c \ [f, g]$
and $g \preceq \text{conn } c \ [f, g]$
proof –
have *a*: 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*: 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 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  $\varphi$ ) 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  $\varphi$ ) = vars-of-prop  $\varphi$  |
vars-of-prop (FAnd  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi \cup \text{vars-of-prop } \psi$  |
vars-of-prop (FOr  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi \cup \text{vars-of-prop } \psi$  |
vars-of-prop (FImp  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi \cup \text{vars-of-prop } \psi$  |
vars-of-prop (FEq  $\varphi$   $\psi$ ) = vars-of-prop  $\varphi \cup \text{vars-of-prop } \psi$ 

```

```

lemma vars-of-prop-incl-conn:
  fixes  $\xi \xi' :: 'v \text{ propo list}$  and  $\psi :: 'v \text{ propo}$  and  $c :: 'v \text{ connective}$ 
  assumes corr: wf-conn c l and incl:  $\psi \in \text{set } l$ 
  shows vars-of-prop  $\psi \subseteq \text{vars-of-prop } (\text{conn } c l)$ 
proof (cases c rule: connective-cases-arity-2)

```

```

case nullary
then have False using corr incl by auto
then show vars-of-prop  $\psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$  by blast
next
case binary note c = this
then obtain a b where ab:  $l = [a, b]$ 
  using wf-conn-bin-list-length list-length2-decomp corr by metis
then have  $\psi = a \vee \psi = b$  using incl by auto
then show vars-of-prop  $\psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$ 
  using ab c unfolding binary-connectives-def by auto
next
case unary note c = this
fix  $\varphi :: 'v \text{ propo}$ 
have  $l = [\psi]$  using corr c incl split-list by force
then show vars-of-prop  $\psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$  using c by auto
qed

```

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:

```

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

```

3.1.4 Positions

Instead of 1 or 2 we use L or R

datatype *sign* = $L \mid R$

We use *nil* instead of ε .

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

```

pos FF = {[]} |
pos FT = {[]} |
pos (FVar x) = {[]} |
pos (FAnd  $\varphi \ \psi$ ) = {[]}  $\cup \{ L \ \# \ p \mid p. p \in \text{pos } \varphi \} \cup \{ R \ \# \ p \mid p. p \in \text{pos } \psi \}$  |
pos (FOr  $\varphi \ \psi$ ) = {[]}  $\cup \{ L \ \# \ p \mid p. p \in \text{pos } \varphi \} \cup \{ R \ \# \ p \mid p. p \in \text{pos } \psi \}$  |
pos (FEq  $\varphi \ \psi$ ) = {[]}  $\cup \{ L \ \# \ p \mid p. p \in \text{pos } \varphi \} \cup \{ R \ \# \ p \mid p. p \in \text{pos } \psi \}$  |
pos (FImp  $\varphi \ \psi$ ) = {[]}  $\cup \{ L \ \# \ p \mid p. p \in \text{pos } \varphi \} \cup \{ R \ \# \ p \mid p. p \in \text{pos } \psi \}$  |
pos (FNot  $\varphi$ ) = {[]}  $\cup \{ L \ \# \ p \mid p. p \in \text{pos } \varphi \}$ 

```

lemma *finite-pos*: *finite* (*pos* φ)

by (*induct* φ , *auto*)

lemma *finite-inj-comp-set*:

```

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

```

proof (*induct* s *rule*: *finite-induct*)

show *card* $\{f \ p \mid p. p \in \{\}\} = \text{card } \{\}$ **by** *auto*

next

```

fix x :: 'v and s :: 'v set
assume f: finite s and notin:  $x \notin s$ 
and IH: card  $\{f \ p \mid p. p \in s\} = \text{card } s$ 

```

have f' : *finite* $\{f\ p \mid p. p \in \text{insert } x\ s\}$ **using** f **by** *auto*
have *notin'*: $f\ x \notin \{f\ p \mid p. p \in s\}$ **using** *notin inj injD* **by** *fastforce*
have $\{f\ p \mid p. p \in \text{insert } x\ s\} = \text{insert } (f\ x) \{f\ p \mid p. p \in s\}$ **by** *auto*
then have $\text{card } \{f\ p \mid p. p \in \text{insert } x\ s\} = 1 + \text{card } \{f\ p \mid p. p \in s\}$
using *finite card-insert-disjoint f' notin'* **by** *auto*
moreover have $\dots = \text{card } (\text{insert } x\ s)$ **using** *notin f IH* **by** *auto*
finally show $\text{card } \{f\ p \mid p. p \in \text{insert } x\ s\} = \text{card } (\text{insert } x\ s)$.
qed

lemma *cons-inject*:

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

lemma *finite-insert-nil-cons*:

finite s \implies 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 φ)) = 1 + card (pos φ)
by (*simp add: cons-inject finite-inj-comp-set finite-pos*)

lemma *card-seperate*:

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

proof –

have *finite ?L* **using** *assms* **by** *auto*
moreover have *finite ?R* **using** *assms* **by** *auto*
moreover have $?L \cap ?R = \{\}$ **by** *blast*
ultimately show *?thesis* **using** *assms card-Un-disjoint* **by** *blast*

qed

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

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

fixes $\varphi :: 'v\ \text{propo}$
shows $\text{card } (\text{vars-of-prop } \varphi) \leq \text{prop-size } \varphi$

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

proof –

fix $\varphi1\ \varphi2 :: 'v\ \text{propo}$
assume *IH1: card (vars-of-prop $\varphi1$) \leq card (pos $\varphi1$)*
and *IH2: card (vars-of-prop $\varphi2$) \leq card (pos $\varphi2$)*
let $?L = \{L \# p \mid p. p \in \text{pos } \varphi1\}$
let $?R = \{R \# p \mid p. p \in \text{pos } \varphi2\}$
have $\text{card } (?L \cup ?R) = \text{card } ?L + \text{card } ?R$
using *card-seperate finite-pos* **by** *blast*
moreover have $\dots = \text{card } (\text{pos } \varphi1) + \text{card } (\text{pos } \varphi2)$
by (*simp add: cons-inject finite-inj-comp-set finite-pos*)
moreover have $\dots \geq \text{card } (\text{vars-of-prop } \varphi1) + \text{card } (\text{vars-of-prop } \varphi2)$ **using** *IH1 IH2* **by** *arith*
then have $\dots \geq \text{card } (\text{vars-of-prop } \varphi1 \cup \text{vars-of-prop } \varphi2)$ **using** *card-Un-le le-trans* **by** *blast*
ultimately
show $\text{card } (\text{vars-of-prop } \varphi1 \cup \text{vars-of-prop } \varphi2) \leq \text{Suc } (\text{card } (?L \cup ?R))$
 $\text{card } (\text{vars-of-prop } \varphi1 \cup \text{vars-of-prop } \varphi2) \leq \text{Suc } (\text{card } (?L \cup ?R))$
 $\text{card } (\text{vars-of-prop } \varphi1 \cup \text{vars-of-prop } \varphi2) \leq \text{Suc } (\text{card } (?L \cup ?R))$

```

      card (vars-of-prop  $\varphi 1 \cup$  vars-of-prop  $\varphi 2$ )  $\leq$  Suc (card (?L  $\cup$  ?R))
    by auto
  qed

```

```

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

```

```

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

```

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

lemma path-to-subformula:

```

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

```

lemma subformula-path-exists:

```

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

```

proof (induct rule: subformula.induct)

```

  case subformula-refl
  have path-to []  $\varphi'$   $\varphi'$  by auto
  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  $\exists p. \text{path-to } p$  (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  $\exists p. \text{path-to } p$  (conn  $c$   $l$ )  $\varphi'$  by blast

```

}

moreover {

```

  assume  $c$ :  $c \in$  binary-connectives
  obtain  $a$   $b$  where  $ab$ :  $[a, b] = l$  using subformula-into-subformula  $c$  wf-conn-bin-list-length
    list-length2-decomp by metis
  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  $\exists p. \text{path-to } p$  (conn  $c$   $l$ )  $\varphi'$  by blast

```

}

ultimately show $\exists p. \text{path-to } p$ (conn c l) φ' **using** connective-cases-arity **by** metis

qed


```

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

```

3.2 Semantics over the syntax

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

```

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

```

```

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

```

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

theorem *deduction-theorem*:

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

proof

```

assume H:  $\varphi \models_f \psi$ 
{
  fix A
  have  $A \models FImp\ \varphi\ \psi$ 
  proof (cases  $A \models \varphi$ )
    case True
    then have  $A \models \psi$  using H unfolding evalf-def by metis
    then show  $A \models FImp\ \varphi\ \psi$  by auto
  next
    case False
    then show  $A \models FImp\ \varphi\ \psi$  by auto
  qed
}
then show  $\forall A. A \models FImp\ \varphi\ \psi$  by blast
next
assume A:  $\forall A. A \models FImp\ \varphi\ \psi$ 
show  $\varphi \models_f \psi$ 
proof (rule ccontr)
  assume  $\neg \varphi \models_f \psi$ 
  then obtain A where  $A \models \varphi$  and  $\neg A \models \psi$  using evalf-def by metis

```

```

    then have  $\neg A \models \text{FImp } \varphi \ \psi$  by auto
    then show False using A by blast
qed

```

A shorter proof:

```

lemma  $\varphi \models_f \psi \iff (\forall A. A \models \text{FImp } \varphi \ \psi)$ 
  by (simp add: evalf-def)

```

definition *same-over-set::* ($'v \Rightarrow \text{bool}$) \Rightarrow ($'v \Rightarrow \text{bool}$) \Rightarrow $'v \text{ set} \Rightarrow \text{bool}$ **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  $A \models \varphi \iff B \models \varphi$ 
  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.

3.3 Rewrite systems and properties

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

```

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

This lemma is only a health condition:

```

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

```

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

```

lemma propo-rew-step-subformula-rec:
  fixes  $\psi \ \psi' \ \varphi :: 'v \text{ propo}$ 
  shows  $\psi \preceq \varphi \implies r \ \psi \ \psi' \implies (\exists \varphi'. \ \psi' \preceq \varphi' \wedge \text{propo-rew-step } r \ \psi \ \varphi')$ 
proof (induct  $\varphi$  rule: subformula.induct)
  case subformula-refl
  then have propo-rew-step  $r \ \psi \ \psi'$  using propo-rew-step.intros by auto
  moreover have  $\psi' \preceq \psi'$  using Prop-Logic.subformula-refl by auto
  ultimately show  $\exists \varphi'. \ \psi' \preceq \varphi' \wedge \text{propo-rew-step } r \ \psi \ \varphi'$  by fastforce
next
  case (subformula-into-subformula  $\psi'' \ l \ c$ )
  note IH = this(4) and  $r = \text{this}(5)$  and  $\psi'' = \text{this}(1)$  and  $wf = \text{this}(2)$  and  $\text{incl} = \text{this}(3)$ 
  then obtain  $\varphi'$  where  $*$ :  $\psi' \preceq \varphi' \wedge \text{propo-rew-step } r \ \psi'' \ \varphi'$  by metis
  moreover obtain  $\xi \ \xi' :: 'v \text{ propo list}$  where
     $l: l = \xi @ \psi'' \# \xi'$  using List.split-list  $\psi''$  by metis
  ultimately have propo-rew-step  $r \ (\text{conn } c \ l) \ (\text{conn } c \ (\xi @ \varphi' \# \xi'))$ 
    using propo-rew-step.intros(2)  $wf$  by metis
  moreover have  $\psi' \preceq \text{conn } c \ (\xi @ \varphi' \# \xi')$ 
    using  $wf * wf\text{-conn-no-arity-change}$  Prop-Logic.subformula-into-subformula
    by (metis (no-types) in-set-conv-decomp  $l \ wf\text{-conn-no-arity-change-helper}$ )
  ultimately show  $\exists \varphi'. \ \psi' \preceq \varphi' \wedge \text{propo-rew-step } r \ (\text{conn } c \ l) \ \varphi'$  by metis
qed

```

```

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

```

```

lemma consistency-decompose-into-list:
  assumes  $wf: wf\text{-conn } c \ l$  and  $wf': wf\text{-conn } c \ l'$ 
  and same:  $\forall n. \ A \models l ! n \longleftrightarrow (A \models l' ! n)$ 
  shows  $A \models \text{conn } c \ l \longleftrightarrow A \models \text{conn } c \ l'$ 
proof (cases  $c$  rule: connective-cases-arity-2)
  case nullary
  then show  $(A \models \text{conn } c \ l) \longleftrightarrow (A \models \text{conn } c \ l')$  using  $wf \ wf'$  by auto
next
  case unary note  $c = \text{this}$ 
  then obtain  $a$  where  $l: l = [a]$  using  $wf\text{-conn-Not-decomp } wf$  by metis
  obtain  $a'$  where  $l': l' = [a']$  using  $wf\text{-conn-Not-decomp } wf' \ c$  by metis
  have  $A \models a \longleftrightarrow A \models a'$  using  $l \ l'$  by (metis nth-Cons-0 same)
  then show  $A \models \text{conn } c \ l \longleftrightarrow A \models \text{conn } c \ l'$  using  $l \ l' \ c$  by auto
next
  case binary note  $c = \text{this}$ 
  then obtain  $a \ b$  where  $l: l = [a, b]$ 
    using  $wf\text{-conn-bin-list-length list-length2-decomp } wf$  by metis
  obtain  $a' \ b'$  where  $l': l' = [a', b']$ 
    using  $wf\text{-conn-bin-list-length list-length2-decomp } wf' \ c$  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  $\text{rel} = \text{this}(1)$  and  $\text{IH0} = \text{this}(2)$  and  $\text{corr} = \text{this}(3)$ 
  obtain  $\psi \ \psi' \ p$  where  $\text{IH}: r \ \psi \ \psi' \wedge \text{path-to } p \ \varphi \ \psi \wedge \text{replace-at } p \ \varphi \ \psi' = \varphi'$  using IH0 by metis

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

```

```

    ultimately have ?case using ab by blast
  }
  ultimately show ?case using connective-cases-arity by blast
qed

```

3.3.2 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*

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

next

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

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

{

fix A

from IH **have** $\forall n. (A \models (\xi @ \varphi \# \xi') ! n) = (A \models (\xi @ \varphi' \# \xi') ! n)$

by (*metis (mono-tags, hide-lams) list-update-length nth-Cons-0 nth-append-length-plus nth-list-update-neg*)

then have $(A \models \text{conn } c (\xi @ \varphi \# \xi')) = (A \models \text{conn } c (\xi @ \varphi' \# \xi'))$

by (*meson consistency-decompose-into-list wf wf-conn-no-arity-change-helper wf-conn-no-arity-change*)

}

then show $\forall A. A \models \text{conn } c (\xi @ \varphi \# \xi') \longleftrightarrow A \models \text{conn } c (\xi @ \varphi' \# \xi')$ **by** *auto*

qed

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

assumes *preserves-un-sat* r

shows *preserves-un-sat* (*propo-rew-step* r)

using *assms* **by** (*simp add: preserves-un-sat-def propo-rew-step-preservers-val-explicit*)

lemma *preserves-un-sat-OO[intro]*:

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

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

lemma *star-consistency-preservation-explicit*:

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

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

3.3.3 Full Lifting

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

lemma *full-ropo-rew-step-preservers-val*[simp]:
preserves-un-sat $r \implies \text{preserves-un-sat } (\text{full } (\text{propo-rew-step } r))$
by (*metis full-def preserves-un-sat-def star-consistency-preservation*)

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

3.4 Transformation testing

3.4.1 Definition and first properties

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\longleftrightarrow (*test-symb* (*FNot* φ) \wedge *all-subformula-st test-symb* φ) **by** *simp*
qed

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

lemma *subformula-all-subformula-st*:

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

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

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

fixes *r*:: '*v* *propo* \Rightarrow '*v* *propo* \Rightarrow *bool* **and** *test-symb*:: '*v* *propo* \Rightarrow *bool* **and** *x*:: '*v*
and φ :: '*v* *propo*

assumes

test-symb-false-nullary: $\forall x. \text{test-symb } FF \wedge \text{test-symb } FT \wedge \text{test-symb } (FVar\ x) \text{ and }$
 $\forall \varphi'. \varphi' \preceq \varphi \longrightarrow (\neg \text{test-symb } \varphi') \longrightarrow (\exists \psi. r\ \varphi'\ \psi) \text{ and }$
 $\neg \text{all-subformula-st test-symb } \varphi$

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

using *assms*

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

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

then show $\exists \psi\ \psi'. \psi \preceq \varphi \wedge r\ \psi\ \psi'$

using *wf-conn-nullary test-symb-false-nullary* **by** *fastforce*

next

case (*unary* φ) **note** *IH* = *this*(1)[*OF this*(2)] **and** *r* = *this*(2) **and** *nst* = *this*(3) **and** *subf* = *this*(4)

from *r IH nst* **have** *H*: $\neg \text{all-subformula-st test-symb } \varphi \implies \exists \psi. \psi \preceq \varphi \wedge (\exists \psi'. r\ \psi\ \psi')$

by (*metis subformula-in-subformula-not subformula-refl subformula-trans*)

{

assume *n*: $\neg \text{test-symb } (FNot\ \varphi)$

obtain ψ **where** *r* (*FNot* φ) ψ **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*: *test-symb* (*FNot* φ)

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

using *all-subformula-st-decomp-explicit*(3) *nst subf* **by** *blast*

then have $\exists \psi\ \psi'. \psi \preceq FNot\ \varphi \wedge r\ \psi\ \psi'$

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* $\varphi1-0$ = *this*(1)[*OF this*(4)] **and** *IH* $\varphi2-0$ = *this*(2)[*OF this*(4)] **and** *r* = *this*(4)

and φ = *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 *conn c* [$\varphi1, \varphi2$] = φ

using φ **by** *fastforce*

then have *corr*: *wf-conn c* [$\varphi1, \varphi2$] **using** *wf-conn.simps unfolding binary-connectives-def* **by** *auto*

have *inc*: $\varphi1 \preceq \varphi\ \varphi2 \preceq \varphi$ **using** *binary-connectives-def c subformula-in-binary-conn* **by** *blast+*


```

from  $r$   $IH\varphi1-0$  have  $IH\varphi1: \neg \text{all-subformula-st test-symb } \varphi1 \implies \exists \psi \psi'. \psi \preceq \varphi1 \wedge r \psi \psi'$ 
  using  $\text{inc}(1)$   $\text{subformula-trans le}$  by  $\text{blast}$ 
from  $r$   $IH\varphi2-0$  have  $IH\varphi2: \neg \text{all-subformula-st test-symb } \varphi2 \implies \exists \psi. \psi \preceq \varphi2 \wedge (\exists \psi'. r \psi \psi')$ 
  using  $\text{inc}(2)$   $\text{subformula-trans le}$  by  $\text{blast}$ 
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  $c \text{ nst}$  by  $\text{auto}$ 
show  $\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi'$ 
  using  $IH\varphi1$   $IH\varphi2$   $\text{subformula-trans inc subformula-refl cases le}$  by  $\text{blast}$ 
qed

```

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

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  $\text{assms}(3-5)$ 
proof (induct rule: propo-rew-step.induct)
case global-rel
  then show ?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  $\Phi$   $\text{corr}$   $\text{subformula-into-subformula subformula-refl subformula-trans}$ 
  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 all-subformula-st-decomp nst by blast
then have *:  $\forall \psi. \psi \in \text{set } (\xi @ \varphi' \# \xi') \longrightarrow \text{all-subformula-st test-symb } \psi$  using  $\varphi$  sq by fastforce
then have test-symb  $\varphi'$  using all-subformula-st-test-symb-true-phi by auto
moreover from corr nst have test-symb (conn c ( $\xi @ \varphi \# \xi'$ ))
  using all-subformula-st-decomp by blast
ultimately have test-symb: test-symb (conn c ( $\xi @ \varphi' \# \xi'$ )) using  $H'$  sq corr rel by blast

have wf-conn c ( $\xi @ \varphi' \# \xi'$ )
  by (metis wf-conn-no-arity-change-helper corr wf-conn-no-arity-change)
then show all-subformula-st test-symb (conn c ( $\xi @ \varphi' \# \xi'$ ))
  using * test-symb by (metis all-subformula-st-decomp)
qed

```

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

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

```

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

```

The lemmas can be lifted to *propo-rew-step* r^\downarrow instead of *propo-rew-step*

Invariant after all rewriting

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

```

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

```

proof –

```

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

```

qed
qed

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

fixes $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **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**

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

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

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

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

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

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

assumes

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

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

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

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

unfolding *full-def*

proof –

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

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

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

using *init*

proof (*induct rule: rtrancpl-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 subformula-refl H H' rel one* **by** *auto*

qed

qed

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

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

assumes

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

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

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

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

proof –

```

have  $\bigwedge(c :: 'v \text{ connective}) \xi \varphi \xi' \varphi'. \text{wf-conn } c (\xi @ \varphi \# \xi')$ 
   $\implies \text{test-symb } (\text{conn } c (\xi @ \varphi \# \xi')) \implies \text{test-symb } \varphi' \implies \text{test-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$ 
  using  $H'$  by (metis wf-conn-no-arity-change-helper wf-conn-no-arity-change)
then show all-subformula-st test-symb  $\psi$ 
  using  $H$  full init full-propo-rew-step-inv-stay by blast
qed

end
theory Prop-Normalisation
imports Main Prop-Logic Prop-Abstract-Transformation ../lib/Multiset-More
begin

```

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

3.5 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.

3.5.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

```

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

```

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

```

lemma no-equiv-symb-conn-characterization[simp]:
fixes  $c :: 'v \text{ connective}$  and  $l :: 'v \text{ propo list}$ 
assumes wf: wf-conn  $c \ l$ 
shows no-equiv-symb (conn  $c \ l$ )  $\longleftrightarrow c \neq \text{CEq}$ 

```

by (metis connective.distinct(13,25,35,43) wf no-equiv-symb.elims(3) no-equiv-symb.simps(1)
wf-conn.cases wf-conn-list(6))

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

lemma no-equiv-eq[simp]:

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

shows

$\neg \text{no-equiv } (FEq \ \varphi \ \psi)$

no-equiv FT

no-equiv FF

using no-equiv-symb.simps(1) all-subformula-st-test-symb-true-phi unfolding no-equiv-def by auto

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

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

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

shows

no-equiv (FNot φ) \longleftrightarrow no-equiv φ

no-equiv (FAnd $\varphi \ \psi$) \longleftrightarrow (no-equiv $\varphi \ \wedge$ no-equiv ψ)

no-equiv (FOr $\varphi \ \psi$) \longleftrightarrow (no-equiv $\varphi \ \wedge$ no-equiv ψ)

no-equiv (FImp $\varphi \ \psi$) \longleftrightarrow (no-equiv $\varphi \ \wedge$ no-equiv ψ)

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:

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

unfolding no-equiv-def by auto

moreover {

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

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

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

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

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

}

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

using elim-equiv.cases by auto

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

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

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

ultimately show ?thesis

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

qed

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

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

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

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

3.5.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* :: 'v propo \Rightarrow 'v propo \Rightarrow bool **where**
[simp]: *elim-imp* (*FImp* φ ψ) (*FOr* (*FNot* φ) ψ)

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

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

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

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

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

lemma *no-imp-symb-conn-characterization*:
 $\text{wf-conn } c \ l \implies \text{no-imp-symb } (\text{conn } c \ l) \longleftrightarrow c \neq \text{CImp}$
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*]:
 $\neg \text{no-imp } (\text{FImp } \varphi \ \psi)$
 $\text{no-imp } \text{FT}$
 $\text{no-imp } \text{FF}$
unfolding *no-imp-def* **by** *auto*

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

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:
 $\text{elim-imp } \varphi \ \psi \implies \text{no-equiv } \varphi \implies \text{no-equiv } \psi$
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*: $\neg \text{no-imp } \varphi$

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

proof –

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

moreover {

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

have *H*: $\text{elim-imp } (\text{conn } c \ l) \ \psi \implies \neg \text{no-imp-symb } (\text{conn } c \ l)$

by (*auto elim: elim-imp.cases*)

}

moreover

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

by (*auto elim: elim-imp.cases*)+

moreover

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

by (*case-tac* φ) (*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*: $\text{full } (\text{propo-rew-step } \text{elim-imp}) \ \varphi \ \psi \implies \text{no-imp } \psi$

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

3.5.3 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* *propo* \Rightarrow bool **where**
no-T-F-symb-comp: $c \neq CF \Rightarrow c \neq CT \Rightarrow wf_conn\ c\ l \Rightarrow (\forall \varphi \in set\ l. \varphi \neq FT \wedge \varphi \neq FF)$
 $\Rightarrow no_T_F_symb\ (conn\ c\ l)$

lemma *wf-conn-no-T-F-symb-iff[simp]*:
 $wf_conn\ c\ \psi s \Rightarrow$
 $no_T_F_symb\ (conn\ c\ \psi s) \longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall \psi \in set\ \psi s. \psi \neq FF \wedge \psi \neq FT))$
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]*:
 $no_T_F_symb\ (FAnd\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
 $no_T_F_symb\ (FOr\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
 $no_T_F_symb\ (FEq\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
 $no_T_F_symb\ (FImp\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
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*:
 $\neg no_T_F_symb\ (FNot\ \varphi) \Rightarrow \varphi = FT \vee \varphi = FF$
proof (*rule ccontr*)
assume *n*: $\neg no_T_F_symb\ (FNot\ \varphi)$
assume $\neg (\varphi = FT \vee \varphi = FF)$
then have $\forall \varphi' \in set\ [\varphi]. \varphi' \neq FT \wedge \varphi' \neq FF$ **by** *auto*
moreover have *wf-conn* *CNot* $[\varphi]$ **by** *simp*
ultimately have *no-T-F-symb* (*FNot* φ)
using *no-T-F-symb.intros* **by** (*metis* *conn.simps(4)* *connective.distinct(5,17)*)

then show *False* **using** *n* **by** *blast*
qed

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

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

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

lemma *no-T-F-symb-except-top-level-false-not[simp]*:
fixes $\varphi \psi :: 'v \text{ propo}$
assumes $\varphi = FT \vee \varphi = FF$
shows

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

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

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

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

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

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

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

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

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

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

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

lemma *no-T-F-except-top-level-simp*[*simp*]: *no-T-F-except-top-level* FF *no-T-F-except-top-level* FT
unfolding *no-T-F-except-top-level-def* **by** *auto*

lemma *no-T-F-no-T-F-except-top-level'*[*simp*]:
no-T-F-except-top-level $\varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee$ *no-T-F* $\varphi)$
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* ∈ *binary-connectives*
shows *no-T-F* (*conn c* [*φ*, *ψ*]) \longleftrightarrow (*no-T-F* *φ* ∧ *no-T-F* *ψ*)
proof –
have *wf*: *wf-conn c* [*φ*, *ψ*] **using** *c* **by** *auto*
then have *no-T-F* (*conn c* [*φ*, *ψ*]) \longleftrightarrow (*no-T-F-symb* (*conn c* [*φ*, *ψ*]) ∧ *no-T-F* *φ* ∧ *no-T-F* *ψ*)
by (*simp add: all-subformula-st-decomp no-T-F-def*)
then show *no-T-F* (*conn c* [*φ*, *ψ*]) \longleftrightarrow (*no-T-F* *φ* ∧ *no-T-F* *ψ*)
using *c wf all-subformula-st-decomp list.discI no-T-F-def no-T-F-symb-except-toplevel-bin-decom*
no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(1,2) wf-conn-helper-facts(2,3)
wf-conn-list(1,2) **by** *metis*
qed

lemma *no-T-F-bin-decomp-expanded*[simp]:
assumes *c*: *c* = *CAnd* ∨ *c* = *COr* ∨ *c* = *CEq* ∨ *c* = *CImp*
shows *no-T-F* (*conn c* [*φ*, *ψ*]) \longleftrightarrow (*no-T-F* *φ* ∧ *no-T-F* *ψ*)
using *no-T-F-bin-decomp assms unfolding binary-connectives-def* **by** *blast*

lemma *no-T-F-comp-expanded-explicit*[simp]:
fixes *φ ψ* :: '*v* *propo*
shows
no-T-F (*FAnd* *φ ψ*) \longleftrightarrow (*no-T-F* *φ* ∧ *no-T-F* *ψ*)
no-T-F (*FOr* *φ ψ*) \longleftrightarrow (*no-T-F* *φ* ∧ *no-T-F* *ψ*)
no-T-F (*FEq* *φ ψ*) \longleftrightarrow (*no-T-F* *φ* ∧ *no-T-F* *ψ*)
no-T-F (*FImp* *φ ψ*) \longleftrightarrow (*no-T-F* *φ* ∧ *no-T-F* *ψ*)
using *conn.simps(5–8) no-T-F-bin-decomp-expanded* **by** (*metis (no-types)*)⁺

lemma *no-T-F-comp-not*[simp]:
fixes *φ ψ* :: '*v* *propo*
shows *no-T-F* (*FNot* *φ*) \longleftrightarrow *no-T-F* *φ*
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 *φ ψ* :: '*v* *propo*
assumes *φ*: *no-T-F* (*FAnd* *φ ψ*) ∨ *no-T-F* (*FOr* *φ ψ*) ∨ *no-T-F* (*FEq* *φ ψ*) ∨ *no-T-F* (*FImp* *φ ψ*)
shows *no-T-F* *ψ* **and** *no-T-F* *φ*
using *assms* **by** *auto*

lemma *no-T-F-decomp-not*:
fixes *φ* :: '*v* *propo*
assumes *φ*: *no-T-F* (*FNot* *φ*)
shows *no-T-F* *φ*
using *assms* **by** *auto*

lemma *no-T-F-symb-except-toplevel-step-exists*:
fixes *φ ψ* :: '*v* *propo*
assumes *no-equiv φ* **and** *no-imp φ*
shows *ψ* ≤ *φ* \implies \neg *no-T-F-symb-except-toplevel* *ψ* \implies $\exists \psi'. \text{elimTB } \psi \ \psi'$
proof (*induct ψ rule: propo-induct-arity*)
case (*nullary φ' 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 ψ*)
then have *ψ* = *FF* ∨ *ψ* = *FT* **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*

```

then show ?case using ElimTB5 ElimTB6 by blast
next
case (binary  $\varphi'$   $\psi 1$   $\psi 2$ )
note IH1 = this(1) and IH2 = this(2) and  $\varphi' = \text{this}(3)$  and  $F\varphi = \text{this}(4)$  and  $n = \text{this}(5)$ 
{
  assume  $\varphi' = FImp \psi 1 \psi 2 \vee \varphi' = FEq \psi 1 \psi 2$ 
  then have False using n F $\varphi$  subformula-all-subformula-st assms
    by (metis (no-types) no-equiv-eq(1) no-equiv-def no-imp-Imp(1) no-imp-def)
  then have ?case by blast
}
moreover {
  assume  $\varphi'$ :  $\varphi' = FAnd \psi 1 \psi 2 \vee \varphi' = FOr \psi 1 \psi 2$ 
  then have  $\psi 1 = FT \vee \psi 2 = FT \vee \psi 1 = FF \vee \psi 2 = FF$ 
    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  $\varphi :: 'v$  propo
assumes noTB:  $\neg$  no-T-F-except-top-level  $\varphi$  and no-equiv: no-equiv  $\varphi$  and no-imp: no-imp  $\varphi$ 
shows  $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elimTB } \psi \psi'$ 

```

proof –

```

have test-symb-false-nullary:  $\forall x. \text{no-T-F-symb-except-toplevel } (FF :: 'v \text{ propo})$ 
   $\wedge \text{no-T-F-symb-except-toplevel } FT \wedge \text{no-T-F-symb-except-toplevel } (FVar (x :: 'v))$  by auto

```

moreover {

```

  fix c :: 'v connective and l :: 'v propo list and  $\psi :: 'v$  propo
  have H: elimTB (conn c l)  $\psi \implies \neg \text{no-T-F-symb-except-toplevel } (\text{conn } c \text{ l})$ 
    by (cases conn c l rule: elimTB.cases, auto)

```

}

moreover {

```

  fix x :: 'v
  have H': no-T-F-except-top-level FT no-T-F-except-top-level FF
    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  $\varphi \psi :: 'v$  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  $\varphi \psi :: 'v$  propo
  have H: 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  $\varphi \psi :: 'v$  propo
  have  $H$ : elimTB  $\varphi \psi \implies$  no-imp  $\varphi \implies$  no-imp  $\psi$ 
    by (induct  $\varphi \psi$  rule: elimTB.induct, auto)
}
then show no-imp  $\psi$ 
  using full-propo-rew-step-inv-stay-conn[of elimTB no-imp-symb  $\varphi \psi$ ] assms
  no-imp-symb-conn-characterization unfolding no-imp-def by metis
qed

```

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

3.5.4 PushNeg

Push the negation inside the formula, until the literal.

inductive pushNeg **where**

```

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

```

lemma pushNeg-transformation-consistent:

```

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

```

lemma pushNeg-explicit: 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-consistent:

```

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

```

fun simple **where**

```

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

```

lemma *simple-decomp*:

simple $\varphi \longleftrightarrow (\varphi = FT \vee \varphi = FF \vee (\exists x. \varphi = FVar\ x))$
by (*cases* φ) *auto*

lemma *subformula-conn-decomp-simple*:

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

assumes *s*: *simple* ψ

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

proof –

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

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

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

shows

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

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

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

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

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

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

simple-not-symb - = *True*

definition *simple-not* **where**

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

declare *simple-not-def[simp]*

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

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

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

by *auto*

lemma *simple-not-step-exists*:

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

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

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

apply (*induct* ψ , *auto*)

apply (*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 $\varphi :: 'v\ \text{propo}$

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

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

proof –

have $\forall x. \text{simple-not-symb}\ (FF :: 'v\ \text{propo}) \wedge \text{simple-not-symb}\ FT \wedge \text{simple-not-symb}\ (FVar\ (x :: 'v))$
by *auto*

moreover {

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

have *H*: $\text{pushNeg}\ (conn\ c\ l)\ \psi \implies \neg \text{simple-not-symb}\ (conn\ c\ l)$

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

proof –

```

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

```

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 $\varphi \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$

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 (*propo-rew-one-step-lift* $\varphi \varphi' c \xi \xi'$)

note *rel = this(1)* and *IH = this(2)* and *wf = this(3)* and *no-T-F = this(4)*

moreover have *wf'*: *wf-conn* c ($\xi @ \varphi' \# \xi'$)

using *wf-conn-no-arity-change wf-conn-no-arity-change-helper wf* by *metis*

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

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

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

and *no-equiv* φ and *no-imp* φ and *no-T-F-except-top-level* φ

shows *no-equiv* ψ and *no-imp* ψ and *no-T-F-except-top-level* ψ

proof –

{

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

assume *rel*: *propo-rew-step pushNeg* $\varphi \psi$

and *no*: *no-T-F-except-top-level* φ

then have *no-T-F-except-top-level* ψ

proof –

{

assume $\varphi = FT \vee \varphi = FF$

from *rel* **this** **have** *False*

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

using *pushNeg.cases* **apply** *blast*

using *wf-conn-list(1) wf-conn-list(2)* **by** *auto*

then have *no-T-F-except-top-level* ψ **by** *blast*

}

moreover {

assume $\varphi \neq FT \wedge \varphi \neq FF$

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 *propo-rew-step-pushNeg-no-T-F rel* **by** *auto*

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

```

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

fixes $\varphi \psi :: 'v \text{ propo}$
assumes
 $\text{no-equiv } \varphi$ **and**
 $\text{no-imp } \varphi$ **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*

3.5.5 Push inside

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

for *c c'* :: '*v* *connective* **where**
push-conn-inside-l[simp]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
 \Longrightarrow *push-conn-inside c c' (conn c [conn c' [$\varphi 1$, $\varphi 2$], ψ])*
 $(conn c' [conn c [\varphi 1, ψ], conn c [\varphi 2, ψ]]) \mid$
push-conn-inside-r[simp]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
 \Longrightarrow *push-conn-inside c c' (conn c [ψ , conn c' [$\varphi 1$, $\varphi 2$]])*
 $(conn c' [conn c [ψ , $\varphi 1$], conn c [ψ , $\varphi 2$]])$

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

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

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

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

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

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

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

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

lemma *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 (*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 $H(1)$ *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** *not-c-in-c'-symb.intros(1)* H **by** *auto*
next
case (*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 $H(1)$ *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 *not-c-in-c'-symb.intros(2)* *conn-inj not-c-in-c'-symb-l.hyps*
 $\text{not-c-in-c'-symb-l.prem}(1,2)$ **by** *blast*
qed

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

lemma *not-c-in-c'-comm*:
assumes *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 $?A \longleftrightarrow (c\text{-in-c'-symb } c \ c' \ (\text{conn } c \ [\varphi, \psi]) \wedge (\forall \xi \in \text{set } [\varphi, \psi]. \text{all-subformula-st } (c\text{-in-c'-symb } c \ c' \ \xi)))$
using *all-subformula-st-decomp wf* **unfolding** *c-in-c'-only-def* **by** *fastforce*
also have $\dots \longleftrightarrow (c\text{-in-c'-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$

```

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

```

```

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

```

```

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

```

```

lemma c-in-c'-symb-step-exists:
  fixes  $\varphi :: 'v \text{ propo}$ 
  assumes c:  $c = C\text{And } \vee \ c = C\text{Or}$  and c':  $c' = C\text{And } \vee \ c' = C\text{Or}$ 
  shows  $\psi \preceq \varphi \implies \neg \text{c-in-c'-symb } c \ c' \ \psi \implies \exists \psi'. \text{push-conn-inside } c \ c' \ \psi \ \psi'$ 
  apply (induct  $\psi$  rule: propo-induct-arity)
  apply auto[2]
proof -
  fix  $\psi 1 \ \psi 2 \ \varphi' :: 'v \text{ propo}$ 
  assume IH $\psi 1$ :  $\psi 1 \preceq \varphi \implies \neg \text{c-in-c'-symb } c \ c' \ \psi 1 \implies \text{Ex } (\text{push-conn-inside } c \ c' \ \psi 1)$ 
  and IH $\psi 2$ :  $\psi 2 \preceq \varphi \implies \neg \text{c-in-c'-symb } c \ c' \ \psi 2 \implies \text{Ex } (\text{push-conn-inside } c \ c' \ \psi 2)$ 
  and  $\varphi'$ :  $\varphi' = F\text{And } \psi 1 \ \psi 2 \vee \varphi' = F\text{Or } \psi 1 \ \psi 2 \vee \varphi' = F\text{Imp } \psi 1 \ \psi 2 \vee \varphi' = F\text{Eq } \psi 1 \ \psi 2$ 
  and in $\varphi$ :  $\varphi' \preceq \varphi$  and n0:  $\neg \text{c-in-c'-symb } c \ c' \ \varphi'$ 
  then have n: not-c-in-c'-symb c c'  $\varphi'$  by auto
  {
    assume  $\varphi'$ :  $\varphi' = \text{conn } c [\psi 1, \psi 2]$ 
    obtain a b where  $\psi 1 = \text{conn } c' [a, b] \vee \psi 2 = \text{conn } c' [a, b]$ 
    using n  $\varphi'$  apply (induct rule: not-c-in-c'-symb.induct)
    using c by force+
    then have  $\text{Ex } (\text{push-conn-inside } c \ c' \ \varphi')$ 
    unfolding  $\varphi'$  apply auto
    using push-conn-inside.intros(1) c c' apply blast
  }

```

```

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

```

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

```

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

```

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

```

  fixes  $\varphi \ \psi :: 'v \text{ propo}$ 
  shows propo-rew-step (push-conn-inside c c')  $\varphi \ \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$ 
proof (induct rule: propo-rew-step.induct)
  case (global-rel  $\varphi \ \psi$ )
  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 rel = this(1) and IH = this(2) and wf = this(3) and no-T-F = this(4)
  have no-T-F  $\varphi$ 
    using wf no-T-F no-T-F-def subformula-into-subformula subformula-all-subformula-st
    subformula-refl by (metis (no-types) in-set-conv-decomp)
  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 wf no-T-F no-T-F-def all-subformula-st-decomp)
  then have  $n: \forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \text{no-T-F } \zeta$  using  $\varphi'$  by auto
  then have  $n': \forall \zeta \in \text{set } (\xi @ \varphi' \# \xi'). \zeta \neq FF \wedge \zeta \neq FT$ 

```

```

using  $\varphi'$  by (metis no-T-F-symb-false(1) no-T-F-symb-false(2) no-T-F-def
  all-subformula-st-test-symb-true-phi)

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

```

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

```

propo-rew-step (push-conn-inside  $c \ c'$ )  $\varphi \ \psi \implies \text{simple } \varphi \implies \text{simple } \psi$ 
apply (induct rule: propo-rew-step.induct)
apply (rename-tac  $\varphi$ , case-tac  $\varphi$ , 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  $\varphi \ \psi :: 'v$  propo
shows propo-rew-step (push-conn-inside  $c \ c'$ )  $\varphi \ \psi \implies \text{simple-not } \varphi \implies \text{simple-not } \psi$ 
proof (induct rule: propo-rew-step.induct)
  case (global-rel  $\varphi \ \psi$ )
  then show ?case by (cases  $\varphi$ , auto simp: push-conn-inside.simps)
next
  case (propo-rew-one-step-lift  $\varphi \ \varphi' \ ca \ \xi \ \xi'$ ) 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  $ca = \text{this}$ 
    obtain  $a \ b$  where  $ab: \xi @ \varphi' \# \xi' = [a, b]$ 
    using wf  $ca$  list-length2-decomp wf-conn-bin-list-length
    by (metis (no-types) wf-conn-no-arity-change-helper)
    have  $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \text{simple-not } \zeta$ 
    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 ( $\xi @ \varphi' \# \xi'$ )) using ca
by (metis ab conn.simps(5-8) helper-fact simple-not-symb.simps(5) simple-not-symb.simps(6)
    simple-not-symb.simps(7) simple-not-symb.simps(8))
ultimately show ?thesis
  by (simp add: ab all-subformula-st-decomp ca)
next
  case unary
  then show ?thesis
    using rew simple-propo-rew-step-push-conn-inside-inv[OF rew] IH local.wf simple by auto
qed
qed

```

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

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

assumes

propo-rew-step (*push-conn-inside* c c') $\varphi \varphi'$ **and**

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

simple-not-symb (conn c ($\xi @ \varphi \# \xi'$)) **and**

simple-not-symb φ'

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* = *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

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

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

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

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 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 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 $\varphi \implies$ all-subformula-st simple-not-symb φ'

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: $\xi = [] \wedge \xi' = []$ 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 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 $\varphi \implies$ all-subformula-st simple-not-symb φ'

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

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

and c: ca \in binary-connectives


```

have all-subformula-st simple-not-symb  $\varphi$ 
  using  $n$   $c$  corr all-subformula-st-decomp by fastforce
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  $c$   $c'$ )  $\varphi \varphi' \Longrightarrow \text{wf-conn } ca (\xi @ \varphi \# \xi')$ 
     $\Longrightarrow \text{simple-not-symb } (\text{conn } ca (\xi @ \varphi \# \xi')) \Longrightarrow \text{simple-not-symb } \varphi'$ 
     $\Longrightarrow \text{simple-not-symb } (\text{conn } ca (\xi @ \varphi' \# \xi'))$ 
  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-tolevel-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'  $\varphi \psi \implies$  no-equiv  $\varphi \implies$  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'  $\varphi \psi \implies$  no-imp  $\varphi \implies$  no-imp  $\psi$ 
    by (induct  $\varphi \psi$  rule: push-conn-inside.induct, auto)
}
then show no-imp  $\psi$ 
using full-propo-rew-step-inv-stay-conn[of push-conn-inside c c' no-imp-symb] assms
no-imp-symb-conn-characterization unfolding no-imp-def by metis

```

qed

lemma *push-conn-inside-full-propo-rew-step*:
fixes $\varphi \ \psi :: 'v \text{ propo}$
assumes
 no-equiv 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*

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* $c \ \varphi$ |
simple-cnot-only-c-inside[*simp*]: *simple* $\varphi \Longrightarrow$ *only-c-inside-symb* $c \ (FNot \ \varphi)$ |
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
 $\vee (\exists \ \varphi'. \ \psi = FNot \ \varphi' \wedge \text{simple } \varphi')$
 $\vee (\exists \ l. \ \psi = conn \ c \ l \wedge \text{wf-conn } c \ l)$)
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* $c \ (FNot \ \psi) \longleftrightarrow$ *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* $c \ (FNot \ \psi) \longleftrightarrow$ *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 $c \ \varphi \longleftrightarrow$
 $(\forall \psi. \ \psi \preceq \varphi \longrightarrow (\text{simple } \psi \vee (\exists \ \varphi'. \ \psi = FNot \ \varphi' \wedge \text{simple } \varphi')$
 $\vee (\exists \ l. \ \psi = conn \ c \ l \wedge \text{wf-conn } c \ l)))$
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 $c\ c' :: 'v\ connective$ **and** $l :: 'v\ propo\ list$ **and** $\varphi :: 'v\ propo$
assumes $cc': c \neq c'$ **and** $c: c = CAnd \vee c = COr$ **and** $c': c' = CAnd \vee c' = COr$
and *only*: *only-c-inside* $c\ \varphi$ **and** *incl*: *conn* $c'\ l \preceq \varphi$ **and** *wf*: *wf-conn* $c'\ l$
shows *False*

proof –

let $? \psi = \text{conn } c'\ l$
have *simple* $? \psi \vee (\exists \varphi'. ? \psi = FNot\ \varphi' \wedge \text{simple } \varphi') \vee (\exists l. ? \psi = \text{conn } c\ l \wedge \text{wf-conn } c\ l)$
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 $? \psi \neq FNot\ \varphi'$ **using** c' *conn-inj-not*(1) *wf* **by** *blast*
}
ultimately obtain $l :: 'v\ propo\ list$ **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** $c: c = CAnd \vee c = COr$ **and** $c': c' = CAnd \vee c' = COr$
shows *only-c-inside* $c\ \varphi \implies c\text{-in-}c'\text{-symb } c\ c'\ \varphi$
apply (*rule ccontr*)
apply (*cases rule: not-c-in-c'-symb.cases, auto*)
by (*metis* $\delta\ c\ c'$ *connective.distinct*(37,39) *list.distinct*(1) *only-c-inside-c-c'-false* *subformula-in-binary-conn*(1,2) *wf-conn.simps*)+

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

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

proof –

have *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 *wf-conn* $ca\ l \implies ca \neq c \implies c\text{-in-}c'\text{-symb } c\ c'\ (\text{conn } ca\ l)$ **by** *blast*

qed

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

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

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

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

using *inv*

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

case (*nullary x*)

```

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  $\varphi1$   $\varphi2$ )
note  $IH\varphi1 = this(1)$  and  $IH\varphi2 = this(2)$  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  $l: l = [\varphi1, \varphi2]$  by (meson wf-conn-list(4-7))
let ? $\varphi = conn\ c\ l$ 

obtain  $c1\ l1\ c2\ l2$  where  $\varphi1: \varphi1 = conn\ c1\ l1$  and  $wf\varphi1: wf-conn\ c1\ l1$ 
and  $\varphi2: \varphi2 = conn\ c2\ l2$  and  $wf\varphi2: wf-conn\ c2\ l2$  using exists-c-conn by metis
then have  $c-in-only\varphi1: c-in-c'-only\ c\ c' (conn\ c1\ l1)$  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  $inc\varphi1: \varphi1 \preceq ?\varphi$  and  $inc\varphi2: \varphi2 \preceq ?\varphi$ 
using  $\varphi1\ \varphi2\ \varphi\ local.wf$  by (metis conn.simps(5-8) helper-fact subformula-in-binary-conn(1,2))+

have  $c1-eq: c1 \neq CEq$  and  $c2-eq: c2 \neq CEq$ 
unfolding no-equiv-def using  $inc\varphi1\ inc\varphi2$  by (metis  $\varphi1\ \varphi2\ wf\varphi1\ wf\varphi2\ assms(1)\ no-equiv$ 
 $no-equiv-eq(1)\ no-equiv-symb.elims(3)\ no-equiv-symb-conn-characterization\ wf-conn-list(4,5)$ 
 $no-equiv-def\ subformula-all-subformula-st$ )+
have  $c1-imp: c1 \neq CImp$  and  $c2-imp: c2 \neq CImp$ 
using no-imp by (metis  $\varphi1\ \varphi2\ all-subformula-st-decomp-explicit-imp(2,3)\ assms(1)$ 
 $conn.simps(5,6)\ l\ no-imp-imp(1)\ no-imp-symb.elims(3)\ no-imp-symb-conn-characterization$ 
 $wf\varphi1\ wf\varphi2\ all-subformula-st-decomp\ no-imp-symb-conn-characterization$ )+
have  $c1c: c1 \neq c'$ 
proof
assume  $c1c: c1 = c'$ 
then obtain  $\xi1\ \xi2$  where  $l1: l1 = [\xi1, \xi2]$ 
by (metis assms(2) connective.distinct(37,39) helper-fact  $wf\varphi1\ wf-conn.simps$ 
 $wf-conn-list-decomp(1-3)$ )
have  $c-in-c'-only\ c\ c' (conn\ c\ [conn\ c'\ l1, \varphi2])$  using  $c1c\ l\ only\ \varphi1$  by auto
moreover have  $not-c-in-c'-symb\ c\ c' (conn\ c\ [conn\ c'\ l1, \varphi2])$ 
using  $l1\ \varphi1\ c1c\ l\ local.wf\ not-c-in-c'-symb-l\ wf\varphi1$  by blast
ultimately show False using  $\varphi1\ c1c\ l\ l1\ local.wf\ not-c-in-c'-simp(4)\ wf\varphi1$  by blast
qed
then have  $(\varphi1 = conn\ c\ l1 \wedge wf-conn\ c\ l1) \vee (\exists\psi1. \varphi1 = FNot\ \psi1) \vee simple\ \varphi1$ 
by (metis  $\varphi1\ assms(1-3)\ c1-eq\ c1-imp\ simple.elims(3)\ wf\varphi1\ wf-conn-list(4)\ wf-conn-list(5-7)$ )
moreover {
assume  $\varphi1 = conn\ c\ l1 \wedge wf-conn\ c\ l1$ 
then have only-c-inside  $c\ \varphi1$ 
by (metis  $IH\varphi1\ \varphi1\ all-subformula-st-decomp-imp\ inc\varphi1\ no-equiv\ no-equiv-def\ no-imp\ no-imp-def$ 
 $c-in-only\varphi1\ only-c-inside-def\ only-c-inside-into-only-c-inside\ simple-not\ simple-not-def$ 
 $subformula-all-subformula-st$ )
}
moreover {
assume  $\exists\psi1. \varphi1 = FNot\ \psi1$ 
then obtain  $\psi1$  where  $\varphi1 = FNot\ \psi1$  by metis
then have only-c-inside  $c\ \varphi1$ 
by (metis  $all-subformula-st-def\ assms(1)\ connective.distinct(37,39)\ inc\varphi1$ 
 $only-c-inside-decomp-not\ simple-not\ simple-not-def\ simple-not-symb.simps(1)$ )
}
moreover {
assume simple  $\varphi1$ 

```

```

then have only-c-inside  $c \varphi 1$ 
  by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
    only-c-inside-decomp-not only-c-inside-def)
}
ultimately have only-c-inside $\varphi 1$ : only-c-inside  $c \varphi 1$  by metis

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

```

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))*

Push Disjunction

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

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

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

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

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

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

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

using *assms push-conn-inside-full-propo-rew-step*
 unfolding *pushDisj-def or-in-and-only-def c-in-c'-only-def* by (*metis (no-types)*)

3.6 The full transformations

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

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* $\varphi \Longrightarrow$ *grouped-by* c φ |
simple-not-is-grouped[simp]: *simple* $\varphi \Longrightarrow$ *grouped-by* c (FNot φ) |
connected-is-group[simp]: *grouped-by* c $\varphi \Longrightarrow$ *grouped-by* c $\psi \Longrightarrow$ wf-conn c [φ , ψ]
 \Longrightarrow *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' [$\varphi 1$, $\varphi 2$]) \Longrightarrow $c' = CAnd \vee c' = COr \Longrightarrow$ wf-conn c' [$\varphi 1$, $\varphi 2$]
 \Longrightarrow $c' = c$
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' [$\varphi 1$, $\varphi 2$]) \Longrightarrow $c' = CAnd \vee c' = COr \Longrightarrow$ wf-conn c' [$\varphi 1$, $\varphi 2$] \Longrightarrow $c = c'$
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 $\varphi \Longrightarrow$ *grouped-by* c φ (**is** ?O $\varphi \Longrightarrow$?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 φ -conn: $\varphi = conn$ c [$\varphi 1$, $\varphi 2$] **and** wf: wf-conn c [$\varphi 1$, $\varphi 2$]
proof –
obtain c'' l'' **where** φ -c'': $\varphi = conn$ c'' l'' **and** wf: wf-conn c'' l''
using *exists-c-conn* **by** *metis*
then have l'': l'' = [$\varphi 1$, $\varphi 2$] **using** φ **by** (*metis* wf-conn-list(4-7))
have *only-c-inside-symb* c (conn c'' [$\varphi 1$, $\varphi 2$])


```

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

```

lemma grouped-by-false:

```

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

```

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

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

lemma simple-cnf[simp]:

```

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

```

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

```

assumes c:  $c = \text{CAnd } \vee c = \text{COr}$  and c':  $c' = \text{CAnd } \vee c' = \text{COr}$  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  $\varphi = \text{FT} \vee \varphi = \text{FF} \vee (\exists x. \varphi = \text{FVar } x)$  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 no-equiv = this(4) and no-imp = this(5)
and simpleN = this(6) and c-in-c'-only = this(7) and  $\varphi' = \text{this}(3)$ 
{
  assume  $\varphi = \text{FImp } \varphi 1 \ \varphi 2 \vee \varphi = \text{FEq } \varphi 1 \ \varphi 2$ 

```

```

    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: c-in-c'-only  $c \ c' \ \varphi 1 \wedge \text{c-in-c'-only } c \ c' \ \varphi 2 \wedge \text{c-in-c'-symb } c \ c' \ \varphi$ 
      using c-in-c'-only  $\varphi'$  unfolding c-in-c'-only-def by auto
    have super-grouped-by  $c \ c' \ \varphi 1$  using  $\varphi \ c'$  no-equiv no-imp simpleN IH  $\varphi 1$  c-in-c'-only by auto
    moreover have super-grouped-by  $c \ c' \ \varphi 2$ 
      using  $\varphi \ c'$  no-equiv no-imp simpleN IH  $\varphi 2$  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  $\varphi$  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$ 
```

3.6.2 Conjunctive Normal Form

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

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

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

definition *is-cnf* **where**

is-cnf $\varphi \equiv \text{is-conj-with-TF } \varphi \wedge \text{no-T-F-except-top-level } \varphi$

Full CNF transformation

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

definition *cnf-rew* **where** *cnf-rew* =

(*full (propo-rew-step elim-equiv)*) *OO*
(*full (propo-rew-step elim-imp)*) *OO*
(*full (propo-rew-step elimTB)*) *OO*
(*full (propo-rew-step pushNeg)*) *OO*
(*full (propo-rew-step pushDisj)*)

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

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

```

lemma cnf-rew-is-cnf: cnf-rew  $\varphi$   $\varphi' \implies$  is-cnf  $\varphi'$ 
  apply (unfold cnf-rew-def OO-def)
  apply auto
proof –
  fix  $\varphi$   $\varphiEq$   $\varphiImp$   $\varphiTB$   $\varphiNeg$   $\varphiDisj$  :: 'v propo
  assume Eq: full (propo-rew-step elim-equiv)  $\varphi$   $\varphiEq$ 
  then have no-equiv: no-equiv  $\varphiEq$  using no-equiv-full-propo-rew-step-elim-equiv by blast

  assume Imp: full (propo-rew-step elim-imp)  $\varphiEq$   $\varphiImp$ 
  then have no-imp: no-imp  $\varphiImp$  using no-imp-full-propo-rew-step-elim-imp by blast
  have no-imp-inv: no-equiv  $\varphiImp$  using no-equiv Imp elim-imp-inv by blast

  assume TB: full (propo-rew-step elimTB)  $\varphiImp$   $\varphiTB$ 
  then have noTB: no-T-F-except-top-level  $\varphiTB$ 
    using no-imp-inv no-imp elimTB-full-propo-rew-step by blast
  have noTB-inv: no-equiv  $\varphiTB$  no-imp  $\varphiTB$  using elimTB-inv TB no-imp no-imp-inv by blast+

  assume Neg: full (propo-rew-step pushNeg)  $\varphiTB$   $\varphiNeg$ 
  then have noNeg: simple-not  $\varphiNeg$ 
    using noTB-inv noTB pushNeg-full-propo-rew-step by blast
  have noNeg-inv: no-equiv  $\varphiNeg$  no-imp  $\varphiNeg$  no-T-F-except-top-level  $\varphiNeg$ 
    using pushNeg-inv Neg noTB noTB-inv by blast+

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

```

3.6.3 Disjunctive Normal Form

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

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

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

Full DNF transform

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

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

(full (propo-rew-step elimTB)) OO
 (full (propo-rew-step pushNeg)) OO
 (full (propo-rew-step pushConj))

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

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

theorem *dnf-transformation-correction:*

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

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

by (meson and-in-or-only-conjunction-in-disj elimTB-full-propo-rew-step elimTB-inv(1,2)

elim-imp-inv is-dnf-def no-equiv-full-propo-rew-step-elim-equiv

no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv(1-4)

pushNeg-full-propo-rew-step pushNeg-inv(1-3))

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

3.7.1 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 $\varphi \psi:: 'b$ propo

```

  have 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 *no-T-F-symb-except-toplevel-step-exists*, 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  $\psi$  rule: propo-induct-arity)
  case (nullary  $\varphi'$ )
  then have False using no-T-F-symb-except-toplevel-true no-T-F-symb-except-toplevel-false by auto
  then show Ex (elimTBFull  $\varphi'$ ) 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 Ex (elimTBFull (FNot  $\psi$ )) 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 Ex (elimTBFull  $\varphi'$ ) using elimTBFull.intros binary.hyps(3) by blast
qed

```

The same applies here. We do not need the assumption, but the deep link between $\neg \text{no-T-F-except-top-level}$ 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:
     $\forall x. \text{no-T-F-symb-except-toplevel } (FF :: 'v \text{ propo}) \wedge \text{no-T-F-symb-except-toplevel } FT$ 
     $\wedge \text{no-T-F-symb-except-toplevel } (FVar (x :: 'v))$ 
  by auto
  moreover {
    fix  $c :: 'v \text{ connective}$  and  $l :: 'v \text{ propo list}$  and  $\psi :: 'v \text{ propo}$ 
    have H:  $\text{elimTBFull } (\text{conn } c \ l) \ \psi \implies \neg \text{no-T-F-symb-except-toplevel } (\text{conn } c \ l)$ 
      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  $\varphi$  elimTBFull] noTB
    no-T-F-symb-except-toplevel-step-exists' unfolding no-T-F-except-top-level-def by metis
qed

```

```

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

```

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

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

```

fix  $\varphi' :: 'v \text{ propo}$  and  $\psi' :: 'v \text{ propo}$ 
assume a1:  $\text{no-T-F } \varphi'$ 
assume a2:  $\text{elim-equiv } \varphi' \ \psi'$ 
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  $\text{no-T-F } \psi' \text{ using } a1 \ a2 \text{ by fastforce}$ 

```

next

```

fix  $\varphi \ \varphi' :: 'v \text{ propo}$  and  $\xi \ \xi' :: 'v \text{ propo list}$  and  $c :: 'v \text{ connective}$ 
assume rel:  $\text{propo-rew-step elim-equiv } \varphi \ \varphi'$ 
and IH:  $\text{no-T-F } \varphi \implies \text{no-T-F } \varphi'$ 
and corr:  $\text{wf-conn } c \ (\xi @ \varphi \# \xi')$ 
and no-T-F:  $\text{no-T-F } (\text{conn } c \ (\xi @ \varphi \# \xi'))$ 
{
  assume c:  $c = \text{CNot}$ 
  then have empty:  $\xi = [] \ \xi' = []$  using corr by auto
  then have  $\text{no-T-F } \varphi$  using no-T-F c no-T-F-decomp-not by auto
  then have  $\text{no-T-F } (\text{conn } c \ (\xi @ \varphi' \# \xi'))$  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  $\varphi = a \vee \varphi = b$ 
  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 $\varphi': \text{no-T-F } \varphi' \text{ using } ab \text{ IH } \varphi \text{ by auto}$

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

moreover

have $\forall \zeta \in \text{set } (\xi @ \varphi \# \xi'). \zeta \neq \text{FT} \wedge \zeta \neq \text{FF}$

using $\zeta \text{ corr no-T-F no-T-F-except-top-level-false no-T-F-no-T-F-except-top-level}$ **by** *blast*

then have $\text{no-T-F-symb } (\text{conn } c \ (\xi @ \varphi' \# \xi'))$

by (*metis* $\varphi' \ l' \ ab \text{ all-subformula-st-test-symb-true-phi } c \text{ list.distinct(1)}$

$\text{list.set-intros(1,2) no-T-F-symb-except-toplevel-bin-decom}$

$\text{no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(1,2) no-T-F-def wf-conn-binary}$

$\text{wf-conn-list(1,2)})$

ultimately have $\text{no-T-F } (\text{conn } c \ (\xi @ \varphi' \# \xi'))$

```

    by (metis l' all-subformula-st-decomp-imp c no-T-F-def wf-conn-binary)
  }
  moreover {
    fix x
    assume c = CVar x  $\vee$  c = CF  $\vee$  c = CT
    then have False using corr by auto
    then have no-T-F (conn c ( $\xi$  @  $\varphi'$  #  $\xi'$ )) by auto
  }
  ultimately show no-T-F (conn c ( $\xi$  @  $\varphi'$  #  $\xi'$ )) using corr wf-conn.cases by metis
qed

lemma elim-equiv-inv':
  fixes  $\varphi \psi :: 'v$  propo
  assumes full (propo-rew-step elim-equiv)  $\varphi \psi$  and no-T-F-except-top-level  $\varphi$ 
  shows no-T-F-except-top-level  $\psi$ 
proof -
  {
    fix  $\varphi \psi :: 'v$  propo
    have propo-rew-step elim-equiv  $\varphi \psi \implies$  no-T-F-except-top-level  $\varphi$ 
       $\implies$  no-T-F-except-top-level  $\psi$ 
    proof -
      assume rel: propo-rew-step elim-equiv  $\varphi \psi$ 
      and no: no-T-F-except-top-level  $\varphi$ 
      {
        assume  $\varphi = FT \vee \varphi = FF$ 
        from rel this have False
        apply (induct rule: propo-rew-step.induct, auto simp: wf-conn-list(1,2))
        using elim-equiv.simps by blast+
        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 propo-rew-step-ElimEquiv-no-T-F rel by blast
        then have no-T-F-except-top-level  $\psi$  by (simp add: no-T-F-no-T-F-except-top-level)
      }
      ultimately show no-T-F-except-top-level  $\psi$  by metis
    qed
  }
  moreover {
    fix c :: 'v connective and  $\xi \xi' :: 'v$  propo list and  $\zeta \zeta' :: 'v$  propo
    assume rel: propo-rew-step elim-equiv  $\zeta \zeta'$ 
    and incl:  $\zeta \preceq \varphi$ 
    and corr: wf-conn c ( $\xi$  @  $\zeta$  #  $\xi'$ )
    and no-T-F: no-T-F-symb-except-toplevel (conn c ( $\xi$  @  $\zeta$  #  $\xi'$ ))
    and n: no-T-F-symb-except-toplevel  $\zeta'$ 
    have no-T-F-symb-except-toplevel (conn c ( $\xi$  @  $\zeta'$  #  $\xi'$ ))
    proof
      have p: no-T-F-symb (conn c ( $\xi$  @  $\zeta$  #  $\xi'$ ))
        using corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F
        by blast
      have l:  $\forall \varphi \in \text{set } (\xi @ \zeta \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$ 
        using corr wf-conn-no-T-F-symb-iff p by blast
      from rel incl have  $\zeta' \neq FT \wedge \zeta' \neq FF$ 
      apply (induction  $\zeta \zeta'$  rule: propo-rew-step.induct)

```

```

    apply (cases rule: elim-equiv.cases, auto simp: 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  $\forall \varphi \in \text{set } (\xi @ \zeta' \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$  using  $l$  by auto
    moreover have  $c \neq CT \wedge c \neq CF$  using  $\text{corr}$  by auto
    ultimately show  $\text{no-}T\text{-}F\text{-symb } (\text{conn } c (\xi @ \zeta' \# \xi'))$ 
    by (metis  $\text{corr}$  wf-conn-no-arity-change wf-conn-no-arity-change-helper no- $T\text{-}F\text{-symb-comp$ )
  qed
}
ultimately show  $\text{no-}T\text{-}F\text{-except-top-level } \psi$ 
using  $\text{full-propo-rew-step-inv-stay-with-inc}$  [of  $\text{elim-equiv no-}T\text{-}F\text{-symb-except-toplevel } \varphi$ ]
   $\text{assms subformula-refl unfolding no-}T\text{-}F\text{-except-top-level-def}$  by metis
qed

```

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

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

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

then show $\text{no-}T\text{-}F \psi'$

using *elim-imp.cases no-T-F-comp-not no-T-F-decomp*(1,2)

by (*metis no-T-F-comp-expanded-explicit*(2))

next

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

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

{

assume $c: c = CNot$

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

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

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

}

moreover {

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

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

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

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

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

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

then have $\varphi': \text{no-}T\text{-}F \varphi'$

using $\text{ab IH } \varphi \ \text{corr no-}T\text{-}F \ \text{no-}T\text{-}F\text{-def all-subformula-st-decomp-explicit}$ by auto

have $\chi: \xi @ \varphi' \# \xi' = [\varphi', b] \vee \xi @ \varphi' \# \xi' = [a, \varphi']$

by (*metis (no-types, hide-lams) ab append-Cons append-Nil append-Nil2 butlast.simps*(2) *butlast-append list.distinct*(1) *list.sel*(3))

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

moreover

have $\text{no-}T\text{-}F (\text{last } (\xi @ \varphi' \# \xi'))$ by (*simp add: calculation*)

then have $\text{no-}T\text{-}F\text{-symb } (\text{conn } c (\xi @ \varphi' \# \xi'))$

by (*metis* $\chi \ \varphi' \ \zeta \ \text{ab all-subformula-st-test-symb-true-phi } c \ \text{last.simps list.distinct}$ (1) *list.set-intros*(1) *no-T-F-bin-decomp no-T-F-def*)

ultimately have $\text{no-}T\text{-}F (\text{conn } c (\xi @ \varphi' \# \xi'))$ using $c \ \chi$ by *fastforce*

}

moreover {

fix x

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

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  $\forall \varphi \in \text{set } (\xi @ \zeta' \# \xi'). \varphi \neq FT \wedge \varphi \neq FF$  using l by auto
  moreover have  $c \neq CT \wedge c \neq CF$  using corr by auto
  ultimately show no-T-F-symb (conn c ( $\xi @ \zeta' \# \xi'$ ))
    using corr wf-conn-no-arity-change no-T-F-symb-comp
    by (metis wf-conn-no-arity-change-helper)
qed
}
ultimately show no-T-F-except-top-level  $\psi$ 
  using full-propo-rew-step-inv-stay-with-inc[of elim-imp no-T-F-symb-except-toplevel  $\varphi$ ]
  assms subformula-refl unfolding no-T-F-except-top-level-def by metis
qed

```

3.7.3 The new CNF and DNF transformation

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

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

```

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 $\text{dnf-rew}'\text{-consistent}$: preserves-un-sat $\text{dnf-rew}'$

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

theorem $\text{cnf-transformation-correction}$:

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

unfolding $\text{dnf-rew}'\text{-def}$ 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 $\text{cnf-rew}' :: 'a \text{ propo} \Rightarrow 'a \text{ propo} \Rightarrow \text{bool}$ **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 $\text{cnf-rew}'\text{-consistent}$: preserves-un-sat $\text{cnf-rew}'$

by (simp add: $\text{cnf-rew}'\text{-def}$ elimEqv-lifted-consistant elim-imp-lifted-consistant
elimTBFull-consistent preserves-un-sat-OO pushDisj-consistent pushNeg-lifted-consistant)

theorem $\text{cnf}'\text{-transformation-correction}$:

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

unfolding $\text{cnf-rew}'\text{-def}$ 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
theory *Prop-Logic-Multiset*
imports *../lib/Multiset-More Prop-Normalisation Partial-Clausal-Logic*
begin

3.8 Link with Multiset Version

3.8.1 Transformation to Multiset

fun *mset-of-conj* :: 'a propo \Rightarrow 'a literal multiset **where**
mset-of-conj (*FOr* φ ψ) = *mset-of-conj* φ + *mset-of-conj* ψ |
mset-of-conj (*FVar* v) = {# *Pos* v #} |
mset-of-conj (*FNot* (*FVar* v)) = {# *Neg* v #} |
mset-of-conj *FF* = {#}

fun *mset-of-formula* :: 'a propo \Rightarrow 'a literal multiset set **where**
mset-of-formula (*FAnd* φ ψ) = *mset-of-formula* φ \cup *mset-of-formula* ψ |
mset-of-formula (*FOr* φ ψ) = {*mset-of-conj* (*FOr* φ ψ)} |
mset-of-formula (*FVar* ψ) = {*mset-of-conj* (*FVar* ψ)} |
mset-of-formula (*FNot* ψ) = {*mset-of-conj* (*FNot* ψ)} |
mset-of-formula *FF* = {{#}} |
mset-of-formula *FT* = {}

3.8.2 Equisatisfiability of the two Version

lemma *is-conj-with-TF-FNot*:
is-conj-with-TF (*FNot* φ) \longleftrightarrow ($\exists v. \varphi = \text{FVar } v \vee \varphi = \text{FF} \vee \varphi = \text{FT}$)
unfolding *is-conj-with-TF-def* **apply** (*rule iffI*)
apply (*induction FNot* φ *rule: super-grouped-by.induct*)
apply (*induction FNot* φ *rule: grouped-by.induct*)
apply *simp*
apply (*cases* φ ; *simp*)
apply *auto*
done

lemma *grouped-by-COr-FNot*:
grouped-by COr (*FNot* φ) \longleftrightarrow ($\exists v. \varphi = \text{FVar } v \vee \varphi = \text{FF} \vee \varphi = \text{FT}$)
unfolding *is-conj-with-TF-def* **apply** (*rule iffI*)
apply (*induction FNot* φ *rule: grouped-by.induct*)
apply *simp*
apply (*cases* φ ; *simp*)
apply *auto*
done

lemma
shows *no-T-F-FF[simp]*: $\neg \text{no-T-F FF}$ **and**
no-T-F-FT[simp]: $\neg \text{no-T-F FT}$
unfolding *no-T-F-def all-subformula-st-def* **by** *auto*

lemma *grouped-by-CAnd-FAnd*:
grouped-by CAnd (*FAnd* φ_1 φ_2) \longleftrightarrow *grouped-by CAnd* $\varphi_1 \wedge$ *grouped-by CAnd* φ_2
apply (*rule iffI*)

apply (*induction* $FAnd\ \varphi1\ \varphi2$ *rule: grouped-by.induct*)
using *connected-is-group*[of $CAnd\ \varphi1\ \varphi2$] **by** *auto*

lemma *grouped-by-COr-FOr*:
 $grouped-by\ COr\ (FOr\ \varphi1\ \varphi2) \longleftrightarrow grouped-by\ COr\ \varphi1 \wedge grouped-by\ COr\ \varphi2$
apply (*rule iffI*)
apply (*induction* $FOr\ \varphi1\ \varphi2$ *rule: grouped-by.induct*)
using *connected-is-group*[of $COr\ \varphi1\ \varphi2$] **by** *auto*

lemma *grouped-by-COr-FAnd[simp]*: $\neg grouped-by\ COr\ (FAnd\ \varphi1\ \varphi2)$
apply *clarify*
apply (*induction* $FAnd\ \varphi1\ \varphi2$ *rule: grouped-by.induct*)
apply *auto*
done

lemma *grouped-by-COr-FEq[simp]*: $\neg grouped-by\ COr\ (FEq\ \varphi1\ \varphi2)$
apply *clarify*
apply (*induction* $FEq\ \varphi1\ \varphi2$ *rule: grouped-by.induct*)
apply *auto*
done

lemma [*simp*]: $\neg grouped-by\ COr\ (FImp\ \varphi\ \psi)$
apply *clarify*
by (*induction* $FImp\ \varphi\ \psi$ *rule: grouped-by.induct*) *simp-all*

lemma [*simp*]: $\neg is-conj-with-TF\ (FImp\ \varphi\ \psi)$
unfolding *is-conj-with-TF-def* **apply** *clarify*
by (*induction* $FImp\ \varphi\ \psi$ *rule: super-grouped-by.induct*) *simp-all*

lemma [*simp*]: $\neg grouped-by\ COr\ (FEq\ \varphi\ \psi)$
apply *clarify*
by (*induction* $FEq\ \varphi\ \psi$ *rule: grouped-by.induct*) *simp-all*

lemma [*simp*]: $\neg is-conj-with-TF\ (FEq\ \varphi\ \psi)$
unfolding *is-conj-with-TF-def* **apply** *clarify*
by (*induction* $FEq\ \varphi\ \psi$ *rule: super-grouped-by.induct*) *simp-all*

lemma *is-conj-with-TF-Fand*:
 $is-conj-with-TF\ (FAnd\ \varphi1\ \varphi2) \implies is-conj-with-TF\ \varphi1 \wedge is-conj-with-TF\ \varphi2$
unfolding *is-conj-with-TF-def*
apply (*induction* $FAnd\ \varphi1\ \varphi2$ *rule: super-grouped-by.induct*)
apply (*auto simp: grouped-by-CAnd-FAnd intro: grouped-is-super-grouped*)[]
apply *auto*[]
done

lemma *is-conj-with-TF-FOr*:
 $is-conj-with-TF\ (FOr\ \varphi1\ \varphi2) \implies grouped-by\ COr\ \varphi1 \wedge grouped-by\ COr\ \varphi2$
unfolding *is-conj-with-TF-def*
apply (*induction* $FOr\ \varphi1\ \varphi2$ *rule: super-grouped-by.induct*)
apply (*auto simp: grouped-by-COr-FOr*)[]
apply *auto*[]
done

lemma *grouped-by-COr-mset-of-formula*:
 $grouped-by\ COr\ \varphi \implies mset-of-formula\ \varphi = (if\ \varphi = FT\ then\ \{\} \ else\ \{mset-of-conj\ \varphi\})$

by (induction φ) (auto simp add: grouped-by-COr-FNot)

When a formula is in CNF form, then there is equisatisfiability between the multiset version and the CNF form. Remark that the definition for the entailment are slightly different: $op \models$ uses a function assigning *True* or *False*, while $op \models s$ uses a set where being in the list means entailment of a literal.

theorem

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

assumes *is-cnf* φ

shows $eval\ A\ \varphi \longleftrightarrow Partial\text{-}Clausal\text{-}Logic.true\text{-}class\ (\{Pos\ v|v. A\ v\} \cup \{Neg\ v|v. \neg A\ v\})$
(*mset-of-formula* φ)

using *assms*

proof (induction φ)

case *FF*

then show ?case by auto

next

case *FT*

then show ?case by auto

next

case (*FVar* v)

then show ?case by auto

next

case (*FAnd* $\varphi\ \psi$)

then show ?case

unfolding *is-cnf-def* by (auto simp: *is-conj-with-TF-FNot* dest: *is-conj-with-TF-Fand*
dest!: *is-conj-with-TF-FOr*)

next

case (*FOr* $\varphi\ \psi$)

then have [*simp*]: *mset-of-formula* $\varphi = \{mset\text{-of-conj}\ \varphi\}$ *mset-of-formula* $\psi = \{mset\text{-of-conj}\ \psi\}$
unfolding *is-cnf-def* by (auto dest!: *is-conj-with-TF-FOr* simp: *grouped-by-COr-mset-of-formula*
split: if-splits)

have *is-conj-with-TF* φ *is-conj-with-TF* ψ

using *FOr*(3) unfolding *is-cnf-def* no-*T-F-def*

by (*metis* *grouped-is-super-grouped* *is-conj-with-TF-FOr* *is-conj-with-TF-def*)+

then show ?case using *FOr*

unfolding *is-cnf-def* by *simp*

next

case (*FImp* $\varphi\ \psi$)

then show ?case

unfolding *is-cnf-def* by auto

next

case (*FEq* $\varphi\ \psi$)

then show ?case

unfolding *is-cnf-def* by auto

next

case (*FNot* φ)

then show ?case

unfolding *is-cnf-def* by (auto simp: *is-conj-with-TF-FNot*)

qed

end

theory *Prop-Resolution*

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

begin

Chapter 4

Resolution-based techniques

This chapter contains the formalisation of resolution and superposition.

4.1 Resolution

4.1.1 Simplification Rules

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

tautology-deletion:

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

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-clss-singleton true-clss-union true-lit-def uminus-Neg union-commute*)

then show ?case **by** (*metis Un-Diff-cancel2 true-clss-singleton true-clss-union*)

next

case (*condensation* $A\ P$)

then show ?case **by** (*metis Diff-insert-absorb Set.set-insert insertE true-clss-union true-clss-def true-clss-singleton true-clss-union*)

next

case (*subsumption* $A\ B$)

have $A \neq B$ **using** *subsumption.hyps*(2) **by** *auto*

then have $I \models_s N - \{B\} \implies I \models A$ **using** $\langle A \in N \rangle$ **by** (*simp add: true-clss-def*)

moreover have $I \models A \implies I \models B$ **using** $\langle A \subset\# B \rangle$ **by** *auto*

ultimately show ?case **by** (*metis insert-Diff-single true-clss-insert*)

qed

lemma *simplify-preserves-un-sat*:

fixes $N\ N' ::$ 'v clauses

```

assumes simplify  $N\ N'$ 
and total-over-m  $I\ N$ 
shows  $I \models_s N \longrightarrow I \models_s N'$ 
using assms apply (induct rule: simplify.induct)
using true-clss-def by fastforce+
```

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

```

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

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

```

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

lemma *simplify-preserves-finite*:

```

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

lemma *rtranclp-simplify-preserves-finite*:

```

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

lemma *simplify-atms-of-ms*:

```

assumes simplify  $\psi\ \psi'$ 
shows atms-of-ms  $\psi' \subseteq \text{atms-of-ms } \psi$ 
using assms unfolding atms-of-ms-def
proof (induct rule: simplify.induct)
  case (tautology-deletion  $A\ P$ )
  then show ?case by auto
next
  case (condensation  $A\ P$ )
  moreover have  $A + \{\#P\} + \{\#P\} \in \psi \implies \exists x \in \psi. \text{atm-of } P \in \text{atm-of 'set-mset } x$ 
    by (metis Un-iff atms-of-def atms-of-plus atms-of-singleton insert-iff)
  ultimately show ?case by (auto simp add: atms-of-def)
next
  case (subsumption  $A\ P$ )
  then show ?case by auto
qed
```

lemma *rtranclp-simplify-atms-of-ms*:

```

assumes rtranclp simplify  $\psi\ \psi'$ 
shows atms-of-ms  $\psi' \subseteq \text{atms-of-ms } \psi$ 
using assms apply (induct rule: rtranclp-induct)
  apply (fastforce intro: simplify-atms-of-ms)
using simplify-atms-of-ms by blast
```

lemma *factoring-imp-simplify*:

assumes $\{\#L\# \} + \{\#L\# \} + C \in N$
shows $\exists N'. \text{ simplify } N N'$
proof –
have $C + \{\#L\# \} + \{\#L\# \} \in N$ **using** *assms* **by** (*simp add: add.commute union-lcomm*)
from *condensation[OF this]* **show** *?thesis* **by** *blast*
qed

4.1.2 Unconstrained Resolution

type-synonym *'v uncon-state* = *'v clauses*
inductive *uncon-res* :: *'v uncon-state* \Rightarrow *'v uncon-state* \Rightarrow *bool* **where**
resolution:
 $\{\#Pos\ p\# \} + C \in N \implies \{\#Neg\ p\# \} + D \in N \implies (\{\#Pos\ p\# \} + C, \{\#Neg\ p\# \} + D) \notin \text{already-used}$
 $\implies \text{uncon-res } (N) (N \cup \{C + D\}) \mid$
factoring: $\{\#L\# \} + \{\#L\# \} + C \in N \implies \text{uncon-res } N (N \cup \{C + \{\#L\# \}\})$

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

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

Subsumption

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

lemma *subsumes-refl[simp]*:
subsumes $\chi\ \chi$
unfolding *subsumes-def* **by** *auto*

lemma *subsumes-subsumption*:
assumes *subsumes* $D\ \chi$
and $C \subset\# D$ **and** $\neg \text{tautology } \chi$
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*)

4.1.3 Inference Rule

type-synonym *'v state* = *'v clauses* \times (*'v clause* \times *'v clause*) *set*
inductive *inference-clause* :: *'v state* \Rightarrow *'v clause* \times (*'v clause* \times *'v clause*) *set* \Rightarrow *bool*
(infix \Rightarrow_{Res} 100) **where**

resolution:

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

inductive *inference* :: 'v state \Rightarrow 'v state \Rightarrow bool **where**
inference-step: *inference-clause* *S* (*clause*, *already-used*)
 $\implies \text{inference } S (\text{fst } S \cup \{\text{clause}\}, \text{already-used})$

abbreviation *already-used-inv*

:: 'a literal multiset set \times ('a literal multiset \times 'a literal multiset) set \Rightarrow bool **where**
already-used-inv state \equiv
 $(\forall (A, B) \in \text{snd state}. \exists p. \text{Pos } p \in\# A \wedge \text{Neg } p \in\# B \wedge$
 $((\exists \chi \in \text{fst state}. \text{subsumes } \chi ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\})))$
 $\vee \text{tautology } ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))))$

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

assumes *inference-clause* *S S'*
and *already-used-inv* *S*
shows *already-used-inv* (*fst* *S* \cup {*fst* *S'*}, *snd* *S'*)
using *assms* **apply** (*induct* rule: *inference-clause.induct*)
by *fastforce*+

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

assumes *inference* *S S'*
and *already-used-inv* *S*
shows *already-used-inv* *S'*
using *assms*

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

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

then show ?*case*

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

qed

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

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

lemma *subsumes-condensation:*

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

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

assumes *simplify* *N N'*
and *already-used-inv* (*N*, *already-used*)
shows *already-used-inv* (*N'*, *already-used*)
using *assms*

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

case (*condensation* *C L*)

```

then show ?case
  using subsumes-condensation by simp fast
next
{
  fix a:: 'a and A :: 'a set and P
  have  $(\exists x \in \text{Set.remove } a \ A. P \ x) \longleftrightarrow (\exists x \in A. x \neq a \wedge P \ x)$  by auto
} note ex-member-remove = this
{
  fix a a0 :: 'v clause and A :: 'v clauses and y
  assume  $a \in A$  and  $a0 \subset\# a$ 
  then have  $(\exists x \in A. \text{subsumes } x \ y) \longleftrightarrow (\text{subsumes } a \ y \vee (\exists x \in A. x \neq a \wedge \text{subsumes } x \ y))$ 
    by auto
} note tt2 = 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  $x: x \in \text{snd } (N - \{B\}, \text{already-used})$  and [simp]:  $x = (a, b)$ 
  obtain p where  $p: \text{Pos } p \in\# a \wedge \text{Neg } p \in\# b$  and
     $q: (\exists \chi \in N. \text{subsumes } \chi (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
     $\vee \text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\}))$ 
  using inv x by fastforce
  consider (taut)  $\text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})) \mid$ 
     $(\chi) \chi$  where  $\chi \in N$   $\text{subsumes } \chi (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\}))$ 
     $\neg \text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\}))$ 
  using q by auto
  then show
     $\exists p. \text{Pos } p \in\# a \wedge \text{Neg } p \in\# b$ 
     $\wedge ((\exists \chi \in \text{fst } (N - \{B\}, \text{already-used}). \text{subsumes } \chi (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
     $\vee \text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
  proof cases
    case taut
    then show ?thesis using p by auto
  next
    case  $\chi$  note H = this
    show ?thesis using p A AB B subsumes-subsumption[OF - AB H(3)] H(1,2) by auto
  qed
qed
next
case (tautology-deletion C P)
then show ?case apply clarify
proof -
  fix a b
  assume  $C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\} \in N$ 
  assume  $\text{already-used-inv } (N, \text{already-used})$ 
  and  $(a, b) \in \text{snd } (N - \{C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}\}, \text{already-used})$ 
  then obtain p where
     $\text{Pos } p \in\# a \wedge \text{Neg } p \in\# b \wedge$ 
     $((\exists \chi \in \text{fst } (N \cup \{C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}\}, \text{already-used}).$ 
     $\text{subsumes } \chi (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
     $\vee \text{tautology } (a - \{\# \text{Pos } p\} + (b - \{\# \text{Neg } p\})))$ 
  by fastforce
  moreover have  $\text{tautology } (C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\})$  by auto
  ultimately show
     $\exists p. \text{Pos } p \in\# a \wedge \text{Neg } p \in\# b$ 
     $\wedge ((\exists \chi \in \text{fst } (N - \{C + \{\# \text{Pos } P\} + \{\# \text{Neg } P\}\}, \text{already-used}).$ 

```

$$\text{subsumes } \chi (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \}))$$

$$\vee \text{tautology } (a - \{\#Pos\ p\# \} + (b - \{\#Neg\ p\# \}))$$
by (*metis* (*no-types*) *Diff-iff* *Un-insert-right* *empty-iff* *fst-conv* *insertE* *subsumes-tautology* *sup-bot.right-neutral*)

qed

qed

lemma

factoring-satisfiable: $I \models \{\#L\# \} + \{\#L\# \} + C \longleftrightarrow I \models \{\#L\# \} + C$ **and**
resolution-satisfiable:
consistent-interp $I \implies I \models \{\#Pos\ p\# \} + C \implies I \models \{\#Neg\ p\# \} + D \implies I \models C + D$ **and**
factoring-same-vars: $\text{atms-of } (\{\#L\# \} + \{\#L\# \} + C) = \text{atms-of } (\{\#L\# \} + C)$
unfolding *true-cls-def* *consistent-interp-def* **by** (*fastforce* *split*: *if-split-asm*)**+**

lemma *inference-increasing*:

assumes *inference* $S\ S'$ **and** $\psi \in \text{fst } S$
shows $\psi \in \text{fst } S'$
using *assms* **by** (*induct* *rule*: *inference.induct*, *auto*)

lemma *rtranclp-inference-increasing*:

assumes *rtranclp inference* $S\ S'$ **and** $\psi \in \text{fst } S$
shows $\psi \in \text{fst } S'$
using *assms* **by** (*induct* *rule*: *rtranclp-induct*, *auto* *simp* *add*: *inference-increasing*)

lemma *inference-clause-already-used-increasing*:

assumes *inference-clause* $S\ S'$
shows $\text{snd } S \subseteq \text{snd } S'$
using *assms* **by** (*induct* *rule*: *inference-clause.induct*, *auto*)

lemma *inference-already-used-increasing*:

assumes *inference* $S\ S'$
shows $\text{snd } S \subseteq \text{snd } S'$
using *assms* **apply** (*induct* *rule*: *inference.induct*)
using *inference-clause-already-used-increasing* **by** *fastforce*

lemma *inference-clause-preserves-un-sat*:

fixes $N\ N' :: 'v\ \text{clauses}$
assumes *inference-clause* $T\ T'$
and *total-over-m* $I\ (\text{fst } T)$
and *consistent*: *consistent-interp* I
shows $I \models_s \text{fst } T \longleftrightarrow I \models_s \text{fst } T \cup \{\text{fst } T'\}$
using *assms* **apply** (*induct* *rule*: *inference-clause.induct*)
unfolding *consistent-interp-def* *true-clss-def* **by** *auto* *force***+**

lemma *inference-preserves-un-sat*:

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

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

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

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

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference* $T T'$
shows $\text{total-over-m } I (\text{fst } T) \implies \text{total-over-m } I (\text{fst } T')$
using *assms* **by** (*induct rule: rtranclp-induct, auto simp add: inference-preserves-total*)

lemma *rtranclp-inference-preserves-un-sat*:
assumes *rtranclp inference* $N N'$
and $\text{total-over-m } I (\text{fst } N)$
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* $(\text{fst } \psi)$
shows *finite* $(\text{fst } \psi')$
using *assms* **by** (*induct rule: inference.induct, auto simp add: simplify-preserves-finite*)

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

lemma *inference-preserves-finite-snd*:
assumes *inference* $\psi \psi'$ **and** *finite* $(\text{snd } \psi)$
shows *finite* $(\text{snd } \psi')$
using *assms* *inference-clause-preserves-finite-snd* **by** (*induct rule: inference.induct, fastforce*)

```

lemma rtrancplp-inference-preserves-finite:
  assumes rtrancplp inference  $\psi$   $\psi'$  and finite (fst  $\psi$ )
  shows finite (fst  $\psi'$ )
  using assms by (induct rule: rtrancplp-induct)
  (auto simp add: simplify-preserves-finite inference-preserves-finite)

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

lemma simplify-clause-preserves-sat:
  assumes simp: simplify  $\psi$   $\psi'$ 
  and satisfiable  $\psi'$ 
  shows satisfiable  $\psi$ 
  using assms
proof induction
  case (tautology-deletion A P) note AP = this(1) and sat = this(2)
  let ?A' = A + {#Pos P#} + {#Neg P#}
  let ? $\psi'$  =  $\psi$  - {?A'}
  obtain I where
    I: I  $\models$  s ? $\psi'$  and
    cons: consistent-interp I and
    tot: total-over-m I ? $\psi'$ 
  using sat unfolding satisfiable-def by auto
  { assume Pos P  $\in$  I  $\vee$  Neg P  $\in$  I
    then have I  $\models$  ?A' by auto
    then have I  $\models$  s  $\psi$  using I by (metis insert-Diff tautology-deletion.hyps true-clss-insert)
    then have ?case using cons tot by auto
  }
  moreover {
    assume Pos: Pos P  $\notin$  I and Neg: Neg P  $\notin$  I
    then have consistent-interp (I  $\cup$  {Pos P}) using cons by simp
    moreover have I'A: I  $\cup$  {Pos P}  $\models$  ?A' by auto
    have {Pos P}  $\cup$  I  $\models$  s  $\psi$  - {A + {#Pos P#} + {#Neg P#}}
      using ⟨I  $\models$  s  $\psi$  - {A + {#Pos P#} + {#Neg P#}}⟩ true-clss-union-increase' by blast
    then have I  $\cup$  {Pos P}  $\models$  s  $\psi$ 
      by (metis (no-types) Un-empty-right Un-insert-left Un-insert-right I'A insert-Diff
        sup-bot.left-neutral tautology-deletion.hyps true-clss-insert)
    ultimately have ?case using satisfiable-carac' by blast
  }
  ultimately show ?case by blast
next
  case (condensation A L) note AL = this(1) and sat = this(2)
  have f3: simplify  $\psi$  ( $\psi$  - {A + {#L#} + {#L#}}  $\cup$  {A + {#L#}})
    using AL simplify.condensation by blast
  obtain LL :: 'a literal multiset set  $\Rightarrow$  'a literal set where
    f4: LL ( $\psi$  - {A + {#L#} + {#L#}}  $\cup$  {A + {#L#}})  $\models$  s  $\psi$  - {A + {#L#} + {#L#}}  $\cup$  {A
  + {#L#}}

```

```

     $\wedge$  consistent-interp (LL ( $\psi - \{A + \{\#L\# \} + \{\#L\#\} \cup \{A + \{\#L\#\}\}$ ))
     $\wedge$  total-over-m (LL ( $\psi - \{A + \{\#L\#\} + \{\#L\#\}$ 
       $\cup \{A + \{\#L\#\}\}$ )) ( $\psi - \{A + \{\#L\#\} + \{\#L\#\} \cup \{A + \{\#L\#\}\}$ ))
    using sat by (meson satisfiable-def)
  have f5: insert ( $A + \{\#L\#\} + \{\#L\#\}$ ) ( $\psi - \{A + \{\#L\#\} + \{\#L\#\}\}$ ) =  $\psi$ 
    using AL by fastforce
  have atms-of ( $A + \{\#L\#\} + \{\#L\#\}$ ) = atms-of ( $\{\#L\#\} + A$ )
    by simp
  then show ?case
    using f5 f4 f3 by (metis (no-types) add.commute satisfiable-def simplify-preserves-un-sat'
      total-over-m-insert total-over-m-union)
next
case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and sat = this(4)
let ? $\psi'$  =  $\psi - \{B\}$ 
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]:

```

( $\bigwedge xs:: 'v \text{ sem-tree. } (\bigwedge ys:: 'v \text{ sem-tree. } \text{sem-tree-size } ys < \text{sem-tree-size } xs \implies P \ ys) \implies P \ xs$ )
 $\implies P \ xs$ 
by (fact Nat.measure-induct-rule)

```

fun partial-interps :: 'v sem-tree \Rightarrow 'v interp \Rightarrow 'v clauses \Rightarrow bool **where**

```

partial-interps Leaf I  $\psi$  = ( $\exists \chi. \neg I \models \chi \wedge \chi \in \psi \wedge \text{total-over-m } I \ \{\chi\}$ ) |
partial-interps (Node v ag ad) I  $\psi \longleftrightarrow$ 
  (partial-interps ag (I  $\cup \{\text{Pos } v\}$ )  $\psi \wedge \text{partial-interps ad (I} \cup \{\text{Neg } v\}$ )  $\psi$ )

```

lemma simplify-preserve-partial-leaf:

```

simplify N N'  $\implies$  partial-interps Leaf I N  $\implies$  partial-interps Leaf I N'
apply (induct rule: simplify.induct)

```

```

    using union-lcomm apply auto[1]
    apply (simp, metis atms-of-plus total-over-set-union true-cls-union)
apply simp
by (metis atms-of-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  $t\ I\ (\text{fst } S)$ 
  shows partial-interps  $t\ I\ (\text{fst } S')$ 
  using assms apply (induct  $t$  arbitrary:  $I$ , simp-all)
  by (meson inference-increasing)

```

```

lemma rtranclp-inference-preserve-partial-tree:
  assumes rtranclp inference  $N\ N'$ 
  and partial-interps  $t\ I\ (\text{fst } N)$ 
  shows partial-interps  $t\ I\ (\text{fst } N')$ 
  using assms apply (induct rule: rtranclp-induct, auto)
  using inference-preserve-partial-tree by force

```

```

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

```

```

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

```

```

lemma partial-interps-build-sem-tree-atms-general:
  fixes  $\psi :: 'v :: linorder$  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:  $\bigwedge \chi. \chi \in \psi \implies \text{total-over-m } Ia \{ \chi \}$ 
      unfolding total-over-m-def total-over-set-def atms-of-ms-def atms-of-s-def
      using atmsIa atms-of-ms-def by fastforce
    have partial-interps Leaf Ia  $\psi$ 
      using  $\chi$  tot by (auto simp add: total-over-m-def total-over-set-def atms-of-ms-def)

    ultimately have ?case by metis
  }
moreover {
  assume atms: atms  $\neq \{\}$ 
  have build-sem-tree atms  $\psi$  = Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    using build-sem-tree.simps[of atms  $\psi$ ] f atms by metis

  have consistent-interp (Ia  $\cup \{Pos (Min\ atms)\}$ ) unfolding consistent-interp-def
    by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
      f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2)
      uminus-Neg uminus-Pos)
  moreover have atms-of-ms  $\psi$  = Set.remove (Min atms) atms  $\cup$  atms-of-s (Ia  $\cup \{Pos (Min\ atms)\}$ )
    using Min-in atms f un by fastforce
  moreover have disj': Set.remove (Min atms) atms  $\cap$  atms-of-s (Ia  $\cup \{Pos (Min\ atms)\}$ ) = {}
    by simp (metis disj disjoint-iff-not-equal member-remove)
  moreover have finite (Set.remove (Min atms) atms) using f by (simp add: remove-def)
  ultimately have subtree1: partial-interps (build-sem-tree (Set.remove (Min atms) atms)  $\psi$ )
    (Ia  $\cup \{Pos (Min\ atms)\}$ )  $\psi$ 
    using IH1[of Ia  $\cup \{Pos (Min\ (atms))\}$ ] atms f unsat finite by metis

  have consistent-interp (Ia  $\cup \{Neg (Min\ atms)\}$ ) unfolding consistent-interp-def
    by (metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff
      f in-atms-of-s-decomp insert-iff literal.distinct(1) literal.exhaust-sel literal.sel(2)
      uminus-Neg)
  moreover have atms-of-ms  $\psi$  = Set.remove (Min atms) atms  $\cup$  atms-of-s (Ia  $\cup \{Neg (Min\ atms)\}$ )
    using  $\langle$ 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 $p :: 'v \text{ literal list}$

assumes *unsat*: unsatisfiable ψ and *finite*: finite ψ

shows partial-interps (build-sem-tree (atms-of-ms ψ) ψ) $\{\}$ ψ

proof –

have consistent-interp $\{\}$ unfolding consistent-interp-def by auto

moreover have atms-of-ms $\psi = \text{atms-of-ms } \psi \cup \text{atms-of-s } \{\}$ unfolding atms-of-s-def by auto

moreover have atms-of-ms $\psi \cap \text{atms-of-s } \{\} = \{\}$ unfolding atms-of-s-def by auto

moreover have finite (atms-of-ms ψ) unfolding atms-of-ms-def using finite by simp

ultimately show partial-interps (build-sem-tree (atms-of-ms ψ) ψ) $\{\}$ ψ

using partial-interps-build-sem-tree-atms-general[of $\psi \{\}$ atms-of-ms ψ] assms by metis

qed

lemma can-decrease-count:

fixes $\psi'' :: 'v \text{ clauses} \times ('v \text{ clause} \times 'v \text{ clause} \times 'v) \text{ set}$

assumes count $\chi \ L = n$

and $L \in \# \chi$ and $\chi \in \text{fst } \psi$

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

$\wedge \text{count } \chi' \ L = 1$

$\wedge (\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi')$

$\wedge (I \models \chi \longleftrightarrow I \models \chi')$

$\wedge (\forall I'. \text{total-over-m } I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi'\})$

using assms

proof (induct n arbitrary: $\chi \psi$)

case 0

then show ?case by (simp add: not-in-iff[symmetric])

next

case (Suc n χ)

note IH = this(1) and count = this(2) and L = this(3) and $\chi = \text{this}(4)$

{

assume $n = 0$

then have inference** $\psi \psi$

and $\chi \in \text{fst } \psi$

and $\forall L. (L \in \# \chi) \longleftrightarrow (L \in \# \chi)$

and count $\chi \ L = (1::\text{nat})$

and $\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi$

by (auto simp add: count L χ)

then have ?case by metis

}

moreover {

assume $n > 0$

then have $\exists C. \chi = C + \{\#L, L\# \}$

by (smt L Suc-eq-plus1-left add.left-commute add-diff-cancel-left' add-diff-cancel-right'
count-greater-zero-iff count-single local.count multi-member-split plus-multiset.rep-eq)

then obtain C where $C: \chi = C + \{\#L, L\# \}$ by metis

let $? \chi' = C + \{\#L\# \}$

let $? \psi' = (\text{fst } \psi \cup \{? \chi'\}, \text{snd } \psi)$

have $\varphi: \forall \varphi \in \text{fst } \psi. (\varphi \in \text{fst } \psi \vee \varphi \neq ? \chi') \longleftrightarrow \varphi \in \text{fst } ? \psi'$ unfolding C by auto

have inf: inference $\psi ? \psi'$

using C factoring χ prod.collapse union-commute inference-step by metis

moreover have count': count $? \chi' \ L = n$ using C count by auto

moreover have $L \chi': L \in \# ? \chi'$ by auto

moreover have $\chi' \psi'$: $? \chi' \in \text{fst } ? \psi'$ **by auto**
ultimately obtain ψ'' **and** χ''
where
*inference*** $? \psi' \psi''$ **and**
 α : $\chi'' \in \text{fst } \psi''$ **and**
 $\forall La. (La \in \# ? \chi') \longleftrightarrow (La \in \# \chi'')$ **and**
 β : $\text{count } \chi'' L = (1::\text{nat})$ **and**
 φ' : $\forall \varphi. \varphi \in \text{fst } ? \psi' \longrightarrow \varphi \in \text{fst } \psi''$ **and**
 $I\chi$: $I \models ? \chi' \longleftrightarrow I \models \chi''$ **and**
 tot : $\forall I'. \text{total-over-m } I' \{? \chi'\} \longrightarrow \text{total-over-m } I' \{\chi''\}$
using $IH[\text{of } ? \chi' ? \psi'] \text{ count' } L\chi' \chi' \psi'$ **by blast**

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

}
ultimately show $? \text{case}$ **by auto**
qed

lemma *can-decrease-tree-size*:
fixes $\psi :: 'v \text{ state}$ **and** $\text{tree} :: 'v \text{ sem-tree}$
assumes *finite* ($\text{fst } \psi$) **and** *already-used-inv* ψ
and *partial-interps tree I* ($\text{fst } \psi$)
shows $\exists (\text{tree}' :: 'v \text{ sem-tree}) \psi'. \text{inference** } \psi \psi' \wedge \text{partial-interps tree' } I (\text{fst } \psi')$
 $\wedge (\text{sem-tree-size tree}' < \text{sem-tree-size tree} \vee \text{sem-tree-size tree} = 0)$
using *assms*

proof (*induct arbitrary: I rule: sem-tree-size*)
case (*bigger xs I*) **note** $IH = \text{this}(1)$ **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 $? \text{case}$ **using** *part* **by blast**
}

moreover {
assume $\text{sn0}: \text{sem-tree-size } xs > 0$
obtain $ag \text{ ad } v$ **where** $xs: xs = \text{Node } v \text{ ag ad}$ **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 $\text{Pos}v: \text{Pos } v \notin \# \chi$ **using** χ **unfolding** *true-cls-def true-lit-def* **by auto**
have $\text{Neg}v: \text{Neg } v \notin \# \chi'$ **using** χ' **unfolding** *true-cls-def true-lit-def* **by auto**

```

{
  assume Negχ: Neg v ∉# χ
  have ¬ I ⊨ χ using χ Posv unfolding true-cls-def true-lit-def by auto
  moreover have total-over-m I {χ}
    using Posv Negχ atm-imp-pos-or-neg-lit totχ unfolding total-over-m-def total-over-set-def
    by fastforce
  ultimately have partial-interps Leaf I (fst ψ)
  and sem-tree-size Leaf < sem-tree-size xs
  and inference** ψ ψ
    unfolding xs by (auto simp add: χψ)
}
moreover {
  assume Posχ: Pos v ∉# χ'
  then have Iχ: ¬ I ⊨ χ' using χ' Posv unfolding true-cls-def true-lit-def by auto
  moreover have total-over-m I {χ'}
    using Negv Posχ atm-imp-pos-or-neg-lit totχ' unfolding total-over-m-def total-over-set-def by fastforce
  ultimately have partial-interps Leaf I (fst ψ) and
    sem-tree-size Leaf < sem-tree-size xs and
    inference** ψ ψ
    using χ'ψ Iχ unfolding xs by auto
}
moreover {
  assume neg: Neg v ∈# χ and pos: Pos v ∈# χ'
  then obtain ψ' χ2 where inf: rtrnclp inference ψ ψ' and χ2incl: χ2 ∈ fst ψ'
    and χχ2incl: ∀ L. L ∈# χ ↔ L ∈# χ2
    and countχ2: count χ2 (Neg v) = 1
    and φ: ∀ φ::'v literal multiset. φ ∈ fst ψ → φ ∈ fst ψ'
    and Iχ: I ⊨ χ ↔ I ⊨ χ2
    and tot-impχ: ∀ I'. total-over-m I' {χ} → total-over-m I' {χ2}
    using can-decrease-count[of χ Neg v count χ (Neg v) ψ I] χψ χ'ψ by auto

  have χ' ∈ fst ψ' by (simp add: χ'ψ φ)
  with pos
  obtain ψ'' χ2' where
    inf': inference** ψ' ψ''
    and χ2'-incl: χ2' ∈ fst ψ''
    and χ'χ2'-incl: ∀ L::'v literal. (L ∈# χ') = (L ∈# χ2')
    and countχ2': count χ2' (Pos v) = (1::nat)
    and φ': ∀ φ::'v literal multiset. φ ∈ fst ψ' → φ ∈ fst ψ''
    and Iχ': I ⊨ χ' ↔ I ⊨ χ2'
    and tot-impχ': ∀ I'. total-over-m I' {χ'} → total-over-m I' {χ2'}
    using can-decrease-count[of χ' Pos v count χ' (Pos v) ψ' I] by auto

  obtain C where χ2: χ2 = C + {#Neg v#} and negC: Neg v ∉# C and posC: Pos v ∉# C
  proof -
    have ∧m. Suc 0 - count m (Neg v) = count (χ2 - m) (Neg v)
      by (simp add: countχ2)
    then show ?thesis
      using that by (metis (no-types) One-nat-def Posv Suc-inject Suc-pred χχ2-incl
        count-diff count-single insert-DiffM2 mem-Collect-eq multi-member-skip neg
        not-gr0 set-mset-def union-commute)
  qed

  obtain C' where
    χ2': χ2' = C' + {#Pos v#} and

```

```

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

have already-used-inv  $\psi'$ 
  using rtranclp-inference-preserves-already-used-inv[of  $\psi\ \psi'$ ] a-u-i inf by blast
then have a-u-i- $\psi''$ : already-used-inv  $\psi''$ 
  using rtranclp-inference-preserves-already-used-inv a-u-i inf' unfolding tautology-def
  by simp

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

```

```

    )
    ∨ tautology (({#Pos v#} + C') - {#Pos p#} + (({#Neg v#} + C) - {#Neg p#})))
using a by (blast intro: allE[OF a-u-i-ψ''[unfolding subsumes-def Ball-def],
    of ({#Pos v#} + C', {#Neg v#} + C)])
{ assume p ≠ v
  then have Pos p ∈# C' ∧ Neg p ∈# C using p n by force
  then have ?thesis unfolding Bex-def by auto
}
moreover {
  assume p = v
  then have ?thesis using decomp by (metis add.commute add-diff-cancel-left')
}
ultimately show ?thesis by auto
qed
moreover {
assume ∃χ ∈ fst ψ''. (∀I. total-over-m I {C+C'} → total-over-m I {χ})
  ∧ (∀I. total-over-m I {χ} → I ⊨ χ → I ⊨ C' + C)
then obtain ∅ where ∅: ∅ ∈ fst ψ'' and
  tot-∅-CC': ∀I. total-over-m I {C+C'} → total-over-m I {∅} and
  ∅-inv: ∀I. total-over-m I {∅} → I ⊨ ∅ → I ⊨ C' + C by blast
have partial-interps Leaf I (fst ψ'')
  using tot-∅-CC' ∅ ∅-inv totC totC' (⊢ I ⊨ C + C') total-over-m-sum by fastforce
moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
ultimately have ?case by (metis inf inf' rtranclp-trans)
}
moreover {
assume tautCC': tautology (C' + C)
have total-over-m I {C'+C} using totC totC' total-over-m-sum by auto
then have ¬tautology (C' + C)
  using (⊢ I ⊨ C + C') unfolding add.commute[of C C'] total-over-m-def
  unfolding tautology-def by auto
then have False using tautCC' unfolding tautology-def by auto
}
ultimately have ?case by auto
}
ultimately have ?case by auto
}
ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
}
moreover {
assume size-ag: sem-tree-size ag > 0
have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
moreover have partial-interps ag (I ∪ {Pos v}) (fst ψ)
  and partad: partial-interps ad (I ∪ {Neg v}) (fst ψ)
  using part partial-interps.simps(2) unfolding xs by metis+
moreover have sem-tree-size ag < sem-tree-size xs → finite (fst ψ) → already-used-inv ψ
  → ( partial-interps ag (I ∪ {Pos v}) (fst ψ) →
    (∃ tree' ψ'. inference** ψ ψ' ∧ partial-interps tree' (I ∪ {Pos v}) (fst ψ')
      ∧ (sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0)))
  using IH by auto
ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
  inf: inference** ψ ψ'
  and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')
  and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
  using finite part rtranclp.rtrancl-refl a-u-i by blast

```

```

have partial-interps ad ( $I \cup \{Neg\ v\}$ ) (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  $\psi :: 'v :: linorder$  state
assumes
  unsat:  $\neg$ satisfiable (fst  $\psi$ ) and
  finite: finite (fst  $\psi$ ) and
  a-u-v: already-used-inv  $\psi$ 
shows  $\exists \psi'. (inference** \psi\ \psi' \wedge \{\#\} \in fst\ \psi')$ 
proof -
  obtain tree where partial-interps tree {} (fst  $\psi$ )
  using partial-interps-build-sem-tree-atms assms by metis
  then show ?thesis
  using unsat finite a-u-v
  proof (induct tree arbitrary:  $\psi$  rule: sem-tree-size)
    case (bigger tree  $\psi$ ) note H = this
    {
      fix  $\chi$ 
      assume tree: tree = Leaf
      obtain  $\chi$  where  $\chi: \neg \{\} \models \chi$  and tot $\chi$ : total-over-m {} { $\chi$ } and  $\chi\psi$ :  $\chi \in fst\ \psi$ 
      using H unfolding tree by auto
      moreover have { $\#\}$  =  $\chi$ 
      using tot $\chi$  unfolding total-over-m-def total-over-set-def by fastforce
      moreover have inference**  $\psi\ \psi$  by auto
      ultimately have ?case by metis
    }
  }

```

```

moreover {
  fix  $v$   $tree1$   $tree2$ 
  assume  $tree: tree = Node\ v\ tree1\ tree2$ 
  obtain
     $tree'\ \psi'$  where  $inf: inference^{**}\ \psi\ \psi'$  and
     $part': partial\text{-}interp\ tree'\ \{\}$   $(fst\ \psi')$  and
     $decrease: sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ tree \vee sem\text{-}tree\text{-}size\ tree = 0$ 
    using  $can\text{-}decrease\text{-}tree\text{-}size[of\ \psi]\ H(2,4,5)$  unfolding  $tautology\text{-}def$  by  $meson$ 
  have  $sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ tree$  using  $decrease$  unfolding  $tree$  by  $auto$ 
  moreover have  $finite\ (fst\ \psi')$  using  $rtranclp\text{-}inference\text{-}preserves\text{-}finite\ inf\ H(4)$  by  $metis$ 
  moreover have  $unsatisfiable\ (fst\ \psi')$ 
    using  $inference\text{-}preserves\text{-}unsat\ inf\ bigger.prem\ s(2)$  by  $blast$ 
  moreover have  $already\text{-}used\text{-}inv\ \psi'$ 
    using  $H(5)\ inf\ rtranclp\text{-}inference\text{-}preserves\text{-}already\text{-}used\text{-}inv[of\ \psi\ \psi']$  by  $auto$ 
  ultimately have  $?case$  using  $inf\ rtranclp\text{-}trans\ part'\ H(1)$  by  $fastforce$ 
}
ultimately show  $?case$  by  $(cases\ tree,\ auto)$ 
qed
qed

```

```

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

```

```

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

```

```

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

```

4.1.4 Lemma about the simplified state

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

```

lemma  $simplified\text{-}count$ :
  assumes  $simp: simplified\ \psi$  and  $\chi: \chi \in \psi$ 
  shows  $count\ \chi\ L \leq 1$ 
proof –
  {
    let  $? \chi' = \chi - \{\#L, L\#\}$ 
    assume  $count\ \chi\ L \geq 2$ 

```



```

then have f1: count ( $\chi - \{\#L, L\# \} + \{\#L, L\# \}$ )  $L = \text{count } \chi \ L$ 
  by simp
then have  $L \in \# \chi - \{\#L\# \}$ 
  by (metis (no-types) add.left-neutral add-diff-cancel-left' count-union diff-diff-add
    diff-single-trivial insert-DiffM mem-Collect-eq multi-member-this not-gr0 set-mset-def)
then have  $\chi'$ :  $?\chi' + \{\#L\# \} + \{\#L\# \} = \chi$ 
  using f1 by (metis diff-diff-add diff-single-eq-union in-diffD)

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

lemma simplified-no-both:
  assumes simp: simplified  $\psi$  and  $\chi$ :  $\chi \in \psi$ 
  shows  $\neg (L \in \# \chi \wedge \neg L \in \# \chi)$ 
proof (rule ccontr)
  assume  $\neg \neg (L \in \# \chi \wedge \neg L \in \# \chi)$ 
  then have  $L \in \# \chi \wedge \neg L \in \# \chi$  by metis
  then obtain  $\chi'$  where  $\chi = \chi' + \{\#Pos \ (atm\text{-of } L)\# \} + \{\#Neg \ (atm\text{-of } L)\# \}$ 
    by (metis Neg-atm-of-iff Pos-atm-of-iff diff-union-swap insert-DiffM2 uminus-Neg uminus-Pos)
  then show False using  $\chi$  simp tautology-deletion by fastforce
qed

lemma simplified-not-tautology:
  assumes simplified  $\{\psi\}$ 
  shows  $\sim \text{tautology } \psi$ 
proof (rule ccontr)
  assume  $\sim ?thesis$ 
  then obtain  $p$  where  $Pos \ p \in \# \psi \wedge Neg \ p \in \# \psi$  using tautology-decomp by metis
  then obtain  $\chi$  where  $\psi = \chi + \{\#Pos \ p\# \} + \{\#Neg \ p\# \}$ 
    by (metis insert-noteq-member literal.distinct(1) multi-member-split)
  then have  $\sim \text{simplified } \{\psi\}$  by (auto intro: tautology-deletion)
  then show False using assms by auto
qed

lemma simplified-remove:
  assumes simplified  $\{\psi\}$ 
  shows simplified  $\{\psi - \{\#l\# \}\}$ 
proof (rule ccontr)
  assume ns:  $\neg \text{simplified } \{\psi - \{\#l\# \}\}$ 
  {
    assume  $l \notin \# \psi$ 
    then have  $\psi - \{\#l\# \} = \psi$  by simp
    then have False using ns assms by auto
  }
  moreover {
    assume  $l\psi$ :  $l \in \# \psi$ 
    have  $A$ :  $\bigwedge A. A \in \{\psi - \{\#l\# \}\} \longleftrightarrow A + \{\#l\# \} \in \{\psi\}$  by (auto simp add:  $l\psi$ )
    obtain  $l'$  where  $l'$ : simplify  $\{\psi - \{\#l\# \}\}$   $l'$  using ns by metis
    then have  $\exists l'. \text{simplify } \{\psi\} \ l'$ 
    proof (induction rule: simplify.induct)
      case (tautology-deletion  $A \ P$ )
      have  $\{\#Neg \ P\# \} + (\{\#Pos \ P\# \} + (A + \{\#l\# \})) \in \{\psi\}$ 

```

```

    by (metis (no-types) A add.commute tautology-deletion.hyps union-lcomm)
  then show ?thesis
    by (metis simplify.tautology-deletion[of A+{#l#} P {ψ}] add.commute)
next
  case (condensation A L)
  have A + {#L#} + {#L#} + {#l#} ∈ {ψ}
    using A condensation.hyps by blast
  then have {#L, L#} + (A + {#l#}) ∈ {ψ}
    by (metis (no-types) union-assoc union-commute)
  then show ?case
    using factoring-imp-simplify by blast
next
  case (subsumption A B)
  then show ?case by blast
qed
then have False using assms(1) by blast
}
ultimately show False by auto
qed

```

lemma *in-simplified-simplified*:

```

  assumes simp: simplified ψ and incl: ψ' ⊆ ψ
  shows simplified ψ'
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain ψ'' where simplify ψ' ψ'' by metis
  then have ∃ l'. simplify ψ l'
    proof (induction rule: simplify.induct)
    case (tautology-deletion A P)
    then show ?thesis using simplify.tautology-deletion[of A P ψ] incl by blast
  next
    case (condensation A L)
    then show ?case using simplify.condensation[of A L ψ] incl by blast
  next
    case (subsumption A B)
    then show ?case using simplify.subsumption[of A ψ B] incl by auto
  qed
  then show False using assms(1) by blast
qed

```

lemma *simplified-in*:

```

  assumes simplified ψ
  and N ∈ ψ
  shows simplified {N}
  using assms by (metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono)

```

lemma *subsumes-imp-formula*:

```

  assumes ψ ≤# φ
  shows {ψ} ⊨p φ
  unfolding true-clss-cls-def apply auto
  using assms true-clss-mono-leD by blast

```

lemma *simplified-imp-distinct-mset-tauto*:

```

  assumes simp: simplified ψ'
  shows distinct-mset-set ψ' and ∀χ ∈ ψ'. ¬tautology χ

```

```

proof –
  show  $\forall \chi \in \psi'. \neg \text{tautology } \chi$ 
    using simp by (auto simp add: simplified-in simplified-not-tautology)

  show distinct-mset-set  $\psi'$ 
    proof (rule ccontr)
      assume  $\neg ?thesis$ 
      then obtain  $\chi$  where  $\chi \in \psi'$  and  $\neg \text{distinct-mset } \chi$  unfolding distinct-mset-set-def by auto
      then obtain  $L$  where  $\text{count } \chi \ L \geq 2$ 
        unfolding distinct-mset-def
        by (meson count-greater-eq-one-iff le-antisym simp simplified-count)
      then show False by (metis Suc-1 'χ ∈ ψ' 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)

```

4.1.5 Resolution and Invariants

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

Invariants

```

lemma resolution-finite:
  assumes resolution  $\psi \ \psi'$  and finite (fst  $\psi$ )
  shows finite (fst  $\psi'$ )
  using assms by (induct rule: resolution.induct)
    (auto simp add: full1-def full-def rtranclp-simplify-preserves-finite
      dest: tranclp-into-rtranclp inference-preserves-finite)

lemma rtranclp-resolution-finite:
  assumes resolution**  $\psi \ \psi'$  and finite (fst  $\psi$ )
  shows finite (fst  $\psi'$ )
  using assms by (induct rule: rtranclp-induct, auto simp add: resolution-finite)

lemma resolution-finite-snd:
  assumes resolution  $\psi \ \psi'$  and finite (snd  $\psi$ )
  shows finite (snd  $\psi'$ )
  using assms apply (induct rule: resolution.induct, auto simp add: inference-preserves-finite-snd)
  using inference-preserves-finite-snd snd-conv by metis

lemma rtranclp-resolution-finite-snd:
  assumes resolution**  $\psi \ \psi'$  and finite (snd  $\psi$ )
  shows finite (snd  $\psi'$ )
  using assms by (induct rule: rtranclp-induct, auto simp add: resolution-finite-snd)

lemma resolution-always-simplified:
  assumes resolution  $\psi \ \psi'$ 
  shows simplified (fst  $\psi'$ )
  using assms by (induct rule: resolution.induct)

```

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

lemma *tranclp-resolution-always-simplified:*

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

shows *simplified (fst ψ')*

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

lemma *resolution-atms-of:*

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

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

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

apply(*simp add: rtranclp-simplify-atms-of-ms tranclp-into-rtranclp full1-def*)

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

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

lemma *rtranclp-resolution-atms-of:*

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

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

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

using *resolution-atms-of rtranclp-resolution-finite by blast+*

lemma *resolution-include:*

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

shows *fst $\psi' \subseteq$ simple-clss (atms-of-ms (fst ψ))*

proof –

have *finite': finite (fst ψ') using local.finite res resolution-finite by blast*

have *simplified (fst ψ') using res finite' resolution-always-simplified by blast*

then have *fst $\psi' \subseteq$ simple-clss (atms-of-ms (fst ψ'))*

using *simplified-in-simple-clss finite' simplified-imp-distinct-mset-tauto[of fst ψ'] by auto*

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

using *res finite resolution-atms-of[of $\psi \psi'$] by auto*

ultimately show *?thesis by (meson atms-of-ms-finite local.finite order.trans rev-finite-subset simple-clss-mono)*

qed

lemma *rtranclp-resolution-include:*

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

shows *fst $\psi' \subseteq$ simple-clss (atms-of-ms (fst ψ))*

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

apply (*simp add: resolution-include*)

by (*meson simple-clss-mono order-class.le-trans resolution-include*

rtranclp-resolution-atms-of rtranclp-resolution-finite tranclp-into-rtranclp)

abbreviation *already-used-all-simple*

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

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

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

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

assumes *vars \subseteq vars'*

shows *already-used-all-simple a vars \implies already-used-all-simple a vars'*

using *assms by fast*

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

assumes *inference-clause S S'*

and *already-used-all-simple (snd S) vars*

```

and simplified (fst S)
and atms-of-ms (fst S)  $\subseteq$  vars
shows already-used-all-simple (snd (fst S  $\cup$  {fst S'}, snd S')) vars
using assms
proof (induct rule: inference-clause.induct)
case (factoring L C N already-used)
then show ?case by (simp add: simplified-in factoring-imp-simplify)
next
case (resolution P C N D already-used) note H = this
show ?case apply clarify
proof -
fix A B v
assume (A, B)  $\in$  snd (fst (N, already-used))
 $\cup$  {fst (C + D, already-used  $\cup$  {({#Pos P#} + C, {#Neg P#} + D)})},
snd (C + D, already-used  $\cup$  {({#Pos P#} + C, {#Neg P#} + D)})}
then have (A, B)  $\in$  already-used  $\vee$  (A, B) = ({#Pos P#} + C, {#Neg P#} + D) by auto
moreover {
assume (A, B)  $\in$  already-used
then have simplified {A}  $\wedge$  simplified {B}  $\wedge$  atms-of A  $\subseteq$  vars  $\wedge$  atms-of B  $\subseteq$  vars
using H(4) by auto
}
moreover {
assume eq: (A, B) = ({#Pos P#} + C, {#Neg P#} + D)
then have simplified {A} using simplified-in H(1,5) by auto
moreover have simplified {B} using eq simplified-in H(2,5) by auto
moreover have atms-of A  $\subseteq$  atms-of-ms N
using eq H(1)
using atms-of-atms-of-ms-mono[of A N] by auto
moreover have atms-of B  $\subseteq$  atms-of-ms N
using eq H(2) atms-of-atms-of-ms-mono[of B N] by auto
ultimately have simplified {A}  $\wedge$  simplified {B}  $\wedge$  atms-of A  $\subseteq$  vars  $\wedge$  atms-of B  $\subseteq$  vars
using H(6) by auto
}
ultimately show simplified {A}  $\wedge$  simplified {B}  $\wedge$  atms-of A  $\subseteq$  vars  $\wedge$  atms-of B  $\subseteq$  vars
by fast
qed
qed

```

```

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

```

```

lemma already-used-all-simple-inv:
assumes resolution S S'
and already-used-all-simple (snd S) vars
and atms-of-ms (fst S)  $\subseteq$  vars

```

```

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

lemma rtrancplp-already-used-all-simple-inv:
  assumes resolution** S S'
  and already-used-all-simple (snd S) vars
  and atms-of-ms (fst S)  $\subseteq$  vars
  and finite (fst S)
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: rtrancplp-induct)
  case base
  then show ?case by simp
next
  case (step S' S'')
  note infstar = this(1) and IH = this(3) and res = this(2) and
    already = this(4) and atms = this(5) and finite = this(6)
  have already-used-all-simple (snd S') vars using IH already atms finite by simp
  moreover have atms-of-ms (fst S')  $\subseteq$  atms-of-ms (fst S)
    by (simp add: infstar local.finite rtrancplp-resolution-atms-of)
  then have atms-of-ms (fst S')  $\subseteq$  vars using atms by auto
  ultimately show ?case
    using already-used-all-simple-inv[OF res] by simp
qed

lemma inference-clause-simplified-already-used-subset:
  assumes inference-clause S S'
  and simplified (fst S)
  shows snd S  $\subset$  snd S'
  using assms apply (induct rule: inference-clause.induct, auto)
  using factoring-imp-simplify by blast

lemma inference-simplified-already-used-subset:
  assumes inference S S'
  and simplified (fst S)
  shows snd S  $\subset$  snd S'
  using assms apply (induct rule: inference.induct)
  by (metis inference-clause-simplified-already-used-subset snd-conv)

lemma resolution-simplified-already-used-subset:
  assumes resolution S S'
  and simplified (fst S)
  shows snd S  $\subset$  snd S'
  using assms apply (induct rule: resolution.induct, simp-all add: full1-def)
  apply (meson trancplpD)
  by (metis inference-simplified-already-used-subset fst-conv snd-conv)

lemma trancplp-resolution-simplified-already-used-subset:
  assumes trancplp resolution S S'

```

and *simplified* (*fst* *S*)
shows *snd* *S* \subset *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 vars* **and** *finite vars*
shows *s* \subseteq *already-used-top vars*

proof

fix *x*
assume *x-s*: *x* \in *s*
obtain *A B* **where** *x*: *x* = (*A*, *B*) **by** (*cases* *x*, *auto*)
then have *simplified* {*A*} **and** *atms-of* *A* \subseteq *vars* **using** *assms*(1) *x-s* **by** *fastforce*+
then have *A*: *A* \in *simple-clss vars*
 using *simple-clss-mono*[*of* *atms-of* *A vars*] *x* *assms*(2)
 simplified-imp-distinct-mset-tauto[*of* {*A*}]
 distinct-mset-not-tautology-implies-in-simple-clss **by** *fast*
moreover have *simplified* {*B*} **and** *atms-of* *B* \subseteq *vars* **using** *assms*(1) *x-s* *x* **by** *fast*+
then have *B*: *B* \in *simple-clss vars*
 using *simplified-imp-distinct-mset-tauto*[*of* {*B*}]
 distinct-mset-not-tautology-implies-in-simple-clss
 simple-clss-mono[*of* *atms-of* *B vars*] *x* *assms*(2) **by** *fast*
ultimately show *x* \in *simple-clss vars* \times *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 *var* \subseteq *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 literal multiset \times 'a literal multiset) set **and** *vars* :: 'a set
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *finite s*
using *assms* *already-used-all-simple-in-already-used-top*[*OF* *assms*(1)]
rev-finite-subset[*OF* *already-used-top-finite*[*of* *vars*]] **by** *auto*

abbreviation *card-simple vars* $\psi \equiv$ *card* (*already-used-top vars* $- \psi$)

lemma *resolution-card-simple-decreasing*:

assumes *res*: *resolution* ψ ψ'
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 $\text{atms-of-ms } (fst \ \psi) \subseteq \text{vars}$
shows $\text{card-simple vars } (snd \ \psi') < \text{card-simple vars } (snd \ \psi)$
proof –
let $?vars = \text{vars}$
let $?top = \text{simple-clss } ?vars \times \text{simple-clss } ?vars$
have 1: $\text{card-simple vars } (snd \ \psi) = \text{card } ?top - \text{card } (snd \ \psi)$
using $\text{card-Diff-subset finite-snd already-used-all-simple-in-already-used-top}[OF \ a-u-s]$
 finite-v **by** metis
have $a-u-s'$: $\text{already-used-all-simple } (snd \ \psi') \ \text{vars}$
using $\text{already-used-all-simple-inv res a-u-s assms}(7)$ **by** blast
have f : $\text{finite } (snd \ \psi')$ **using** $\text{already-used-all-simple-finite a-u-s' finite-v}$ **by** auto
have 2: $\text{card-simple vars } (snd \ \psi') = \text{card } ?top - \text{card } (snd \ \psi')$
using $\text{card-Diff-subset}[OF \ f]$ $\text{already-used-all-simple-in-already-used-top}[OF \ a-u-s' \ \text{finite-v}]$
by auto
have $\text{card } (\text{already-used-top vars}) \geq \text{card } (snd \ \psi')$
using $\text{already-used-all-simple-in-already-used-top}[OF \ a-u-s' \ \text{finite-v}]$
 $\text{card-mono}[of \ \text{already-used-top vars } snd \ \psi']$ $\text{already-used-top-finite}[OF \ \text{finite-v}]$ **by** metis
then show $?thesis$
using $\text{psubset-card-mono}[OF \ f \ \text{resolution-simplified-already-used-subset}[OF \ res \ \text{simp}]]$
unfolding 1 2 **by** linarith
qed

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

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

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

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

well-foundness if the relation

lemma *wf-simplified-resolution*:

assumes *f-vars*: *finite vars*
shows *wf* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (\text{fst } x) \subseteq \text{vars} \wedge \text{simplified } (\text{fst } x) \wedge \text{finite } (\text{snd } x) \wedge \text{already-used-all-simple } (\text{snd } x) \text{ vars}) \wedge \text{resolution } x y\}$
proof –
{
fix *a b* :: *'v::linorder state*
assume $(b, a) \in \{(y, x). (\text{atms-of-ms } (\text{fst } x) \subseteq \text{vars} \wedge \text{simplified } (\text{fst } x) \wedge \text{finite } (\text{snd } x) \wedge \text{already-used-all-simple } (\text{snd } x) \text{ vars}) \wedge \text{resolution } x y\}$
then have
atms-of-ms (*fst* *a*) \subseteq *vars* **and**
simp: *simplified* (*fst* *a*) **and**
finite (*snd* *a*) **and**
finite (*fst* *a*) **and**
a-u-v: *already-used-all-simple* (*snd* *a*) *vars* **and**
res: *resolution* *a b* **by** *auto*
have *finite* (*already-used-top vars*) **using** *f-vars* *already-used-top-finite* **by** *blast*
moreover **have** *already-used-top vars* \subseteq *already-used-top vars* **by** *auto*
moreover **have** *snd b* \subseteq *already-used-top vars*
using *already-used-all-simple-in-already-used-top[of* *snd b vars* *]*
a-u-v *already-used-all-simple-inv[OF res]* (*finite* (*fst* *a*)) (*atms-of-ms* (*fst* *a*) \subseteq *vars*) *f-vars*
by *presburger*
moreover **have** *snd a* \subset *snd b* **using** *resolution-simplified-already-used-subset[OF res simp]* .
ultimately have *finite* (*already-used-top vars*) \wedge *already-used-top vars* \subseteq *already-used-top vars*
 \wedge *snd b* \subseteq *already-used-top vars* \wedge *snd a* \subset *snd b* **by** *metis*
}
then show *?thesis* **using** *wf-bounded-set[of* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (\text{fst } x) \subseteq \text{vars} \wedge \text{simplified } (\text{fst } x) \wedge \text{finite } (\text{snd } x) \wedge \text{finite } (\text{fst } x) \wedge \text{already-used-all-simple } (\text{snd } x) \text{ vars}) \wedge \text{resolution } x y\}$ $\lambda\cdot. \text{already-used-top vars snd}$ *]* **by** *auto*
qed

lemma *wf-simplified-resolution'*:

assumes *f-vars*: *finite vars*
shows *wf* $\{(y:: 'v:: \text{linorder state}, x). (\text{atms-of-ms } (\text{fst } x) \subseteq \text{vars} \wedge \neg \text{simplified } (\text{fst } x) \wedge \text{finite } (\text{snd } x) \wedge \text{finite } (\text{fst } x) \wedge \text{already-used-all-simple } (\text{snd } x) \text{ vars}) \wedge \text{resolution } x y\}$
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 *rtranclp-simplify-already-used-inv*:
assumes *simplify*^{**} *S S'*
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S'*, *N*)
using *assms* **apply** *induction*
using *simplify-preserves-already-used-inv* **by** *fast+*

lemma *full1-simplify-already-used-inv*:
assumes *full1 simplify S S'*
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S'*, *N*)
using *assms* *trancplp-into-rtranclp*[*of simplify S S'*] *rtranclp-simplify-already-used-inv*
unfolding *full1-def* **by** *fast*

lemma *full-simplify-already-used-inv*:
assumes *full simplify S S'*
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S'*, *N*)
using *assms* *rtranclp-simplify-already-used-inv* **unfolding** *full-def* **by** *fast*

lemma *resolution-already-used-inv*:
assumes *resolution S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms*

proof *induction*
case (*full1-simp N N' already-used*)
then show $?case$ **using** *full1-simplify-already-used-inv* **by** *fast*

next
case (*inferring N already-used N' already-used' N''*) **note** $inf = this(1)$ **and** $full = this(3)$ **and** $a-u-v = this(4)$
then show $?case$
using *inference-preserves-already-used-inv*[*OF inf a-u-v*] *full-simplify-already-used-inv full*
by *fast*

qed

lemma *rtranclp-resolution-already-used-inv*:
assumes *resolution*^{**} *S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms* **apply** *induction*
using *resolution-already-used-inv* **by** *fast+*

```

lemma rtanclp-simplify-preserves-unsat:
  assumes simplify**  $\psi$   $\psi'$ 
  shows satisfiable  $\psi' \longrightarrow$  satisfiable  $\psi$ 
  using assms apply induction
  using simplify-clause-preserves-sat by blast

lemma full1-simplify-preserves-unsat:
  assumes full1 simplify  $\psi$   $\psi'$ 
  shows satisfiable  $\psi' \longrightarrow$  satisfiable  $\psi$ 
  using assms rtanclp-simplify-preserves-unsat[of  $\psi$   $\psi'$ ] tranclp-into-rtranclp
  unfolding full1-def by metis

lemma full-simplify-preserves-unsat:
  assumes full simplify  $\psi$   $\psi'$ 
  shows satisfiable  $\psi' \longrightarrow$  satisfiable  $\psi$ 
  using assms rtanclp-simplify-preserves-unsat[of  $\psi$   $\psi'$ ] unfolding full-def by metis

lemma resolution-preserves-unsat:
  assumes resolution  $\psi$   $\psi'$ 
  shows satisfiable (fst  $\psi'$ )  $\longrightarrow$  satisfiable (fst  $\psi$ )
  using assms apply (induct rule: resolution.induct)
  using full1-simplify-preserves-unsat apply (metis fst-conv)
  using full-simplify-preserves-unsat simplify-preserves-unsat by fastforce

lemma rtranclp-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 rtranclp-simplify-preserve-partial-tree:
  assumes simplify**  $N$   $N'$ 
  and partial-interps  $t$   $I$   $N$ 
  shows partial-interps  $t$   $I$   $N'$ 
  using assms apply (induction, simp)
  using simplify-preserve-partial-tree by metis

lemma full1-simplify-preserve-partial-tree:
  assumes full1 simplify  $N$   $N'$ 
  and partial-interps  $t$   $I$   $N$ 
  shows partial-interps  $t$   $I$   $N'$ 
  using assms rtranclp-simplify-preserve-partial-tree[of  $N$   $N'$   $t$   $I$ ] tranclp-into-rtranclp
  unfolding full1-def by fast

lemma full-simplify-preserve-partial-tree:
  assumes full simplify  $N$   $N'$ 
  and partial-interps  $t$   $I$   $N$ 
  shows partial-interps  $t$   $I$   $N'$ 
  using assms rtranclp-simplify-preserve-partial-tree[of  $N$   $N'$   $t$   $I$ ] tranclp-into-rtranclp
  unfolding full-def by fast

lemma resolution-preserve-partial-tree:
  assumes resolution  $S$   $S'$ 
  and partial-interps  $t$   $I$  (fst  $S$ )
  shows partial-interps  $t$   $I$  (fst  $S'$ )
  using assms apply induction

```

using *full1-simplify-preserve-partial-tree fst-conv* **apply** *metis*
using *full-simplify-preserve-partial-tree inference-preserve-partial-tree* **by** *fastforce*

lemma *rtrancp-resolution-preserve-partial-tree*:
assumes *resolution** S S'*
and *partial-interps t I (fst S)*
shows *partial-interps t I (fst S')*
using *assms* **apply** *induction*
using *resolution-preserve-partial-tree* **by** *fast+*
thm *nat-less-induct nat.induct*

lemma *nat-ge-induct[case-names 0 Suc]*:
assumes *P 0*
and $\bigwedge n. (\bigwedge m. m < \text{Suc } n \implies P m) \implies P (\text{Suc } n)$
shows *P n*
using *assms* **apply** (*induct rule: nat-less-induct*)
by (*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 $\{f \varphi L \mid \varphi L. (\varphi = x \vee \varphi \in N) \wedge L \in \# \varphi \wedge P \varphi L\} \subseteq \{f x L \mid L. L \in \# x \wedge P x L\}$
 $\cup \{f \varphi L \mid \varphi L. \varphi \in N \wedge L \in \# \varphi \wedge P \varphi L\}$ **by** *auto*
moreover **have** *finite {f x L | L. L \in # x}* **by** *auto*
ultimately show *?case* **using** *IH finite-subset* **by** *fastforce*

qed

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

interpretation *sum-count-ge-2*:

folding ($\lambda \varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L\}\})) 0$

rewrites

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

proof –

show *folding ($\lambda \varphi. \text{op} + (\text{msetsum } (\text{image-mset } (\text{count } \varphi) \{\# L \in \# \varphi. 2 \leq \text{count } \varphi L\}\}))$*
by *standard auto*

then interpret *sum-count-ge-2*:

folding ($\lambda \varphi. \text{op} + (\text{msetsum } \{\# \text{count } \varphi L \mid L \in \# \varphi. 2 \leq \text{count } \varphi L\}\})) 0$.

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

qed

lemma *finite-incl-le-setsum*:
finite ($B :: 'a$ multiset set) $\implies A \subseteq B \implies \Xi A \leq \Xi B$
proof (*induction arbitrary:A rule: finite-induct*)
 case *empty*
 then show *?case* **by** *simp*
next
 case (*insert a F*) **note** *finite = this(1)* **and** *aF = this(2)* **and** *IH = this(3)* **and** *AF = this(4)*
 show *?case*
 proof (*cases a ∈ A*)
 assume $a \notin A$
 then have $A \subseteq F$ **using** *AF* **by** *auto*
 then show *?case* **using** *IH[of A]* **by** (*simp add: aF local.finite*)
next
 assume $aA: a \in A$
 then have $A - \{a\} \subseteq F$ **using** *AF* **by** *auto*
 then have $\Xi (A - \{a\}) \leq \Xi F$ **using** *IH* **by** *blast*
 then show *?case*
 proof –
 obtain $nn :: nat \Rightarrow nat \Rightarrow nat$ **where**
 $\forall x0\ x1. (\exists v2. x0 = x1 + v2) = (x0 = x1 + nn\ x0\ x1)$
 by *moura*
 then have $\Xi F = \Xi (A - \{a\}) + nn\ (\Xi F)\ (\Xi (A - \{a\}))$
 by (*meson* $\langle \Xi (A - \{a\}) \leq \Xi F \rangle$ *le-iff-add*)
 then show *?thesis*
 by (*metis* (*no-types*) *le-iff-add aA aF add.assoc finite.insertI finite-subset insert.premis local.finite sum-count-ge-2.insert sum-count-ge-2.remove*)
 qed
qed
qed

lemma *simplify-finite-measure-decrease*:
simplify $N\ N' \implies \text{finite } N \implies \text{card } N' + \Xi N' < \text{card } N + \Xi N$
proof (*induction rule: simplify.induct*)
 case (*tautology-deletion A P*) **note** *an = this(1)* **and** *fin = this(2)*
 let $?N' = N - \{A + \{\#Pos\ P\# \} + \{\#Neg\ P\# \}\}$
 have $\text{card } ?N' < \text{card } N$
 by (*meson card-Diff1-less tautology-deletion.hyps tautology-deletion.premis*)
 moreover have $?N' \subseteq N$ **by** *auto*
 then have $\text{sum-count-ge-2 } ?N' \leq \text{sum-count-ge-2 } N$ **using** *finite-incl-le-setsum[OF fin]* **by** *blast*
 ultimately show *?case* **by** *linarith*
next
 case (*condensation A L*) **note** $AN = \text{this}(1)$ **and** $fin = \text{this}(2)$
 let $?C' = A + \{\#L\#\}$
 let $?C = A + \{\#L\#\} + \{\#L\#\}$
 let $?N' = N - \{?C\} \cup \{?C'\}$
 have $\text{card } ?N' \leq \text{card } N$
 using AN **by** (*metis* (*no-types, lifting*) *Diff-subset Un-empty-right Un-insert-right card.remove card-insert-if card-mono fin finite-Diff order-refl*)
 moreover have $\Xi \{?C'\} < \Xi \{?C\}$
 proof –
 have *mset-decomp*:
 $\{\# L a \in \# A. (L = La \longrightarrow La \in \# A) \wedge (L \neq La \longrightarrow 2 \leq \text{count } A\ La)\#\}$
 $= \{\# L a \in \# A. L \neq La \wedge 2 \leq \text{count } A\ La\#\} +$
 $\{\# L a \in \# A. L = La \wedge \text{Suc } 0 \leq \text{count } A\ L\#\}$
 by (*auto simp: multiset-eq-iff ac-simps*)
 have *mset-decomp2*: $\{\# L a \in \# A. L \neq La \longrightarrow 2 \leq \text{count } A\ La\#\} =$

```

    {# 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  $\Xi \ ?N' < \Xi \ N$ 
proof cases
  assume a1:  $?C' \in N$ 
  then show ?thesis
    proof -
      have f2:  $\bigwedge m \ M. \text{insert } (m::'a \text{ literal multiset}) (M - \{m\}) = M \cup \{m\} \vee m \notin M$ 
        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  $\langle \Xi \ \{A + \{\#L\# \}\rangle < \Xi \ \{A + \{\#L\# \} + \{\#L\# \}\rangle$ 
          add.right-neutral add-diff-cancel-left' add-gr-0 diff-less fin finite.emptyI not-le
          sum-count-ge-2.empty sum-count-ge-2.insert-remove trans-le-add2)
      then show ?thesis
        using f3 f2 a1 by (metis (no-types) Un-empty-right Un-insert-right condensation.hyps
          insert-iff multi-self-add-other-not-self)
    qed
  next
    assume  $?C' \notin N$ 
    have mset-decomp:
       $\{\# La \in \# A. (L = La \longrightarrow \text{Suc } 0 \leq \text{count } A \ La) \wedge (L \neq La \longrightarrow 2 \leq \text{count } A \ La)\#\}$ 
      =  $\{\# La \in \# A. L \neq La \wedge 2 \leq \text{count } A \ La\# \} +$ 
       $\{\# La \in \# A. L = La \wedge \text{Suc } 0 \leq \text{count } A \ L\#\}$ 
      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  $\langle \Xi \ \{A + \{\#L\# \}\rangle < \Xi \ \{A + \{\#L\# \} + \{\#L\# \}\rangle$  condensation.hyps fin
        sum-count-ge-2.remove[of - A +  $\{\#L\# \} + \{\#L\# \}$ ]  $\langle ?C' \notin N \rangle$ 
      by (auto simp: mset-decomp mset-decomp2 filter-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  $(N - \{B\}) < \text{card } N$  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  $\{(N', N). \text{finite } N \wedge \text{simplify } N \ N'\}$ 
  apply (rule wfP-if-measure[of finite simplify  $\lambda N. \text{card } N + \Xi \ N$ ])

```

using *simplify-finite-measure-decrease* by *blast*

lemma *wf-terminates*:

assumes *wf r*

shows $\exists N'. (N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r)$

proof –

let $?P = \lambda N. (\exists N'. (N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r))$

have $(\forall x. (\forall y. (y, x) \in r \longrightarrow ?P y) \longrightarrow ?P x)$

proof *clarify*

fix *x*

assume *H*: $\forall y. (y, x) \in r \longrightarrow ?P y$

{ assume $\exists y. (y, x) \in r$

then obtain *y* where *y*: $(y, x) \in r$ by *blast*

then have $?P y$ using *H* by *blast*

then have $?P x$ using *y* by (*meson rtrancl.rtrancl-into-rtrancl*)

}

moreover {

assume $\neg(\exists y. (y, x) \in r)$

then have $?P x$ by *auto*

}

ultimately show $?P x$ by *blast*

qed

moreover have $(\forall x. (\forall y. (y, x) \in r \longrightarrow ?P y) \longrightarrow ?P x) \longrightarrow \text{All } ?P$

using *assms unfolding wf-def* by (*rule allE*)

ultimately have *All ?P* by *blast*

then show $?P N$ by *blast*

qed

lemma *rtranclp-simplify-terminates*:

assumes *fin*: *finite N*

shows $\exists N'. \text{simplify}^{**} N N' \wedge \text{simplified } N'$

proof –

have *H*: $\{(N', N). \text{finite } N \wedge \text{simplify } N N'\} = \{(N', N). \text{simplify } N N' \wedge \text{finite } N\}$ by *auto*

then have *wf*: $\text{wf } \{(N', N). \text{simplify } N N' \wedge \text{finite } N\}$

using *simplify-terminates* by (*simp add: H*)

obtain *N'* where *N'*: $(N', N) \in \{(b, a). \text{simplify } a b \wedge \text{finite } a\}^*$ and

more: $(\forall N''. (N'', N') \notin \{(b, a). \text{simplify } a b \wedge \text{finite } a\})$

using *Prop-Resolution.wf-terminates[OF wf, of N]* by *blast*

have *1*: $\text{simplify}^{**} N N'$

using *N'* by (*induction rule: rtrancl.induct*) *auto*

then have *finite N'* using *fin rtranclp-simplify-preserves-finite* by *blast*

then have *2*: $\forall N''. \neg \text{simplify } N' N''$ using *more* by *auto*

show *?thesis* using *1 2* by *blast*

qed

lemma *finite-simplified-full1-simp*:

assumes *finite N*

shows $\text{simplified } N \vee (\exists N'. \text{full1 simplify } N N')$

using *rtranclp-simplify-terminates[OF assms]* *unfolding full1-def*

by (*metis Nitpick.rtranclp-unfold*)

lemma *finite-simplified-full-simp*:

assumes *finite N*

shows $\exists N'. \text{full simplify } N N'$

using *rtrancp-simplify-terminates*[*OF assms*] **unfolding** *full-def* **by** *metis*

lemma *can-decrease-tree-size-resolution*:

fixes $\psi :: 'v \text{ state}$ **and** $\text{tree} :: 'v \text{ sem-tree}$

assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ

and *partial-interps* *tree* *I* (*fst* ψ)

and *simplified* (*fst* ψ)

shows $\exists (\text{tree}' :: 'v \text{ sem-tree}) \psi'. \text{resolution}^{**} \psi \psi' \wedge \text{partial-interps } \text{tree}' I (\text{fst } \psi')$

$\wedge (\text{sem-tree-size } \text{tree}' < \text{sem-tree-size } \text{tree} \vee \text{sem-tree-size } \text{tree} = 0)$

using *assms*

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

case (*bigger* *xs* *I*) **note** *IH* = *this*(1) **and** *finite* = *this*(2) **and** *a-u-i* = *this*(3) **and** *part* = *this*(4)

and *simp* = *this*(5)

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

moreover {

assume *sn0*: *sem-tree-size* *xs* > 0

obtain *ag ad v* **where** *xs*: *xs* = *Node v ag ad* **using** *sn0* **by** (*cases xs*, *auto*)

{

assume *sem-tree-size* *ag* = 0 \wedge *sem-tree-size* *ad* = 0

then have *ag*: *ag* = *Leaf* **and** *ad*: *ad* = *Leaf* **by** (*cases ag*, *auto*, *cases ad*, *auto*)

{

then obtain $\chi \chi'$ **where**

χ : $\neg I \cup \{\text{Pos } v\} \models \chi$ **and**

tot χ : *total-over-m* ($I \cup \{\text{Pos } v\}$) $\{\chi\}$ **and**

$\chi\psi$: $\chi \in \text{fst } \psi$ **and**

χ' : $\neg I \cup \{\text{Neg } v\} \models \chi'$ **and**

tot χ' : *total-over-m* ($I \cup \{\text{Neg } v\}$) $\{\chi'\}$ **and** $\chi'\psi$: $\chi' \in \text{fst } \psi$

using *part* **unfolding** *xs* **by** *auto*

have *Posv*: *Pos v* $\notin \# \chi$ **using** 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 v* $\notin \# \chi$

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*^{**} $\psi \psi$

unfolding *xs* **by** (*auto* *simp* *add*: $\chi\psi$)

}

moreover {

assume *Pos* χ : *Pos v* $\notin \# \chi'$

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*^{**} $\psi \psi$

using $\chi'\psi$ *I* χ **unfolding** *xs* **by** *auto*

}


```

moreover {
  assume  $neg: Neg\ v \in \# \chi$  and  $pos: Pos\ v \in \# \chi'$ 
  have  $count\ \chi\ (Neg\ v) = 1$ 
    using  $simplified-count[OF\ simp\ \chi\psi]$   $neg$ 
    by ( $simp\ add: dual-order.antisym$ )
  have  $count\ \chi'\ (Pos\ v) = 1$ 
    using  $simplified-count[OF\ simp\ \chi'\psi]$   $pos$ 
    by ( $simp\ add: dual-order.antisym$ )

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

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

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

moreover {
  assume  $a: (\{\#Pos\ v\# \} + C', \{\#Neg\ v\# \} + C) \in snd\ \psi$ 
  then have  $(\exists \chi \in fst\ \psi. (\forall I. total-over-m\ I\ \{C+C'\} \longrightarrow total-over-m\ I\ \{\chi\})$ 
     $\wedge (\forall I. total-over-m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)) \vee tautology\ (C' + C)$ 
  proof -
    obtain  $p$  where  $p: Pos\ p \in \# (\{\#Pos\ v\# \} + C') \wedge Neg\ p \in \# (\{\#Neg\ v\# \} + C)$ 
       $\wedge ((\exists \chi \in fst\ \psi. (\forall I. total-over-m\ I\ \{(\{\#Pos\ v\# \} + C') - \{\#Pos\ p\# \} + (\{\#Neg\ v\# \}$ 

```

```

+ C) - {#Neg p#})} → total-over-m I {χ} ∧ (∀ I. total-over-m I {χ} → I ⊨ χ → I ⊨ ({#Pos
v#} + C') - {#Pos p#} + (({#Neg v#} + C) - {#Neg p#}))) ∨ tautology (({#Pos v#} + C') -
{#Pos p#} + (({#Neg v#} + C) - {#Neg p#})))
  using a by (blast intro: allE[OF a-u-i[unfolded subsumes-def Ball-def],
    of ({#Pos v#} + C', {#Neg v#} + C)])
  { assume p ≠ v
    then have Pos p ∈# C' ∧ Neg p ∈# C using p by force
    then have ?thesis by auto
  }
  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 ∃χ ∈ fst ψ. (∀ I. total-over-m I {C+C'} → total-over-m I {χ})
  ∧ (∀ I. total-over-m I {χ} → I ⊨ χ → I ⊨ C' + C)
  then obtain ϑ where
    ϑ: ϑ ∈ fst ψ and
    tot-ϑ-CC': ∀ I. total-over-m I {C+C'} → total-over-m I {ϑ} and
    ϑ-inv: ∀ I. total-over-m I {ϑ} → I ⊨ ϑ → I ⊨ C' + C by blast
  have partial-interps Leaf I (fst ψ)
    using tot-ϑ-CC' ϑ ϑ-inv totC totC' ⊢ I ⊨ C + C' total-over-m-sum by fastforce
  moreover have sem-tree-size Leaf < sem-tree-size xs unfolding xs by auto
  ultimately have ?case by blast
}
moreover {
  assume tautCC': tautology (C' + C)
  have total-over-m I {C'+C} using totC totC' total-over-m-sum by auto
  then have ¬tautology (C' + C)
    using ⊢ I ⊨ C + C' unfolding add.commute[of C C'] total-over-m-def
    unfolding tautology-def by auto
  then have False using tautCC' unfolding tautology-def by auto
}
ultimately have ?case by auto
}
ultimately have ?case by auto
}
ultimately have ?case using part by (metis (no-types) sem-tree-size.simps(1))
}
moreover {
  assume size-ag: sem-tree-size ag > 0
  have sem-tree-size ag < sem-tree-size xs unfolding xs by auto
  moreover have partial-interps ag (I ∪ {Pos v}) (fst ψ)
  and partad: partial-interps ad (I ∪ {Neg v}) (fst ψ)
  using part partial-interps.simps(2) unfolding xs by metis+
  moreover
    have sem-tree-size ag < sem-tree-size xs ⇒ finite (fst ψ) ⇒ already-used-inv ψ
    ⇒ partial-interps ag (I ∪ {Pos v}) (fst ψ) ⇒ simplified (fst ψ)
    ⇒ ∃ tree' ψ'. resolution** ψ ψ' ∧ partial-interps tree' (I ∪ {Pos v}) (fst ψ')
    ∧ (sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0)
    using IH[of ag I ∪ {Pos v}] by auto
  ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
    inf: resolution** ψ ψ'
    and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')

```

```

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

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

  have partial-interps ag (I ∪ {Pos v}) (fst ψ')
    using rtranclp-resolution-preserve-partial-tree inf partag by fast
  then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
  then have ?case using inf size size-ad unfolding xs by fastforce
}
ultimately have ?case by auto
}
ultimately show ?case by auto
qed

lemma resolution-completeness-inv:
  fixes ψ :: 'v :: linorder state
  assumes
    unsat: ¬satisfiable (fst ψ) and
    finite: finite (fst ψ) and
    a-u-v: already-used-inv ψ
  shows ∃ ψ'. (resolution** ψ ψ' ∧ {#} ∈ fst ψ')
proof -
  obtain tree where partial-interps tree {} (fst ψ)
    using partial-interps-build-sem-tree-atms assms by metis
  then show ?thesis
    using unsat finite a-u-v
  proof (induct tree arbitrary: ψ rule: sem-tree-size)
    case (bigger tree ψ) note H = this
    {
      fix χ
      assume tree: tree = Leaf
      obtain χ where χ: ¬ {} ⊨ χ and totχ: total-over-m {} {χ} and χψ: χ ∈ fst ψ
      using H unfolding tree by auto
    }
  qed

```

```

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, 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 rtranclp-resolution-finite by blast
have unsat: unsatisfiable (fst  $\psi'$ )
  using rtranclp-resolution-preserves-unsat inf uns by metis
have a-u-i': already-used-inv  $\psi'$ 
  using a-u-v inf rtranclp-resolution-already-used-inv[of  $\psi_0$   $\psi'$ ] by auto
have ?case
  using inf rtranclp-trans[of resolution] H(1)[OF s part' unsat fin a-u-i']  $\psi_0$  by blast
}
ultimately show ?case by (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 *rtranclp-resolution-preserves-already-used-inv*:
 assumes *resolution*** $S\ S'$
 and *already-used-inv* S
 shows *already-used-inv* S'
 using *assms*
 apply (induct rule: *rtranclp-induct*)
 apply *simp*
 using *resolution-preserves-already-used-inv* by *fast*

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

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

lemma *resolution-preserves-sat*:
 assumes *resolution* $S\ S'$
 and *satisfiable* (fst S)
 shows *satisfiable* (fst S')
 using *assms* **apply** (*induction* rule: *resolution.induct*)
 using *rtranclp-preserves-sat* *tranclp-into-rtranclp* **unfolding** *full1-def* **apply** *fastforce*
by (*metis fst-conv full-def inference-preserves-un-sat rtranclp-preserves-sat*
satisfiable-carac' satisfiable-def)

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

```

using assms apply (induction rule: rtrancpl-induct)
apply simp
using resolution-preserves-sat by blast

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

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

lemma simplified-falsity:
  assumes simp: simplified  $\psi$ 
  and  $\{\#\} \in \psi$ 
  shows  $\psi = \{\{\#\}\}$ 
proof (rule ccontr)
  assume H:  $\neg ?thesis$ 
  then obtain  $\chi$  where  $\chi \in \psi$  and  $\chi \neq \{\#\}$  using assms(2) by blast
  then have  $\{\#\} \subsetneq \chi$  by (simp add: mset-less-empty-nonempty)
  then have simplify  $\psi$  ( $\psi - \{\chi\}$ )
    using simplify.subsumption[OF assms(2)  $\langle \{\#\} \subsetneq \chi \rangle \langle \chi \in \psi \rangle$ ] by blast
  then show False using simp by blast
qed

lemma simplify-falsity-in-preserved:
  assumes simplify  $\chi s \ \chi s'$ 
  and  $\{\#\} \in \chi s$ 
  shows  $\{\#\} \in \chi s'$ 
  using assms
  by induction auto

lemma rtrancpl-simplify-falsity-in-preserved:
  assumes simplify**  $\chi s \ \chi s'$ 
  and  $\{\#\} \in \chi s$ 
  shows  $\{\#\} \in \chi s'$ 
  using assms
  by induction (auto intro: simplify-falsity-in-preserved)

lemma resolution-falsity-get-falsity-alone:
  assumes finite (fst  $\psi$ )
  shows  $(\exists \psi'. (\text{resolution** } \psi \ \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow (\exists a-u-v. \text{resolution** } \psi (\{\{\#\}\}, a-u-v))$ 
  (is  $?A \longleftrightarrow ?B$ )
proof
  assume  $?B$ 
  then show  $?A$  by auto
next
  assume  $?A$ 
  then obtain  $\chi s \ a-u-v$  where  $\chi s: \text{resolution** } \psi (\chi s, a-u-v)$  and  $F: \{\#\} \in \chi s$  by auto

```

```

{ assume simplified  $\chi s$ 
  then have ?B using simplified-falsity[OF - F]  $\chi s$  by blast
}
moreover {
  assume  $\neg$  simplified  $\chi s$ 
  then obtain  $\chi s'$  where full1 simplify  $\chi s \chi s'$ 
    by (metis  $\chi s$  assms finite-simplified-full1-simp fst-conv rtranclp-resolution-finite)
  then have  $\{\#\} \in \chi s'$ 
    unfolding full1-def by (meson F rtranclp-simplify-falsity-in-preserved
      trancpl-into-rtranclp)
  then have ?B
    by (metis  $\chi s$  full1 simplify  $\chi s \chi s'$  fst-conv full1-simp resolution-always-simplified
      rtranclp.rtrancl-into-rtrancl simplified-falsity)
}
ultimately show ?B by blast
qed

lemma resolution-soundness-and-completeness':
  fixes  $\psi :: 'v :: \text{linorder state}$ 
  assumes
    finite: finite (fst  $\psi$ ) and
    snd: snd  $\psi = \{\}$ 
  shows  $(\exists a-u-v. (\text{resolution}^{**} \psi (\{\#\}, a-u-v))) \longleftrightarrow \text{unsatisfiable} (\text{fst } \psi)$ 
    using assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone
    by metis

end
theory Prop-Superposition
imports Partial-Clausal-Logic ../lib/Herbrand-Interpretation
begin

```

4.2 Superposition

```

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:
   $S \models_h C \longleftrightarrow \{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\} \models C$ 
  unfolding Herbrand-Interpretation.true-cls-def Partial-Clausal-Logic.true-cls-def
  by auto

lemma herbrand-consistent-interp:
  consistent-interp  $(\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\})$ 
  unfolding consistent-interp-def by auto

lemma herbrand-total-over-set:
  total-over-set  $(\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\})\ T$ 
  unfolding total-over-set-def by auto

lemma herbrand-total-over-m:
  total-over-m  $(\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\})\ T$ 
  unfolding total-over-m-def by (auto simp add: herbrand-total-over-set)

```

lemma *herbrand-interp-iff-partial-interp-clss*:

$S \models_{hs} C \longleftrightarrow \{Pos\ P | P. P \in S\} \cup \{Neg\ P | P. P \notin S\} \models_s C$

unfolding *true-clss-def Ball-def herbrand-interp-iff-partial-interp-clss*

Partial-Clausal-Logic.true-clss-def **by** *auto*

definition *clss-lt* :: '*a*::wellorder clauses \Rightarrow '*a* clause \Rightarrow '*a* clauses **where**

clss-lt *N C* = $\{D \in N. D \# \subset \# C\}$

notation (*latex output*)

clss-lt ($-\hat{<}^{bsup} - \hat{<}^{esup}$)

locale *selection* =

fixes *S* :: '*a* clause \Rightarrow '*a* clause

assumes

S-selects-subseteq: $\bigwedge C. S\ C \leq \# C$ **and**

S-selects-neg-lits: $\bigwedge C\ L. L \in \# S\ C \implies is_neg\ L$

locale *ground-resolution-with-selection* =

selection S **for** *S* :: ('*a* :: wellorder) clause \Rightarrow '*a* clause

begin

context

fixes *N* :: '*a* clause set

begin

We do not create an equivalent of δ , but we directly defined N_C by inlining the definition.

function

production :: '*a* clause \Rightarrow '*a* interp

where

production C =

$\{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max}(\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1$
 $\wedge \neg (\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D) \models_h C \wedge S\ C = \{\#\}\}$

by *auto*

termination by (*relation* $\{(D, C). D \# \subset \# C\}$) (*auto simp: wf-less-multiset*)

declare *production.simps*[*simp del*]

definition *interp* :: '*a* clause \Rightarrow '*a* interp **where**

interp C = $(\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D)$

lemma *production-unfold*:

production C = $\{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max}(\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge \neg$
 $\neg \text{interp } C \models_h C \wedge S\ C = \{\#\}\}$

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 $\implies C \in N \wedge C \neq \{\#\} \wedge \text{Pos } A = \text{Max}(\text{set-mset } C) \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge$
 $\neg \text{interp } C \models_h C \wedge S\ C = \{\#\}$

unfolding *production-unfold* **by** *auto*

lemma *produces C A \implies Pos A $\in \#$ C*
by (*simp add: Max-in-lits producesD*)

lemma *interp'-def-in-set:*
interp C = ($\bigcup D \in \{D \in N. D \# \subseteq \# C\}. \text{production } D$)
unfolding *interp-def* **apply** *auto*
unfolding *production-unfold* **apply** *auto*
done

lemma *production-iff-produces:*
produces D A \longleftrightarrow A \in production D
unfolding *production-unfold* **by** *auto*

definition *Interp :: 'a clause \Rightarrow 'a interp where*
Interp C = interp C \cup production C

lemma
assumes *produces C P*
shows *Interp C \models_h C*
unfolding *Interp-def* **assms** **using** *producesD[OF assms]*
by (*metis Max-in-lits Un-insert-right insertI1 pos-literal-in-imp-true-cl*)

definition *INTERP :: 'a interp where*
INTERP = ($\bigcup D \in N. \text{production } D$)

lemma *interp-subseteq-Interp[simp]: interp C \subseteq Interp C*
unfolding *Interp-def* **by** *simp*

lemma *Interp-as-UNION: Interp C = ($\bigcup D \in \{D. D \# \subseteq \# C\}. \text{production } D$)*
unfolding *Interp-def* *interp-def* *le-multiset-def* **by** *fast*

lemma *productive-not-empty: productive C \implies C $\neq \{\#\}$*
unfolding *production-unfold* **by** *auto*

lemma *productive-imp-produces-Max-literal: productive C \implies produces C (atm-of (Max (set-mset C)))*
unfolding *production-unfold* **by** (*auto simp del: atm-of-Max-lit*)

lemma *productive-imp-produces-Max-atom: productive C \implies produces C (Max (atms-of C))*
unfolding *atms-of-def* *Max-atm-of-set-mset-commute[OF productive-not-empty]*
by (*rule productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-literal: produces C A \implies A = atm-of (Max (set-mset C))*
by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-atom: produces C A \implies A = Max (atms-of C)*
by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-atom*)

lemma *produces-imp-Pos-in-lits: produces C A \implies Pos A $\in \#$ C*
by (*auto intro: Max-in-lits dest!: producesD*)

lemma *productive-in-N: productive C \implies C \in N*
unfolding *production-unfold* **by** *auto*

lemma *produces-imp-atms-leq: produces C A \implies B \in atms-of C \implies B \leq A*
by (*metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom*)

singleton-inject)

lemma *produces-imp-neg-notin-lits*: $\text{produces } C \ A \implies \text{Neg } A \notin \# \ C$
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*: $C \# \subseteq \# \ D \implies \text{production } C \subseteq \text{interp } D$
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*: $C \# \subseteq \# \ D \implies \text{Interp } C \subseteq \text{interp } D$
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*: $A \notin \text{Interp } C \implies A \in \text{interp } D \implies C \# \subseteq \# \ D$
using *less-eq-imp-interp-subseteq-Interp multiset-linorder.not-less* **by** *blast*

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin \text{interp } C \implies A \in \text{interp } D \implies C \# \subseteq \# \ D$
using *less-eq-imp-interp-subseteq-interp multiset-linorder.not-less* **by** *blast*

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin \text{Interp } C \implies A \in \text{Interp } D \implies C \# \subseteq \# \ D$
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 67 of Weidenbach's book.

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*
then have $D \# \subseteq \# \ \{\# \ \text{Neg } A \# \}$
by (*auto intro: Max-pos-neg-less-multiset*)

moreover have $\{\#Neg\ A\#\} \# \subseteq \# C$
by (rule less-eq-imp-le-multiset) (rule mset-le-single[OF a-in-c])
ultimately show ?thesis
using d **by** (blast dest: less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp)
qed

lemma neg-notin-Interp-not-produce: $Neg\ A \in \# C \implies A \notin Interp\ D \implies C \# \subseteq \# D \implies \neg\ produces\ D''\ A$
by (auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-Interp)

lemma in-production-imp-produces: $A \in production\ C \implies produces\ C\ A$
by (metis insert-absorb productive-imp-produces-Max-atom singleton-insert-inj-eq')

lemma not-produces-imp-notin-production: $\neg\ produces\ C\ A \implies A \notin production\ C$
by (metis in-production-imp-produces)

lemma not-produces-imp-notin-interp: $(\bigwedge D. \neg\ produces\ D\ A) \implies A \notin interp\ C$
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: $Interp\ D \models_h C$ **and**
subs: $interp\ D' \subseteq (\bigcup C \in CC. production\ C)$
shows $(\bigcup C \in CC. production\ C) \models_h C$
proof (cases $\exists A. Pos\ A \in \# C \wedge A \in Interp\ D$)
case True
then obtain A **where** a-in-c: $Pos\ A \in \# C$ **and** a-at-d: $A \in Interp\ D$
by blast
from a-at-d **have** $A \in interp\ D'$
using d-lt-d' less-imp-Interp-subseteq-interp **by** blast
then show ?thesis
using subs a-in-c **by** (blast dest: contra-subsetD)
next
case False
then obtain A **where** a-in-c: $Neg\ A \in \# C$ **and** $A \notin Interp\ D$
using c-at-d **unfolding** true-cls-def **by** blast
then have $\bigwedge D''. \neg\ produces\ D''\ A$
using c-le-d neg-notin-Interp-not-produce **by** simp
then show ?thesis
using a-in-c subs not-produces-imp-notin-production **by** auto
qed

lemma true-Interp-imp-interp: $C \# \subseteq \# D \implies D \# \subset \# D' \implies Interp\ D \models_h C \implies interp\ D' \models_h C$
using interp-def true-Interp-imp-general **by** simp

lemma true-Interp-imp-Interp: $C \# \subseteq \# D \implies D \# \subset \# D' \implies Interp\ D \models_h C \implies Interp\ D' \models_h C$
using Interp-as-UNION interp-subseteq-Interp true-Interp-imp-general **by** simp

lemma true-Interp-imp-INTERP: $C \# \subseteq \# D \implies Interp\ D \models_h C \implies INTERP \models_h C$
using INTERP-def interp-subseteq-INTERP
true-Interp-imp-general[OF - less-multiset-right-total]

by *simp*

lemma *true-interp-imp-general*:

assumes

c-le-d: $C \# \subseteq \# D$ **and**

d-lt-d': $D \# \subset \# D'$ **and**

c-at-d: $\text{interp } D \models_h C$ **and**

subs: $\text{interp } D' \subseteq (\bigcup C \in CC. \text{production } C)$

shows $(\bigcup C \in CC. \text{production } C) \models_h C$

proof (*cases* $\exists A. \text{Pos } A \in \# C \wedge A \in \text{interp } D$)

case *True*

then obtain *A* **where** *a-in-c*: $\text{Pos } A \in \# C$ **and** *a-at-d*: $A \in \text{interp } D$

by *blast*

from *a-at-d* **have** $A \in \text{interp } D'$

using *d-lt-d'* *less-eq-imp-interp-subseteq-interp*[*OF multiset-order.less-imp-le*] **by** *blast*

then show *?thesis*

using *subs a-in-c* **by** (*blast dest: contra-subsetD*)

next

case *False*

then obtain *A* **where** *a-in-c*: $\text{Neg } A \in \# C$ **and** $A \notin \text{interp } D$

using *c-at-d* *unfolding true-cls-def* **by** *blast*

then have $\bigwedge D''. \neg \text{produces } D'' A$

using *c-le-d* **by** (*auto dest: produces-imp-in-interp less-eq-imp-interp-subseteq-interp*)

then show *?thesis*

using *a-in-c subs not-produces-imp-notin-production* **by** *auto*

qed

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *true-interp-imp-interp*: $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{interp } D' \models_h C$
using *interp-def true-interp-imp-general* **by** *simp*

lemma *true-interp-imp-Interp*: $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{Interp } D' \models_h C$
using *Interp-as-UNION interp-subseteq-Interp*[*of D'*] *true-interp-imp-general* **by** *simp*

lemma *true-interp-imp-INTERP*: $C \# \subseteq \# D \implies \text{interp } D \models_h C \implies \text{INTERP} \models_h C$
using *INTERP-def interp-subseteq-INTERP*
true-interp-imp-general[*OF - less-multiset-right-total*]
by *simp*

lemma *productive-imp-false-interp*: $\text{productive } C \implies \neg \text{interp } C \models_h C$
unfolding *production-unfold* **by** *auto*

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *cls-gt-double-pos-no-production*:

assumes *D*: $\{\# \text{Pos } P, \text{Pos } P \#\} \# \subset \# C$

shows $\neg \text{produces } C P$

proof –

let $?D = \{\# \text{Pos } P, \text{Pos } P \#\}$

note $D' = D[\text{unfolded less-multiset}_{HO}]$

consider

(*P*) *count* *C* (*Pos* *P*) ≥ 2

| (*Q*) *Q* **where** $Q > \text{Pos } P$ **and** $Q \in \# C$

using *HOL.spec*[*OF HOL.conjunct2*[*OF D'*], *of Pos P*] **by** (*auto split: if-split-asm*)

```

then show ?thesis
proof cases
  case Q
  have  $Q \in \text{set-mset } C$ 
    using  $Q(2)$  by (auto split: if-split-asm)
  then have  $\text{Max } (\text{set-mset } C) > \text{Pos } P$ 
    using  $Q(1)$  Max-gr-iff by blast
  then show ?thesis
    unfolding production-unfold by auto
next
  case P
  then show ?thesis
    unfolding production-unfold by auto
qed
qed

```

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma

assumes $D: C + \{\# \text{Neg } P \# \} \# \subset \# D$
 shows $\text{production } D \neq \{P\}$

proof –

note $D' = D[\text{unfolded less-multiset}_{HO}]$

consider

(P) $\text{Neg } P \in \# D$

| (Q) Q **where** $Q > \text{Neg } P$ **and** $\text{count } D \ Q > \text{count } (C + \{\# \text{Neg } P \# \}) \ Q$

using $\text{HOL.spec}[OF \ \text{HOL.conjunct2}[OF \ D], \text{ of } \text{Neg } P] \text{ count-greater-zero-iff}$ by fastforce

then show ?thesis

proof cases

case Q

have $Q \in \text{set-mset } D$

using $Q(2)$ *gr-implies-not0* by fastforce

then have $\text{Max } (\text{set-mset } D) > \text{Neg } P$

using $Q(1)$ *Max-gr-iff* by blast

then have $\text{Max } (\text{set-mset } D) > \text{Pos } P$

using *less-trans*[of $\text{Pos } P \ \text{Neg } P \ \text{Max } (\text{set-mset } D)$] by auto

then show ?thesis

unfolding *production-unfold* by auto

next

case P

then have $\text{Max } (\text{set-mset } D) > \text{Pos } P$

by (meson *Max-ge finite-set-mset le-less-trans linorder-not-le pos-less-neg*)

then show ?thesis

unfolding *production-unfold* by auto

qed

qed

lemma *in-interp-is-produced:*

assumes $P \in \text{INTERP}$

shows $\exists D. D + \{\# \text{Pos } P \# \} \in N \wedge \text{produces } (D + \{\# \text{Pos } P \# \}) \ P$

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 $MMax\ M \equiv Max\ (set-mset\ M)$

4.2.1 We can now define the rules of the calculus

inductive *superposition-rules* :: 'a clause \Rightarrow 'a clause \Rightarrow 'a clause \Rightarrow bool **where**
factoring: *superposition-rules* ($C + \{\#Pos\ P\# \} + \{\#Pos\ P\# \}$) $B\ (C + \{\#Pos\ P\# \}) \mid$
superposition-l: *superposition-rules* ($C_1 + \{\#Pos\ P\# \}$) ($C_2 + \{\#Neg\ P\# \}$) ($C_1 + C_2$)

inductive *superposition* :: 'a clauses \Rightarrow 'a clauses \Rightarrow bool **where**
superposition: $A \in N \Longrightarrow B \in N \Longrightarrow superposition-rules\ A\ B\ C$
 $\Longrightarrow superposition\ N\ (N \cup \{C\})$

definition *abstract-red* :: 'a::wellorder clause \Rightarrow 'a clauses \Rightarrow bool **where**
abstract-red $C\ N = (clss-lt\ N\ C \models_p C)$

lemma *less-multiset*[*iff*]: $M < N \longleftrightarrow M \# \subset \# N$
unfolding *less-multiset-def* **by** *auto*

lemma *less-eq-multiset*[*iff*]: $M \leq N \longleftrightarrow M \# \subseteq \# N$
unfolding *less-eq-multiset-def* **by** *auto*

lemma *herbrand-true-clss-true-clss-clss-herbrand-true-clss*:

assumes

$AB: A \models_{hs} B$ **and**

$BC: B \models_p C$

shows $A \models_h C$

proof –

let $?I = \{Pos\ P \mid P. P \in A\} \cup \{Neg\ P \mid P. P \notin A\}$

have $B: ?I \models_s B$ **using** AB

by (*auto simp add: herbrand-interp-iff-partial-interp-clss*)

have $IH: \bigwedge I. total-over-set\ I\ (atms-of\ C) \Longrightarrow total-over-m\ I\ B \Longrightarrow consistent-interp\ I$
 $\Longrightarrow I \models_s B \Longrightarrow I \models C$ **using** BC

by (*auto simp add: true-clss-clss-def*)

show *?thesis*

unfolding *herbrand-interp-iff-partial-interp-clss*

by (*auto intro: IH[of ?I] simp add: herbrand-total-over-set herbrand-total-over-m herbrand-consistent-interp B*)

qed

lemma *abstract-red-subset-mset-abstract-red*:

assumes

abstr: *abstract-red* $C\ N$ **and**

c-lt-d: $C \subseteq \# D$

shows *abstract-red* $D\ N$

proof –

have $\{D \in N. D \# \subset \# C\} \subseteq \{D' \in N. D' \# \subset \# D\}$

using *c-lt-d less-eq-imp-le-multiset* **by** *fastforce*

then show *?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-clb-extended*:

assumes

$A \models_p B$ **and**

tot: *total-over-m* I A **and**

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

$I-A: I \models_s A$

shows $I \models B$

proof –

let $?I = I \cup \{Pos\ P \mid P. P \in \text{atms-of } B \wedge P \notin \text{atms-of-s } I\}$

have *consistent-interp* $?I$

using *cons* **unfolding** *consistent-interp-def* *atms-of-s-def* *atms-of-def*

apply (*auto* 1 5 *simp* *add: image-iff*)

by (*metis* *atm-of-uminus* *literal.sel*(1))

moreover **have** *total-over-m* $?I$ ($A \cup \{B\}$)

proof –

obtain $aa :: 'a \text{ set} \Rightarrow 'a \text{ literal set} \Rightarrow 'a$ **where**

$f2: \forall x0\ x1. (\exists v2. v2 \in x0 \wedge Pos\ v2 \notin x1 \wedge Neg\ v2 \notin x1)$

$\longleftrightarrow (aa\ x0\ x1 \in x0 \wedge Pos\ (aa\ x0\ x1) \notin x1 \wedge Neg\ (aa\ x0\ x1) \notin x1)$

by *moura*

have $\forall a. a \notin \text{atms-of-ms } A \vee Pos\ a \in I \vee Neg\ a \in I$

using *tot* **by** (*simp* *add: total-over-m-def* *total-over-set-def*)

then **have** $aa\ (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\})\ (I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})$

$\notin \text{atms-of-ms } A \cup \text{atms-of-ms } \{B\} \vee Pos\ (aa\ (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\}))$

$(I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}) \in I$

$\cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}$

$\vee Neg\ (aa\ (\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\}))$

$(I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}) \in I$

$\cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\}$

by *auto*

then **have** *total-over-set* $(I \cup \{Pos\ a \mid a. a \in \text{atms-of } B \wedge a \notin \text{atms-of-s } I\})$

$(\text{atms-of-ms } A \cup \text{atms-of-ms } \{B\})$

using $f2$ **by** (*meson* *total-over-set-def*)

then **show** *?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-clb-def* **by** *auto*

then **show** *?thesis*

oops

lemma

assumes

$CP: \neg \text{clss-lt } N\ (\{\#C\# \} + \{\#E\# \}) \models_p \{\#C\# \} + \{\#Neg\ P\# \}$ **and**

$\text{clss-lt } N\ (\{\#C\# \} + \{\#E\# \}) \models_p \{\#E\# \} + \{\#Pos\ P\# \} \vee \text{clss-lt } N\ (\{\#C\# \} + \{\#E\# \}) \models_p \{\#C\# \} + \{\#Neg\ P\# \}$

shows $\text{clss-lt } N\ (\{\#C\# \} + \{\#E\# \}) \models_p \{\#E\# \} + \{\#Pos\ P\# \}$

oops

locale *ground-ordered-resolution-with-redundancy* =

ground-resolution-with-selection +

fixes *redundant* :: $'a::\text{wellorder clause} \Rightarrow 'a \text{ clauses} \Rightarrow \text{bool}$

assumes

redundant-iff-abstract: $\text{redundant } A\ N \longleftrightarrow \text{abstract-red } A\ N$

begin

definition *saturated* :: 'a clauses \Rightarrow bool **where**

saturated $N \longleftrightarrow (\forall A B C. A \in N \longrightarrow B \in N \longrightarrow \neg \text{redundant } A N \longrightarrow \neg \text{redundant } B N$
 $\longrightarrow \text{superposition-rules } A B C \longrightarrow \text{redundant } C N \vee C \in N)$

lemma

assumes

saturated: *saturated* N **and**

finite: *finite* N **and**

empty: $\{\#\} \notin N$

shows *INTERP* $N \models_{hs} N$

proof (rule *ccontr*)

let $?N_{\mathcal{I}} = \text{INTERP } N$

assume $\neg ?thesis$

then have *not-empty*: $\{E \in N. \neg ?N_{\mathcal{I}} \models_h E\} \neq \{\}$

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

def $D \equiv \text{Min } \{E \in N. \neg ?N_{\mathcal{I}} \models_h E\}$

have [*simp*]: $D \in N$

unfolding *D-def*

by (*metis* (*mono-tags*, *lifting*) *Min-in not-empty finite mem-Collect-eq rev-finite-subset subsetI*)

have *not-d-interp*: $\neg ?N_{\mathcal{I}} \models_h D$

unfolding *D-def*

by (*metis* (*mono-tags*, *lifting*) *Min-in finite mem-Collect-eq not-empty rev-finite-subset subsetI*)

have *cls-not-D*: $\bigwedge E. E \in N \implies E \neq D \implies \neg ?N_{\mathcal{I}} \models_h E \implies D \leq E$

using *finite D-def* **by** (*auto simp del: less-eq-multiset*)

obtain $C L$ **where** $D: D = C + \{\#L\#$ **and** $LSD: L \in \# S D \vee (S D = \{\#\} \wedge \text{Max } (\text{set-mset } D)$
 $= L)$

proof (*cases* $S D = \{\#\}$)

case *False*

then obtain L **where** $L \in \# S D$

using *Max-in-lits* **by** *blast*

moreover

then have $L \in \# D$

using *S-selects-subseteq*[of D] **by** *auto*

then have $D = (D - \{\#L\#$

by *auto*

ultimately show *?thesis* **using** *that* **by** *blast*

next

let $?L = \text{MMax } D$

case *True*

moreover

have $?L \in \# D$

by (*metis* (*no-types*, *lifting*) *Max-in-lits* ($D \in N$) *empty*)

then have $D = (D - \{\#?L\#$

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*

then have $?N_{\mathcal{I}} \models_{hs} \text{clss-lt } N D$

unfolding *clss-lt-def true-clss-def Ball-def* **by** *blast*

then show *False*

using *red not-d-interp* **unfolding** *abstract-red-def redundant-iff-abstract*


```

    using herbrand-true-clss-true-clss-clss-herbrand-true-clss by fast
qed

consider
  (L) P where L = Pos P and S D = {#} and Max (set-mset D) = Pos P
| (Lneg) P where L = Neg P
  using LSD S-selects-neg-lits[of L D] by (cases L) auto
then show False
proof cases
  case L note P = this(1) and S = this(2) and max = this(3)
  have count D L > 1
  proof (rule ccontr)
    assume ~ ?thesis
    then have count: count D L = 1
    unfolding D by (auto simp: not-in-iff)
    have  $\neg ?N_{\mathcal{I}} \models_h D$ 
    using not-d-interp true-interp-imp-INTERP ground-resolution-with-selection-axioms
    by blast
    then have 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)
    then have INTERP N  $\models_h D$ 
    unfolding D
    by (metis pos-literal-in-imp-true-clss produces-imp-Pos-in-lits
    production-subseteq-INTERP singletonI subsetCE)
    then show False
    using not-d-interp by blast
  qed
  then have Pos P  $\in \# C$ 
  by (simp add: P D)
  then obtain C' where C':D = C' + {#Pos P#} + {#Pos P#}
  unfolding D by (metis (full-types) P insert-DiffM2)
  have sup: superposition-rules D D (D - {#L#})
  unfolding C' L by (auto simp add: superposition-rules.simps)
  have C' + {#Pos P#}  $\# \subset \# C' + \{ \#Pos P \# \} + \{ \#Pos P \# \}$ 
  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' clss-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: if-split-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*
then have *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*
then have $\bigwedge y. y \in \# (E + \{\#Pos P\# \})$
 $\implies count\ (E + \{\#Pos P\# \})\ (Neg\ P) < count\ (C + \{\#Neg P\# \})\ (Neg\ P)$
using *count-greater-zero-iff* **by** *fastforce*
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 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 count-greater-zero-iff less-iff-Suc-add zero-less-Suc*)
have *ce-lt-d: $C + E \# \subset \# D$*
unfolding $D\ L$ **by** (*simp add: $(\bigwedge y. y \in \# E + \{\#Pos P\# \}) \implies y < Neg\ P$ ex-gt-imp-less-multiset*)
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 simp: not-in-iff*)
then have $\bigwedge y. y \in \# C+E$
 $\implies count\ (C+E)\ (Pos\ P) < count\ (E + \{\#Pos P\# \})\ (Pos\ P)$
using *set-mset-def* **by** *fastforce*

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**
 $I \models_s clss-lt\ N\ (C + E)$
then have $I \models C + E$

```

      using abs sorry
    moreover have  $\neg I \models C + \{\#Neg P\}$ 
      using CP unfolding true-clss-clss-def
      sorry
    ultimately have  $I \models E + \{\#Pos P\}$  by auto
  }
  then show  $clss\text{-}lt\ N\ (C + E) \models_p E + \{\#Pos P\}$ 
    unfolding true-clss-clss-def by auto
  qed
  moreover have  $clss\text{-}lt\ N\ (C + E) \subseteq clss\text{-}lt\ N\ (C + \{\#Neg P\})$ 
    using ce-lt-d mult-less-trans unfolding clss-lt-def D L by force
  ultimately have  $redundant\ (C + \{\#Neg P\})\ N \vee clss\text{-}lt\ N\ (C + E) \models_p E + \{\#Pos P\}$ 
    unfolding redundant-iff-abstract abstract-red-def using true-clss-clss-subset by blast
  show False sorry
  qed
  moreover have  $\neg redundant\ (E + \{\#Pos P\})\ N$ 
    sorry
  ultimately have  $CEN: C + E \in N$ 
    using  $\langle D \in N \rangle \langle E + \{\#Pos P\} \in N \rangle$  saturated sup-EC red unfolding saturated-def D L
    by (metis union-commute)
  have  $CED: C + E \neq D$ 
    using D ce-lt-d by auto
  have  $interp: \neg INTERP\ N \models_h C + E$ 
    sorry
  show False
    using cls-not-D[OF CEN CED interp] ce-lt-d unfolding INTERP-def less-eq-multiset-def by
  auto
  qed
  qed
end

```

lemma *tautology-is-redundant*:

```

  assumes tautology C
  shows abstract-red C N
  using assms unfolding abstract-red-def true-clss-clss-def tautology-def by auto

```

lemma *subsumed-is-redundant*:

```

  assumes AB:  $A \subset \# B$ 
  and AN:  $A \in N$ 
  shows abstract-red B N

```

proof –

```

  have  $A \in clss\text{-}lt\ N\ B$  using AN AB unfolding clss-lt-def
    by (auto dest: less-eq-imp-le-multiset simp add: multiset-order.dual-order.order-iff-strict)
  then show ?thesis
    using AB unfolding abstract-red-def true-clss-clss-def Partial-Clausal-Logic.true-clss-def
    by blast
  qed

```

inductive *redundant* :: 'a clause \Rightarrow 'a clauses \Rightarrow bool **where**

subsumption: $A \in N \Longrightarrow A \subset \# B \Longrightarrow redundant\ B\ N$

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)
  then show ?case
    using subsumed-is-redundant[of A N B] unfolding abstract-red-def class-lt-def by auto
qed

```

```

lemma redundant-mono:
  redundant A N  $\implies$  A  $\subseteq$  B  $\implies$  redundant B N
  apply (induction rule: redundant.induct)
  by (meson subset-mset.less-le-trans subsumption)

```

```

locale trac =
  selection S for S :: nat clause  $\Rightarrow$  nat clause
begin

```

```

end

```

```

end

```

4.3 Partial Clausal Logic

We here define decided literals (that will be used in both DPLL and CDCL) and the entailment corresponding to it.

```

theory Partial-Annotated-Clausal-Logic
imports Partial-Clausal-Logic

```

```

begin

```

4.3.1 Decided Literals

Definition

```

datatype ('v, 'mark) ann-lit =
  is-decided: Decided (lit-of: 'v literal) |
  is-proped: Propagated (lit-of: 'v literal) (mark-of: 'mark)

```

```

lemma ann-lit-list-induct[case-names Nil Decided Propagated]:
  assumes P [] and
   $\bigwedge L\ xs. P\ xs \implies P\ (\text{Decided } L \# xs)$  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-decided-ex-Decided:
  is-decided L  $\implies$  ( $\bigwedge K. L = \text{Decided } K \implies P$ )  $\implies$  P
  by (cases L) auto

```

```

type-synonym ('v, 'm) ann-lits = ('v, 'm) ann-lit list

```

```

definition lits-of :: ('a, 'b) ann-lit set  $\Rightarrow$  'a literal set where
  lits-of Ls = lit-of ' Ls

```

```

abbreviation lits-of-l :: ('a, 'b) ann-lits  $\Rightarrow$  'a literal set where

```

$lits\text{-}of\text{-}l\ Ls \equiv lits\text{-}of\ (set\ Ls)$

lemma *lits-of-l-empty[simp]*:
 $lits\text{-}of\ \{\} = \{\}$
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-insert[simp]*:
 $lits\text{-}of\ (insert\ L\ Ls) = insert\ (lit\text{-}of\ L)\ (lits\text{-}of\ Ls)$
unfolding *lits-of-def* **by** *auto*

lemma *lits-of-l-Un[simp]*:
 $lits\text{-}of\ (l \cup l') = lits\text{-}of\ l \cup lits\text{-}of\ l'$
unfolding *lits-of-def* **by** *auto*

lemma *finite-lits-of-def[simp]*:
 $finite\ (lits\text{-}of\text{-}l\ L)$
unfolding *lits-of-def* **by** *auto*

abbreviation *unmark* **where**
 $unmark \equiv (\lambda a. \{\#lit\text{-}of\ a\# \})$

abbreviation *unmark-s* **where**
 $unmark\text{-}s\ M \equiv unmark\ 'M$

abbreviation *unmark-l* **where**
 $unmark\text{-}l\ M \equiv unmark\text{-}s\ (set\ M)$

lemma *atms-of-ms-lambda-lit-of-is-atm-of-lit-of[simp]*:
 $atms\text{-}of\text{-}ms\ (unmark\text{-}l\ M') = atm\text{-}of\ 'lits\text{-}of\text{-}l\ M'$
unfolding *atms-of-ms-def* *lits-of-def* **by** *auto*

lemma *lits-of-l-empty-is-empty[iff]*:
 $lits\text{-}of\text{-}l\ M = \{\} \longleftrightarrow M = []$
by (*induct* *M*) (*auto simp: lits-of-def*)

Entailment

definition *true-annot* :: ('a, 'm) *ann-lits* \Rightarrow 'a *clause* \Rightarrow bool (**infix** \models_a 49) **where**
 $I \models_a C \longleftrightarrow (lits\text{-}of\text{-}l\ I) \models C$

definition *true-annots* :: ('a, 'm) *ann-lits* \Rightarrow 'a *clauses* \Rightarrow bool (**infix** \models_{as} 49) **where**
 $I \models_{as} CC \longleftrightarrow (\forall C \in CC. I \models_a C)$

lemma *true-annot-empty-model[simp]*:
 $\neg [] \models_a \psi$
unfolding *true-annot-def* *true-cls-def* **by** *simp*

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

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

lemma *true-annots-empty[simp]*:

$I \models_{as} \{\}$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-single-true-annot*[*iff*]:
 $I \models_{as} \{C\} \longleftrightarrow I \models_a C$
unfolding *true-annots-def* **by** *auto*

lemma *true-annot-insert-l*[*simp*]:
 $M \models_a A \implies L \# M \models_a A$
unfolding *true-annot-def* **by** *auto*

lemma *true-annots-insert-l* [*simp*]:
 $M \models_{as} A \implies L \# M \models_{as} A$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-union*[*iff*]:
 $M \models_{as} A \cup B \longleftrightarrow (M \models_{as} A \wedge M \models_{as} B)$
unfolding *true-annots-def* **by** *auto*

lemma *true-annots-insert*[*iff*]:
 $M \models_{as} \text{insert } a \ A \longleftrightarrow (M \models_a a \wedge M \models_{as} A)$
unfolding *true-annots-def* **by** *auto*

Link between \models_{as} and \models_s :

lemma *true-annots-true-cls*:
 $I \models_{as} CC \longleftrightarrow \text{lits-of-l } I \models_s CC$
unfolding *true-annots-def* *Ball-def* *true-annot-def* *true-clss-def* **by** *auto*

lemma *in-lit-of-true-annot*:
 $a \in \text{lits-of-l } M \longleftrightarrow M \models_a \{\#a\# \}$
unfolding *true-annot-def* *lits-of-def* **by** *auto*

lemma *true-annot-lit-of-notin-skip*:
 $L \# M \models_a A \implies \text{lit-of } L \not\in \# A \implies M \models_a A$
unfolding *true-annot-def* *true-cls-def* **by** *auto*

lemma *true-clss-singleton-lit-of-implies-incl*:
 $I \models_s \text{unmark-l } MLs \implies \text{lits-of-l } MLs \subseteq I$
unfolding *true-clss-def* *lits-of-def* **by** *auto*

lemma *true-annot-true-clss-cls*:
 $MLs \models_a \psi \implies \text{set } (\text{map } \text{unmark } MLs) \models_p \psi$
unfolding *true-annot-def* *true-clss-cls-def* *true-cls-def*
by (*auto dest: true-clss-singleton-lit-of-implies-incl*)

lemma *true-annots-true-clss-cls*:
 $MLs \models_{as} \psi \implies \text{set } (\text{map } \text{unmark } MLs) \models_{ps} \psi$
by (*auto*
dest: true-clss-singleton-lit-of-implies-incl
simp add: true-clss-def true-annots-def true-annot-def lits-of-def true-cls-def
true-clss-clss-def)

lemma *true-annots-decided-true-cls*[*iff*]:
 $\text{map } \text{Decided } M \models_{as} N \longleftrightarrow \text{set } M \models_s N$
proof –

have *: *lit-of* ‘ *Decided* ‘ *set M = set M* **unfolding** *lits-of-def* **by** *force*
show ?thesis **by** (*simp add: true-annots-true-cls * lits-of-def*)
qed

lemma *true-annot-singleton[iff]*: $M \models_a \{\#L\# \} \longleftrightarrow L \in \text{lits-of-}l\ M$
unfolding *true-annot-def lits-of-def* **by** *auto*

lemma *true-annots-true-clss-clss*:
 $A \models_{as} \Psi \implies \text{unmark-}l\ A \models_{ps} \Psi$
unfolding *true-clss-clss-def true-annots-def true-clss-def*
by (*auto dest!: true-clss-singleton-lit-of-implies-incl*
simp: lits-of-def true-annot-def true-cls-def)

lemma *true-annot-commute*:
 $M @ M' \models_a D \longleftrightarrow M' @ M \models_a D$
unfolding *true-annot-def* **by** (*simp add: Un-commute*)

lemma *true-annots-commute*:
 $M @ M' \models_{as} D \longleftrightarrow M' @ M \models_{as} D$
unfolding *true-annots-def* **by** (*auto simp: 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-cls-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*

Defined and undefined literals

We introduce the functions *defined-lit* and *undefined-lit* to know whether a literal is defined with respect to a list of decided literals (aka a trail in most cases).

Remark that *undefined* already exists and is a completely different Isabelle function.

definition *defined-lit* :: (*'a*, *'m*) *ann-lits* \Rightarrow *'a literal* \Rightarrow *bool*
where
 $\text{defined-lit } I\ L \longleftrightarrow (\text{Decided } L \in \text{set } I) \vee (\exists P. \text{Propagated } L\ P \in \text{set } I)$
 $\vee (\text{Decided } (-L) \in \text{set } I) \vee (\exists P. \text{Propagated } (-L)\ P \in \text{set } I)$

abbreviation *undefined-lit* :: (*'a*, *'m*) *ann-lits* \Rightarrow *'a literal* \Rightarrow *bool*
where $\text{undefined-lit } I\ L \equiv \neg \text{defined-lit } I\ L$

lemma *defined-lit-rev[simp]*:
 $\text{defined-lit } (\text{rev } M)\ L \longleftrightarrow \text{defined-lit } M\ L$
unfolding *defined-lit-def* **by** *auto*

lemma *atm-imp-decided-or-proped*:
assumes $x \in \text{set } I$
shows
 $(\text{Decided } (- \text{lit-of } x) \in \text{set } I)$
 $\vee (\text{Decided } (\text{lit-of } x) \in \text{set } I)$
 $\vee (\exists l. \text{Propagated } (- \text{lit-of } x)\ l \in \text{set } I)$
 $\vee (\exists l. \text{Propagated } (\text{lit-of } x)\ l \in \text{set } I)$
using *assms ann-lit.exhaust-sel* **by** *metis*

lemma *literal-is-lit-of-decided*:

assumes $L = \text{lit-of } x$
shows $(x = \text{Decided } L) \vee (\exists l'. x = \text{Propagated } L \ l')$
using *assms* **by** (*cases* x) *auto*

lemma *true-annot-iff-decided-or-true-lit*:

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

lemma *consistent-inter-true-annots-satisfiable*:

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

lemma *defined-lit-map*:

defined-lit $Ls \ L \longleftrightarrow \text{atm-of } L \in (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } Ls$
unfolding *defined-lit-def* **apply** (*rule iffI*)
using *image-iff* **apply** *fastforce*
by (*fastforce simp add: atm-of-eq-atm-of dest: atm-imp-decided-or-proped*)

lemma *defined-lit-uminus[iff]*:

defined-lit $I \ (-L) \longleftrightarrow \text{defined-lit } I \ L$
unfolding *defined-lit-def* **by** *auto*

lemma *Decided-Propagated-in-iff-in-lits-of-l*:

defined-lit $I \ L \longleftrightarrow (L \in \text{lits-of-l } I \vee \neg L \in \text{lits-of-l } I)$
unfolding *lits-of-def* **by** (*metis lits-of-def true-annot-iff-decided-or-true-lit true-lit-def*)

lemma *consistent-add-undefined-lit-consistent[simp]*:

assumes
consistent-interp (*lits-of-l* Ls) **and**
undefined-lit $Ls \ L$
shows *consistent-interp* (*insert* L (*lits-of-l* Ls))
using *assms* **unfolding** *consistent-interp-def* **by** (*auto simp: Decided-Propagated-in-iff-in-lits-of-l*)

lemma *decided-empty[simp]*:

$\neg \text{defined-lit } [] \ L$
unfolding *defined-lit-def* **by** *simp*

4.3.2 Backtracking

fun *backtrack-split* :: $('v, 'm) \text{ ann-lits}$

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

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

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

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

lemma *backtrack-split-fst-not-decided*: $a \in \text{set } (\text{fst } (\text{backtrack-split } l)) \implies \neg \text{is-decided } a$

by (*induct l rule: ann-lit-list-induct*) *auto*

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

snd (*backtrack-split* l) $\neq [] \implies \text{is-decided } (\text{hd } (\text{snd } (\text{backtrack-split } l)))$

by (*induct l rule: ann-lit-list-induct*) *auto*

lemma *backtrack-split-list-eq[simp]*:

fst (backtrack-split l) @ (snd (backtrack-split l)) = l
by (induct l rule: ann-lit-list-induct) auto

lemma *backtrack-snd-empty-not-decided*:

backtrack-split M = (M'', []) $\implies \forall l \in \text{set } M. \neg \text{is-decided } l$
by (metis append-Nil2 backtrack-split-fst-not-decided backtrack-split-list-eq snd-conv)

lemma *backtrack-split-some-is-decided-then-snd-has-hd*:

$\exists l \in \text{set } M. \text{is-decided } l \implies \exists M' L' M''. \text{backtrack-split } M = (M'', L' \# M')$
by (metis backtrack-snd-empty-not-decided list.exhaust prod.collapse)

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma *backtrack-split-takeWhile-dropWhile*:

backtrack-split M = (takeWhile (Not o is-decided) M, dropWhile (Not o is-decided) M)
by (induction M rule: ann-lit-list-induct) auto

4.3.3 Decomposition with respect to the First Decided Literals

In this section we define a function that returns a decomposition with the first decided literal. This function is useful to define the backtracking of DPLL.

Definition

The pattern *get-all-ann-decomposition* [] = [([], [])] is necessary otherwise, we can call the *hd* function in the other pattern.

fun *get-all-ann-decomposition* :: ('a, 'm) ann-lits
 $\Rightarrow ((\text{'a}, \text{'m}) \text{ ann-lits} \times (\text{'a}, \text{'m}) \text{ ann-lits}) \text{ list}$ **where**
get-all-ann-decomposition (Decided L # Ls) =
 (Decided L # Ls, []) # *get-all-ann-decomposition* Ls |
get-all-ann-decomposition (Propagated L P # Ls) =
 (apsnd ((op #) (Propagated L P)) (hd (*get-all-ann-decomposition* Ls)))
 # tl (*get-all-ann-decomposition* Ls) |
get-all-ann-decomposition [] = [([], [])]

value *get-all-ann-decomposition* [Propagated A5 B5, Decided C4, Propagated A3 B3,
 Propagated A2 B2, Decided C1, Propagated A0 B0]

Now we can prove several simple properties about the function.

lemma *get-all-ann-decomposition-never-empty[iff]*:

get-all-ann-decomposition M = [] \longleftrightarrow False
by (induct M, simp) (rename-tac a xs, case-tac a, auto)

lemma *get-all-ann-decomposition-never-empty-sym[iff]*:

[] = *get-all-ann-decomposition M \longleftrightarrow False*
using *get-all-ann-decomposition-never-empty[of M]* **by** presburger

lemma *get-all-ann-decomposition-decomp*:

hd (get-all-ann-decomposition S) = (a, c) $\implies S = c @ a$

proof (induct S arbitrary: a c)

case Nil

then show ?case **by** simp

next

case (Cons x A)

then show ?case **by** (cases x; cases hd (get-all-ann-decomposition A)) auto
qed

lemma get-all-ann-decomposition-backtrack-split:

backtrack-split $S = (M, M') \longleftrightarrow \text{hd } (\text{get-all-ann-decomposition } S) = (M', M)$

proof (induction S arbitrary: M M')

case Nil

then show ?case **by** auto

next

case (Cons a S)

then show ?case **using** backtrack-split-takeWhile-dropWhile **by** (cases a) force+

qed

lemma get-all-ann-decomposition-Nil-backtrack-split-snd-Nil:

get-all-ann-decomposition $S = [([], A)] \implies \text{snd } (\text{backtrack-split } S) = []$

by (simp add: get-all-ann-decomposition-backtrack-split sndI)

This functions says that the first element is either empty or starts with a decided element of the list.

lemma get-all-ann-decomposition-length-1-fst-empty-or-length-1:

assumes get-all-ann-decomposition $M = (a, b) \# []$

shows $a = [] \vee (\text{length } a = 1 \wedge \text{is-decided } (\text{hd } a) \wedge \text{hd } a \in \text{set } M)$

using assms

proof (induct M arbitrary: a b rule: ann-lit-list-induct)

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

next

case (Decided L mark)

then show ?case **by** simp

next

case (Propagated L mark M)

then show ?case **by** (cases get-all-ann-decomposition M) force+

qed

lemma get-all-ann-decomposition-fst-empty-or-hd-in-M:

assumes get-all-ann-decomposition $M = (a, b) \# l$

shows $a = [] \vee (\text{is-decided } (\text{hd } a) \wedge \text{hd } a \in \text{set } M)$

using assms **apply** (induct M arbitrary: a b rule: ann-lit-list-induct)

apply auto[2]

by (metis UnCI backtrack-split-snd-hd-decided get-all-ann-decomposition-backtrack-split
get-all-ann-decomposition-decomp hd-in-set list.sel(1) set-append snd-conv)

lemma get-all-ann-decomposition-snd-not-decided:

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

and $L \in \text{set } b$

shows $\neg \text{is-decided } L$

using assms **apply** (induct M arbitrary: a b rule: ann-lit-list-induct, simp)

by (rename-tac L' xs a b, case-tac get-all-ann-decomposition xs; fastforce)+

lemma tl-get-all-ann-decomposition-skip-some:

assumes $x \in \text{set } (\text{tl } (\text{get-all-ann-decomposition } M1))$

shows $x \in \text{set } (\text{tl } (\text{get-all-ann-decomposition } (M0 @ M1)))$

using assms

by (induct M0 rule: ann-lit-list-induct)

(auto simp add: list.set-sel(2))

lemma hd-get-all-ann-decomposition-skip-some:

```

assumes (x, y) = hd (get-all-ann-decomposition M1)
shows (x, y) ∈ set (get-all-ann-decomposition (M0 @ Decided K # M1))
using assms
proof (induction M0 rule: ann-lit-list-induct)
  case Nil
  then show ?case by auto
next
  case (Decided L M0)
  then show ?case by auto
next
  case (Propagated L C M0) note xy = this(1)[OF this(2-)] and hd = this(2)
  then show ?case
    by (cases get-all-ann-decomposition (M0 @ Decided K # M1))
      (auto dest!: get-all-ann-decomposition-decomp
        arg-cong[of get-all-ann-decomposition - - hd])
qed

```

lemma *in-get-all-ann-decomposition-in-get-all-ann-decomposition-prepend:*
 $(a, b) \in \text{set } (\text{get-all-ann-decomposition } M') \implies$
 $\exists b'. (a, b' @ b) \in \text{set } (\text{get-all-ann-decomposition } (M @ M'))$
apply (induction M rule: ann-lit-list-induct)
apply (metis append-Nil)
apply auto[]
by (rename-tac L' m xs, case-tac get-all-ann-decomposition (xs @ M')) auto

lemma *in-get-all-ann-decomposition-decided-or-empty:*
assumes $(a, b) \in \text{set } (\text{get-all-ann-decomposition } M)$
shows $a = [] \vee (\text{is-decided } (\text{hd } a))$
using *assms*
proof (induct M arbitrary: a b rule: ann-lit-list-induct)
case Nil **then show** ?case **by** simp
next
case (Decided l M)
then show ?case **by** auto
next
case (Propagated l mark M)
then show ?case **by** (cases get-all-ann-decomposition M) force+
qed

lemma *get-all-ann-decomposition-remove-undecided-length:*
assumes $\forall l \in \text{set } M'. \neg \text{is-decided } l$
shows $\text{length } (\text{get-all-ann-decomposition } (M' @ M'')) = \text{length } (\text{get-all-ann-decomposition } M'')$
using *assms* **by** (induct M' arbitrary: M'' rule: ann-lit-list-induct) auto

lemma *get-all-ann-decomposition-not-is-decided-length:*
assumes $\forall l \in \text{set } M'. \neg \text{is-decided } l$
shows $1 + \text{length } (\text{get-all-ann-decomposition } (\text{Propagated } (-L) P \# M))$
 $= \text{length } (\text{get-all-ann-decomposition } (M' @ \text{Decided } L \# M))$
using *assms* *get-all-ann-decomposition-remove-undecided-length* **by** fastforce

lemma *get-all-ann-decomposition-last-choice:*
assumes $\text{tl } (\text{get-all-ann-decomposition } (M' @ \text{Decided } L \# M)) \neq []$
and $\forall l \in \text{set } M'. \neg \text{is-decided } l$
and $\text{hd } (\text{tl } (\text{get-all-ann-decomposition } (M' @ \text{Decided } L \# M))) = (M0', M0)$
shows $\text{hd } (\text{get-all-ann-decomposition } (\text{Propagated } (-L) P \# M)) = (M0', \text{Propagated } (-L) P \# M0)$
using *assms* **by** (induct M' rule: ann-lit-list-induct) auto

```

lemma get-all-ann-decomposition-except-last-choice-equal:
  assumes  $\forall l \in \text{set } M'. \neg \text{is-decided } l$ 
  shows  $tl \ (get-all-ann-decomposition \ (Propagated \ (-L) \ P \ \# \ M))$ 
   $= tl \ (tl \ (get-all-ann-decomposition \ (M' \ @ \ Decided \ L \ \# \ M)))$ 
  using assms by (induct  $M'$  rule: ann-lit-list-induct) auto

lemma get-all-ann-decomposition-hd-hd:
  assumes  $get-all-ann-decomposition \ Ls = (M, C) \ \# \ (M0, M0') \ \# \ l$ 
  shows  $tl \ M = M0' \ @ \ M0 \wedge \text{is-decided} \ (hd \ M)$ 
  using assms
proof (induct  $Ls$  arbitrary:  $M \ C \ M0 \ M0' \ l$ )
  case Nil
  then show ?case by simp
next
  case (Cons  $a \ Ls \ M \ C \ M0 \ M0' \ l$ ) note  $IH = \text{this}(1)$  and  $g = \text{this}(2)$ 
  { fix  $L \ \text{level}$ 
    assume  $a: a = Decided \ L$ 
    have  $Ls = M0' \ @ \ M0$ 
    using  $g \ a$  by (force intro: get-all-ann-decomposition-decomp)
    then have  $tl \ M = M0' \ @ \ M0 \wedge \text{is-decided} \ (hd \ M)$  using  $g \ a$  by auto
  }
  moreover {
    fix  $L \ P$ 
    assume  $a: a = Propagated \ L \ P$ 
    have  $tl \ M = M0' \ @ \ M0 \wedge \text{is-decided} \ (hd \ M)$ 
    using  $IH \ Cons.premis$  unfolding  $a$  by (cases get-all-ann-decomposition  $Ls$ ) auto
  }
  ultimately show ?case by (cases  $a$ ) auto
qed

lemma get-all-ann-decomposition-exists-prepend[dest]:
  assumes  $(a, b) \in \text{set} \ (get-all-ann-decomposition \ M)$ 
  shows  $\exists c. M = c \ @ \ b \ @ \ a$ 
  using assms apply (induct  $M$  rule: ann-lit-list-induct)
  apply simp
  by (rename-tac  $L' \ xs$ , case-tac get-all-ann-decomposition  $xs$ ;
    auto dest!: arg-cong[of get-all-ann-decomposition - - hd]
    get-all-ann-decomposition-decomp) +

lemma get-all-ann-decomposition-incl:
  assumes  $(a, b) \in \text{set} \ (get-all-ann-decomposition \ M)$ 
  shows  $\text{set } b \subseteq \text{set } M$  and  $\text{set } a \subseteq \text{set } M$ 
  using assms get-all-ann-decomposition-exists-prepend by fastforce +

lemma get-all-ann-decomposition-exists-prepend':
  assumes  $(a, b) \in \text{set} \ (get-all-ann-decomposition \ M)$ 
  obtains  $c$  where  $M = c \ @ \ b \ @ \ a$ 
  using assms apply (induct  $M$  rule: ann-lit-list-induct)
  apply auto[1]
  by (rename-tac  $L' \ xs$ , case-tac hd (get-all-ann-decomposition  $xs$ ),
    auto dest!: get-all-ann-decomposition-decomp simp add: list.set-sel(2)) +

lemma union-in-get-all-ann-decomposition-is-subset:
  assumes  $(a, b) \in \text{set} \ (get-all-ann-decomposition \ M)$ 
  shows  $\text{set } a \cup \text{set } b \subseteq \text{set } M$ 

```

using *assms* by *force*

lemma *Decided-cons-in-get-all-ann-decomposition-append-Decided-cons*:
 $\exists M1\ M2. (Decided\ K\ \# \ M1, M2) \in set\ (get-all-ann-decomposition\ (c\ @\ Decided\ K\ \# \ c'))$
apply (*induction* *c* *rule*: *ann-lit-list-induct*)
apply *auto*[2]
apply (*rename-tac* *L* *xs*,
case-tac *hd* (*get-all-ann-decomposition* (*xs* @ *Decided* *K* # *c'*)))
apply (*case-tac* *get-all-ann-decomposition* (*xs* @ *Decided* *K* # *c'*))
by *auto*

lemma *fst-get-all-ann-decomposition-prepend-not-decided*:
assumes $\forall m \in set\ MS. \neg\ is-decided\ m$
shows $set\ (map\ fst\ (get-all-ann-decomposition\ M))$
 $=\ set\ (map\ fst\ (get-all-ann-decomposition\ (MS\ @\ M)))$
using *assms* **apply** (*induction* *MS* *rule*: *ann-lit-list-induct*)
apply *auto*[2]
by (*rename-tac* *L* *m* *xs*; *case-tac* *get-all-ann-decomposition* (*xs* @ *M*)) *simp-all*

Entailment of the Propagated by the Decided Literal

lemma *get-all-ann-decomposition-snd-union*:
 $set\ M = \bigcup (set\ 'snd\ 'set\ (get-all-ann-decomposition\ M)) \cup \{L\ |\ L.\ is-decided\ L \wedge L \in set\ M\}$
(is $?M\ M = ?U\ M \cup ?Ls\ M$ **)**
proof (*induct* *M* *rule*: *ann-lit-list-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Decided* *L* *M*) **note** *IH* = *this*(1)
then have *Decided* *L* $\in ?Ls\ (Decided\ L\ \# \ M)$ **by** *auto*
moreover have $?U\ (Decided\ L\ \# \ M) = ?U\ M$ **by** *auto*
moreover have $?M\ M = ?U\ M \cup ?Ls\ M$ **using** *IH* **by** *auto*
ultimately show *?case* **by** *auto*
next
case (*Propagated* *L* *m* *M*)
then show *?case* **by** (*cases* (*get-all-ann-decomposition* *M*)) *auto*
qed

definition *all-decomposition-implies* :: '*a* literal multiset set
 $\Rightarrow (('a, 'm)\ ann-lits \times ('a, 'm)\ ann-lits)\ list \Rightarrow bool$ **where**
 $all-decomposition-implies\ N\ S \longleftrightarrow (\forall (Ls, seen) \in set\ S. unmark-l\ Ls \cup N \models_{ps} unmark-l\ seen)$

lemma *all-decomposition-implies-empty[iff]*:
 $all-decomposition-implies\ N\ []$ **unfolding** *all-decomposition-implies-def* **by** *auto*

lemma *all-decomposition-implies-single[iff]*:
 $all-decomposition-implies\ N\ [(Ls, seen)] \longleftrightarrow unmark-l\ Ls \cup N \models_{ps} unmark-l\ seen$
unfolding *all-decomposition-implies-def* **by** *auto*

lemma *all-decomposition-implies-append[iff]*:
 $all-decomposition-implies\ N\ (S\ @\ S')$
 $\longleftrightarrow (all-decomposition-implies\ N\ S \wedge all-decomposition-implies\ N\ S')$
unfolding *all-decomposition-implies-def* **by** *auto*

lemma *all-decomposition-implies-cons-pair[iff]*:
 $all-decomposition-implies\ N\ ((Ls, seen) \# \ S')$

\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
 $(\text{unmark-}l \text{ (fst } l) \cup N \models_{ps} \text{unmark-}l \text{ (snd } l) \wedge$
 $\text{all-decomposition-implies } N S')$
unfolding *all-decomposition-implies-def* **by** *auto*

lemma *all-decomposition-implies-trail-is-implied*:
assumes *all-decomposition-implies* N (*get-all-ann-decomposition* M)
shows $N \cup \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M\}$
 $\models_{ps} \text{unmark } ' \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-ann-decomposition } M))$
using *assms*
proof (*induct length (get-all-ann-decomposition M) arbitrary: M*)
case 0
then show ?*case* **by** *auto*
next
case (*Suc n*) **note** $IH = \text{this}(1)$ **and** $\text{length} = \text{this}(2)$ **and** $\text{decomp} = \text{this}(3)$
consider
 $(le1) \text{length } (\text{get-all-ann-decomposition } M) \leq 1$
 $\mid (gt1) \text{length } (\text{get-all-ann-decomposition } M) > 1$
by *arith*
then show ?*case*
proof *cases*
case *le1*
then obtain a b **where** $g: \text{get-all-ann-decomposition } M = (a, b) \# []$
by (*cases get-all-ann-decomposition M*) *auto*
moreover {
assume $a = []$
then have ?*thesis* **using** *Suc.premis g* **by** *auto*
}
moreover {
assume $l: \text{length } a = 1$ **and** $m: \text{is-decided } (\text{hd } a)$ **and** $hd: \text{hd } a \in \text{set } M$
then have $\text{unmark } (\text{hd } a) \in \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M\}$ **by** *auto*
then have $H: \text{unmark-}l \ a \cup N \subseteq N \cup \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M\}$
using l **by** (*cases a*) *auto*
have $f1: \text{unmark-}l \ a \cup N \models_{ps} \text{unmark-}l \ b$
using *decomp* **unfolding** *all-decomposition-implies-def g* **by** *simp*
have ?*thesis*
apply (*rule true-clss-clss-subset*) **using** $f1 \ H \ g$ **by** *auto*
}
ultimately show ?*thesis*
using *get-all-ann-decomposition-length-1-fst-empty-or-length-1* **by** *blast*
next
case *gt1*
then obtain $Ls0$ $seen0$ M' **where**
 $Ls0: \text{get-all-ann-decomposition } M = (Ls0, \text{seen0}) \# \text{get-all-ann-decomposition } M'$ **and**
 $\text{length}' : \text{length } (\text{get-all-ann-decomposition } M') = n$ **and**
 $M'\text{-in-}M: \text{set } M' \subseteq \text{set } M$
using length **by** (*induct M rule: ann-lit-list-induct*) (*auto simp: subset-insertI2*)
let ? $d = \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-ann-decomposition } M'))$
let ? $unM = \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M\}$
let ? $unM' = \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M'\}$
{
assume $n = 0$

```

then have get-all-ann-decomposition  $M' = []$  using length' by auto
then have ?thesis using Suc.prems unfolding all-decomposition-implies-def Ls0 by auto
}
moreover {
  assume  $n: n > 0$ 
  then obtain Ls1 seen1 l where
    Ls1: get-all-ann-decomposition  $M' = (Ls1, seen1) \# l$ 
    using length' by (induct  $M'$  rule: ann-lit-list-induct) auto

  have all-decomposition-implies N (get-all-ann-decomposition  $M'$ )
    using decomp unfolding Ls0 by auto
  then have  $N: N \cup ?unM' \models_{ps} unmark-s ?d$ 
    using IH length' by auto
  have  $l: N \cup ?unM' \subseteq N \cup ?unM$ 
    using M'-in-M by auto
  from true-clss-clss-subset[OF this N]
  have  $\Psi N: N \cup ?unM \models_{ps} unmark-s ?d$  by auto
  have is-decided (hd Ls0) and  $LS: tl Ls0 = seen1 @ Ls1$ 
    using get-all-ann-decomposition-hd-hd[of M] unfolding Ls0 Ls1 by auto

  have  $LSM: seen1 @ Ls1 = M'$  using get-all-ann-decomposition-decomp[of M] Ls1 by auto
  have  $M': set M' = ?d \cup \{L \mid L. is-decided L \wedge L \in set M'\}$ 
    using get-all-ann-decomposition-snd-union by auto

  {
    assume  $Ls0 \neq []$ 
    then have  $hd Ls0 \in set M$ 
      using get-all-ann-decomposition-fst-empty-or-hd-in-M Ls0 by blast
    then have  $N \cup ?unM \models_p unmark (hd Ls0)$ 
      using  $\langle is-decided (hd Ls0) \rangle$  by (metis (mono-tags, lifting) UnCI mem-Collect-eq true-clss-clss-in)
  } note  $hd-Ls0 = this$ 

  have  $l: unmark \text{ ` } (?d \cup \{L \mid L. is-decided L \wedge L \in set M'\}) = unmark-s ?d \cup ?unM'$ 
    by auto
  have  $N \cup ?unM' \models_{ps} unmark \text{ ` } (?d \cup \{L \mid L. is-decided L \wedge L \in set M'\})$ 
    unfolding  $l$  using  $N$  by (auto simp: all-in-true-clss-clss)
  then have  $t: N \cup ?unM' \models_{ps} unmark-l (tl Ls0)$ 
    using  $M'$  unfolding  $LS LSM$  by auto
  then have  $N \cup ?unM \models_{ps} unmark-l (tl Ls0)$ 
    using M'-in-M true-clss-clss-subset[OF - t, of  $N \cup ?unM$ ] by auto
  then have  $N \cup ?unM \models_{ps} unmark-l Ls0$ 
    using hd-Ls0 by (cases Ls0) auto

  moreover have  $unmark-l Ls0 \cup N \models_{ps} unmark-l seen0$ 
    using decomp unfolding Ls0 by simp
  moreover have  $\bigwedge M Ma. (M::'a \text{ literal multiset set}) \cup Ma \models_{ps} M$ 
    by (simp add: all-in-true-clss-clss)
  ultimately have  $\Psi: N \cup ?unM \models_{ps} unmark-l seen0$ 
    by (meson true-clss-clss-left-right true-clss-clss-union-and true-clss-clss-union-l-r)

  moreover have  $unmark \text{ ` } (set seen0 \cup ?d) = unmark-l seen0 \cup unmark-s ?d$ 
    by auto
  ultimately have ?thesis using  $\Psi N$  unfolding Ls0 by simp
}
ultimately show ?thesis by auto

```

qed
qed

lemma *all-decomposition-implies-propagated-lits-are-implied*:
assumes *all-decomposition-implies* N (*get-all-ann-decomposition* M)
shows $N \cup \{\text{unmark } L \mid L. \text{ is-decided } L \wedge L \in \text{set } M\} \models_{ps} \text{unmark-l } M$
(is $?I \models_{ps} ?A$)

proof –

have $?I \models_{ps} \text{unmark-s } \{L \mid L. \text{ is-decided } L \wedge L \in \text{set } M\}$
by (*auto intro: all-in-true-clss-clss*)
moreover have $?I \models_{ps} \text{unmark } ' \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-ann-decomposition } M))$
using *all-decomposition-implies-trail-is-implied* **assms** **by** *blast*
ultimately have $N \cup \{\text{unmark } m \mid m. \text{ is-decided } m \wedge m \in \text{set } M\}$
 $\models_{ps} \text{unmark } ' \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-ann-decomposition } M))$
 $\cup \text{unmark } ' \{m \mid m. \text{ is-decided } m \wedge m \in \text{set } M\}$
by *blast*
then show $?thesis$
by (*metis (no-types) get-all-ann-decomposition-snd-union[of M] image-Un*)

qed

lemma *all-decomposition-implies-insert-single*:
all-decomposition-implies N $M \implies \text{all-decomposition-implies } (\text{insert } C \ N) \ M$
unfolding *all-decomposition-implies-def* **by** *auto*

4.3.4 Negation of Clauses

We define the negation of a '*a Partial-Clausal-Logic.clause*': it converts it from the a single clause to a set of clauses, wherein each clause is a single negated literal.

definition $CNot :: 'v \text{ clause} \Rightarrow 'v \text{ clauses}$ **where**
 $CNot \ \psi = \{ \{ \# - L \# \} \mid L. L \in \# \ \psi \}$

lemma *in-CNot-uminus[iff]*:
shows $\{ \# L \# \} \in CNot \ \psi \longleftrightarrow -L \in \# \ \psi$
unfolding *CNot-def* **by** *force*

lemma
shows
CNot-singleton[simp]: $CNot \ \{ \# L \# \} = \{ \{ \# - L \# \} \}$ **and**
CNot-empty[simp]: $CNot \ \{ \# \} = \{ \}$ **and**
CNot-plus[simp]: $CNot \ (A + B) = CNot \ A \cup CNot \ B$
unfolding *CNot-def* **by** *auto*

lemma *CNot-eq-empty[iff]*:
 $CNot \ D = \{ \} \longleftrightarrow D = \{ \# \}$
unfolding *CNot-def* **by** (*auto simp add: multiset-eqI*)

lemma *in-CNot-implies-uminus*:
assumes $L \in \# \ D$ **and** $M \models_{as} CNot \ D$
shows $M \models_a \{ \# - L \# \}$ **and** $-L \in \text{lits-of-l } M$
using *assms* **by** (*auto simp: true-annots-def true-annot-def CNot-def*)

lemma *CNot-remdups-mset[simp]*:
 $CNot \ (\text{remdups-mset } A) = CNot \ A$
unfolding *CNot-def* **by** *auto*

lemma *Ball-CNot-Ball-mset[simp]*:
 $(\forall x \in CNot\ D. P\ x) \longleftrightarrow (\forall L \in \# D. P\ \{\# - L\# \})$
unfolding *CNot-def* **by** *auto*

lemma *consistent-CNot-not*:
assumes *consistent-interp I*
shows $I \models_s CNot\ \varphi \implies \neg I \models \varphi$
using *assms unfolding consistent-interp-def true-clss-def true-cls-def* **by** *auto*

lemma *total-not-true-cls-true-clss-CNot*:
assumes *total-over-m I {φ}* **and** $\neg I \models \varphi$
shows $I \models_s CNot\ \varphi$
using *assms unfolding total-over-m-def total-over-set-def true-clss-def true-cls-def CNot-def*
apply *clarify*
by *(rename-tac x L, case-tac L) (force intro: pos-lit-in-atms-of neg-lit-in-atms-of)+*

lemma *total-not-CNot*:
assumes *total-over-m I {φ}* **and** $\neg I \models_s CNot\ \varphi$
shows $I \models \varphi$
using *assms total-not-true-cls-true-clss-CNot* **by** *auto*

lemma *atms-of-ms-CNot-atms-of[simp]*:
 $atms-of-ms\ (CNot\ C) = atms-of\ C$
unfolding *atms-of-ms-def atms-of-def CNot-def* **by** *fastforce*

lemma *true-clss-clss-contradiction-true-clss-cls-false*:
 $C \in D \implies D \models_{ps} CNot\ C \implies D \models_p \{\#\}$
unfolding *true-clss-clss-def true-clss-cls-def total-over-m-def*
by *(metis Un-commute atms-of-empty atms-of-ms-CNot-atms-of atms-of-ms-insert atms-of-ms-union consistent-CNot-not insert-absorb sup-bot.left-neutral true-clss-def)*

lemma *true-annots-CNot-all-atms-defined*:
assumes $M \models_{as} CNot\ T$ **and** $a1: L \in \# T$
shows $atm-of\ L \in atm-of\ \text{'lits-of-l}\ M$
by *(metis assms atm-of-uminus image-eqI in-CNot-implies-uminus(1) true-annot-singleton)*

lemma *true-annots-CNot-all-uminus-atms-defined*:
assumes $M \models_{as} CNot\ T$ **and** $a1: -L \in \# T$
shows $atm-of\ L \in atm-of\ \text{'lits-of-l}\ M$
by *(metis assms atm-of-uminus image-eqI in-CNot-implies-uminus(1) true-annot-singleton)*

lemma *true-clss-clss-false-left-right*:
assumes $\{\{\#L\#\}\} \cup B \models_p \{\#\}$
shows $B \models_{ps} CNot\ \{\#L\#\}$
unfolding *true-clss-clss-def true-clss-cls-def*
proof *(intro allI impI)*
fix I
assume
 $tot: total-over-m\ I\ (B \cup CNot\ \{\#L\#\})$ **and**
 $cons: consistent-interp\ I$
 $I: I \models_s B$
have $total-over-m\ I\ (\{\{\#L\#\}\} \cup B)$ **using** *tot* **by** *auto*
then have $\neg I \models_s insert\ \{\#L\#\}\ B$
using *assms cons unfolding true-clss-cls-def* **by** *simp*
then show $I \models_s CNot\ \{\#L\#\}$
using *tot I* **by** *(cases L) auto*

qed

lemma *true-annots-true-cls-def-iff-negation-in-model*:

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

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

lemma *true-annot-CNot-diff*:

$I \models_{as} CNot\ C \implies I \models_{as} CNot\ (C - C')$

by (*auto simp: true-annots-true-cls-def-iff-negation-in-model dest: in-diffD*)

lemma *CNot-mset-replicate[simp]*:

$CNot\ (mset\ (replicate\ n\ L)) = (if\ n = 0\ then\ \{\}\ else\ \{\{ \# - L \# \} \})$

by (*induction n*) *auto*

lemma *consistent-CNot-not-tautology*:

$consistent_interp\ M \implies M \models_s CNot\ D \implies \neg tautology\ D$

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

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

by *simp*

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

$total_over_m\ I\ (CNot\ C) = total_over_set\ I\ (atms_of\ C)$

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

The following lemma is very useful when in the goal appears an axioms like $\neg L = K$: this lemma allows the simplifier to rewrite L.

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

by *auto*

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

assumes

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

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

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

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

proof (*intro allI impI*)

fix *I*

assume

$tot: total_over_set\ I\ (atms_of_ms\ (A \cup \{ \# L \# \}))$ **and**

$cons: consistent_interp\ I$ **and**

$I: I \models_s A$

let $?I = I \cup \{ Pos\ P \mid P. P \in atms_of\ CC \wedge P \notin atm_of\ 'I \}$

have $cons'$: $consistent_interp\ ?I$

using *cons unfolding consistent-interp-def*

by (*auto simp: uminus-lit-swap atms-of-def rev-image-eqI*)

have I' : $?I \models_s A$

using *I true-clss-union-increase* **by** *blast*

have $tot-CNot$: $total_over_m\ ?I\ (A \cup CNot\ CC)$

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

then have $tot-I-A-CC-L$: $total_over_m\ ?I\ (A \cup \{ CC + \{ \# L \# \} \})$

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

then have $?I \models CC + \{ \# L \# \}$ **using** *CC-L cons' I' unfolding true-clss-cls-def* **by** *blast*

moreover
 have $?I \models_s CNot\ CC$ **using** $CNot-CC\ cons'\ I'\ tot-CNot$ **unfolding** $true-clss-clss-def$ **by** $auto$
 then have $\neg A \models_p CC$
 by $(metis\ (no-types,\ lifting)\ I'\ atms-of-ms-CNot-atms-of-ms\ atms-of-ms-union\ cons'\ consistent-CNot-not\ tot-CNot\ total-over-m-def\ true-clss-clss-def)$
 then have $\neg ?I \models CC$ **using** $\langle ?I \models_s CNot\ CC \rangle\ cons'\ consistent-CNot-not$ **by** $blast$
 ultimately have $?I \models \{\#L\# \}$ **by** $blast$
 then show $I \models \{\#L\# \}$
 by $(metis\ (no-types,\ lifting)\ atms-of-ms-union\ cons'\ consistent-CNot-not\ tot\ total-not-CNot\ total-over-m-def\ total-over-set-union\ true-clss-union-increase)$
qed

lemma $true-annots-CNot-lit-of-notin-skip$:
 assumes $LM: L \# M \models_{as} CNot\ A$ **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** $l: l \in CNot\ A$
 then have $L \# M \models_a l$ **by** $auto$
 then show $M \models_a l$ **using** $LA\ l$ **by** $(cases\ L)\ (auto\ simp: CNot-def)$
qed

lemma $true-clss-clss-union-false-true-clss-clss-cnot$:
 $A \cup \{B\} \models_{ps} \{\{\#\}\} \longleftrightarrow A \models_{ps} CNot\ B$
using $total-not-CNot\ consistent-CNot-not$ **unfolding** $total-over-m-def\ true-clss-clss-def$
by $fastforce$

lemma $true-annot-remove-hd-if-notin-vars$:
 assumes $a \# M' \models_a D$ **and** $atm-of\ (lit-of\ a) \notin atms-of\ D$
 shows $M' \models_a D$
using $assms\ true-clss-remove-hd-if-notin-vars$ **unfolding** $true-annot-def$ **by** $auto$

lemma $true-annot-remove-if-notin-vars$:
 assumes $M @ M' \models_a D$ **and** $\forall x \in atms-of\ D. x \notin atm-of\ 'lits-of-l\ M$
 shows $M' \models_a D$
using $assms$ **by** $(induct\ M)\ (auto\ dest: true-annot-remove-hd-if-notin-vars)$

lemma $true-annots-remove-if-notin-vars$:
 assumes $M @ M' \models_{as} D$ **and** $\forall x \in atms-of-ms\ D. x \notin atm-of\ 'lits-of-l\ M$
 shows $M' \models_{as} D$ **unfolding** $true-annots-def$
using $assms$ **unfolding** $true-annots-def\ atms-of-ms-def$
by $(force\ dest: true-annot-remove-if-notin-vars)$

lemma $all-variables-defined-not-imply-cnot$:
 assumes
 $\forall s \in atms-of-ms\ \{B\}. s \in atm-of\ 'lits-of-l\ A$ **and**
 $\neg A \models_a B$
 shows $A \models_{as} CNot\ B$
unfolding $true-annot-def\ true-annots-def\ Ball-def\ CNot-def\ true-lit-def$
proof $(clarify,\ rule\ ccontr)$
 fix L
 assume $LB: L \in \# B$ **and** $\neg lits-of-l\ A \models_l \neg L$
 then have $atm-of\ L \in atm-of\ 'lits-of-l\ A$
 using $assms(l)$ **by** $(simp\ add: atm-of-lit-in-atms-of\ lits-of-def)$
 then have $L \in lits-of-l\ A \vee \neg L \in lits-of-l\ A$

using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set* by *metis*
 then have $L \in \text{ lits-of-l } A$ using $\langle \neg \text{ lits-of-l } A \models l - L \rangle$ by *auto*
 then show *False*
 using *LB assms(2)* unfolding *true-annot-def true-lit-def true-cls-def Bex-def*
 by *blast*
 qed

lemma *CNot-union-mset[simp]*:
 $CNot (A \# \cup B) = CNot A \cup CNot B$
 unfolding *CNot-def* by *auto*

4.3.5 Other

abbreviation *no-dup* $L \equiv \text{ distinct } (\text{ map } (\lambda l. \text{ atm-of } (\text{ lit-of } l)) L)$

lemma *no-dup-rev[simp]*:
 $\text{ no-dup } (\text{ rev } M) \longleftrightarrow \text{ no-dup } M$
 by (*auto simp: rev-map[symmetric]*)

lemma *no-dup-length-eq-card-atm-of-lits-of-l*:
 assumes *no-dup* M
 shows $\text{ length } M = \text{ card } (\text{ atm-of ' lits-of-l } M)$
 using *assms* unfolding *lits-of-def* by (*induct M*) (*auto simp add: image-image*)

lemma *distinct-consistent-interp*:
 $\text{ no-dup } M \implies \text{ consistent-interp } (\text{ lits-of-l } M)$

proof (*induct M*)

case *Nil*
 show *?case* by *auto*

next

case (*Cons L M*)
 then have *a1*: *consistent-interp* (*lits-of-l M*) by *auto*
 have *a2*: $\text{ atm-of } (\text{ lit-of } L) \notin (\lambda l. \text{ atm-of } (\text{ lit-of } l)) \text{ ' set } M$ using *Cons.prem*s by *auto*
 have *undefined-lit M* (*lit-of L*)
 using *a2* unfolding *defined-lit-map* by *fastforce*
 then show *?case*
 using *a1* by *simp*

qed

lemma *distinct-get-all-ann-decomposition-no-dup*:
 assumes $(a, b) \in \text{ set } (\text{ get-all-ann-decomposition } M)$
 and *no-dup* M
 shows *no-dup* $(a @ b)$
 using *assms* by *force*

lemma *true-annot-lit-of-notin-skip*:

assumes $L \# M \models_{as} CNot A$
 and $\neg \text{ lit-of } L \notin \# A$
 and *no-dup* $(L \# M)$
 shows $M \models_{as} CNot A$

proof –

have $\forall l \in \# A. \neg l \in \text{ lits-of-l } (L \# M)$
 using *assms(1)* *in-CNot-implies-uminus(2)* by *blast*

moreover

have $\text{ atm-of } (\text{ lit-of } L) \notin \text{ atm-of ' lits-of-l } M$
 using *assms(3)* unfolding *lits-of-def* by *force*

then have $\text{lit-of } L \notin \text{lits-of-l } M$ **unfolding** *lits-of-def*
by (*metis (no-types) atm-of-uminus imageI*)
ultimately have $\forall l \in \# A. \neg l \in \text{lits-of-l } M$
using *assms(2)* **by** (*metis insert-iff list.simps(15) lits-of-insert uminus-of-uminus-id*)
then show *?thesis* **by** (*auto simp add: true-annots-def*)
qed

4.3.6 Extending Entailments to multisets

We have defined previous entailment with respect to sets, but we also need a multiset version depending on the context. The conversion is simple using the function *set-mset* (in this direction, there is no loss of information).

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*:: $'v \text{ clause multiset} \Rightarrow 'v \text{ clause multiset} \Rightarrow \text{bool}$ (**infix** \models_{psm} 50) **where**
 $I \models_{psm} C \equiv \text{set-mset } I \models_{ps} (\text{set-mset } C)$

Analog of theorem *true-clss-clss-subsetE*

lemma *true-clss-clss-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*:: $'a \text{ clause multiset} \Rightarrow 'a \text{ clause} \Rightarrow \text{bool}$ (**infix** \models_{pm} 50) **where**
 $I \models_{pm} C \equiv \text{set-mset } I \models_p C$

abbreviation *distinct-mset-mset* :: $'a \text{ multiset multiset} \Rightarrow \text{bool}$ **where**
 $\text{distinct-mset-mset } \Sigma \equiv \text{distinct-mset-set } (\text{set-mset } \Sigma)$

abbreviation *all-decomposition-implies-m* **where**
 $\text{all-decomposition-implies-m } A B \equiv \text{all-decomposition-implies } (\text{set-mset } A) B$

abbreviation *atms-of-mm* :: $'a \text{ literal multiset multiset} \Rightarrow 'a \text{ set}$ **where**
 $\text{atms-of-mm } U \equiv \text{atms-of-ms } (\text{set-mset } U)$

Other definition using *Union-mset*

lemma *atms-of-mm* $U \equiv \text{set-mset } (\bigcup \# \text{ image-mset } (\text{image-mset } \text{atm-of}) U)$
unfolding *atms-of-ms-def* **by** (*auto simp: atms-of-def*)

abbreviation *true-clss-m*:: $'a \text{ interp} \Rightarrow 'a \text{ clause multiset} \Rightarrow \text{bool}$ (**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**
 $I \models_{sextm} C \equiv I \models_{sext} \text{set-mset } C$

type-synonym $'v \text{ clauses} = 'v \text{ clause multiset}$
end

Chapter 5

NOT's CDCL and DPLL

```
theory CDCL-WNOT-Measure
imports Main List-More
begin
```

The organisation of the development is the following:

- `CDCL_WNOT_Measure.thy` contains the measure used to show the termination the core of CDCL.
- `CDCL_NOT.thy` contains the specification of the rules: the rules are defined, and we proof the correctness and termination for some strategies CDCL.
- `DPLL_NOT.thy` contains the DPLL calculus based on the CDCL version.
- `DPLL_W.thy` contains Weidenbach's version of DPLL and the proof of equivalence between the two DPLL versions.

5.1 Measure

This measure show the termination of the core of CDCL: each step improves the number of literals we know for sure.

This measure can also be seen as the increasing lexicographic order: it is an order on bounded sequences, when each element is bounded. The proof involves a measure like the one defined here (the same?).

definition $\mu_C :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{nat}$ **where**
 $\mu_C s b M \equiv (\sum i=0..<\text{length } M. M!i * b^{\wedge} (s + i - \text{length } M))$

lemma $\mu_C\text{-Nil}[simp]$:
 $\mu_C s b [] = 0$
unfolding $\mu_C\text{-def}$ **by** *auto*

lemma $\mu_C\text{-single}[simp]$:
 $\mu_C s b [L] = L * b^{\wedge} (s - \text{Suc } 0)$
unfolding $\mu_C\text{-def}$ **by** *auto*

lemma *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 μ_C -*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 μ_C -*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 μ_C -*def* .

}

ultimately show *?thesis* **by** *presburger*

qed

lemma μ_C -*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 μ_C -*def* **by** *blast*

moreover then have $\dots = (\sum i=0..< \text{length } M. (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')))$
 $+ (\sum i=\text{length } M..< \text{length } (M @ M'). (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')))$

by (*auto intro!: setsum-add-nat-ivl[symmetric]*)

moreover

have $\forall i \in \{0..< \text{length } M\}. (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')) = M!i * b \wedge (s - \text{length } M' + i - \text{length } M)$

using $\langle s \geq \text{length } (M @ M') \rangle$ **by** (*auto simp add: nth-append ac-simps*)

then have $\mu_C \ (s - \text{length } M') \ b \ M = (\sum i=0..< \text{length } M. (M @ M')!i * b \wedge (s + i - \text{length } (M @ M')))$
 $(M @ M'))$

unfolding μ_C -*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 μ_C -*def* .

}

ultimately show *?thesis* **by** *presburger*

qed

lemma μ_C -cons-non-empty-inf:
assumes M -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* μ_C -cons)

Copy of `~~/src/HOL/ex/NatSum.thy` (but generalized to $0 \leq k$)

lemma *sum-of-powers*: $0 \leq k \implies (k - 1) * (\sum_{i=0..<n. k^\wedge i} = k^\wedge n - (1::nat)$
apply (*cases* $k = 0$)
apply (*cases* n ; *simp*)
by (*induct* n) (*auto simp: Nat.nat-distrib*)

In the degenerated cases, we only have the large inequality holds. In the other cases, the following strict inequality holds:

lemma μ_C -bounded-non-degenerated:
fixes $b :: nat$
assumes
 $b > 0$ **and**
 $M \neq []$ **and**
 M -le: $\forall i < \text{length } M. M!i < b$ **and**
 $s \geq \text{length } M$
shows $\mu_C s b M < b^\wedge s$

proof –

consider ($b1$) $b = 1 \mid (b) b > 1$ **using** $\langle b > 0 \rangle$ **by** (*cases* b) *auto*
then show *?thesis*

proof *cases*

case $b1$

then have $\forall i < \text{length } M. M!i = 0$ **using** M -le **by** *auto*
then have $\mu_C s b M = 0$ **unfolding** μ_C -def **by** *auto*
then show *?thesis* **using** $\langle b > 0 \rangle$ **by** *auto*

next

case b

have $\forall i \in \{0..<\text{length } M\}. M!i * b^\wedge (s + i - \text{length } M) \leq (b-1) * b^\wedge (s + i - \text{length } M)$
using M -le $\langle b > 1 \rangle$ **by** *auto*

then have $\mu_C s b M \leq (\sum_{i=0..<\text{length } M. (b-1) * b^\wedge (s + i - \text{length } M)})$
using $\langle M \neq [] \rangle \langle b > 0 \rangle$ **unfolding** μ_C -def **by** (*auto intro: setsum-mono*)

also

have $\forall i \in \{0..<\text{length } M\}. (b-1) * b^\wedge (s + i - \text{length } M) = (b-1) * b^\wedge i * b^\wedge (s - \text{length } M)$
by (*metis* *Nat.add-diff-assoc2* *add.commute* *assms(4)* *mult.assoc* *power-add*)

then have $(\sum_{i=0..<\text{length } M. (b-1) * b^\wedge (s + i - \text{length } M))$
 $= (\sum_{i=0..<\text{length } M. (b-1) * b^\wedge i * b^\wedge (s - \text{length } M))$
by (*auto simp add: ac-simps*)

also have $\dots = (\sum_{i=0..<\text{length } M. b^\wedge i) * b^\wedge (s - \text{length } M) * (b-1)$
by (*simp add: setsum-left-distrib setsum-right-distrib ac-simps*)

finally have $\mu_C s b M \leq (\sum_{i=0..<\text{length } M. b^\wedge i) * (b-1) * b^\wedge (s - \text{length } M)$
by (*simp add: ac-simps*)

also

have $(\sum_{i=0..<\text{length } M. b^\wedge i) * (b-1) = b^\wedge (\text{length } M) - 1$
using *sum-of-powers*[*of* b $\text{length } M$] $\langle b > 1 \rangle$
by (*auto simp add: ac-simps*)

finally have $\mu_C s b M \leq (b^\wedge (\text{length } M) - 1) * b^\wedge (s - \text{length } M)$
by *auto*

also have $\dots < b^\wedge (\text{length } M) * b^\wedge (s - \text{length } M)$

```

    using <b>1> by auto
  also have ... = b ^ s
    by (metis assms(4) le-add-diff-inverse power-add)
  finally show ?thesis unfolding  $\mu_C$ -def by (auto simp add: ac-simps)
qed
qed

```

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

```

lemma  $\mu_C$ -bounded:
  fixes b :: nat
  assumes
    M-le:  $\forall i < \text{length } M. M!i < b$  and
    s  $\geq \text{length } M$ 
    b > 0
  shows  $\mu_C \ s \ b \ M < b \wedge s$ 
proof -
  consider (M0)  $M = [] \mid (M) \ b > 0$  and  $M \neq []$ 
  using M-le by (cases b, cases M) auto
  then show ?thesis
  proof cases
    case M0
    then show ?thesis using M-le <b > 0> by auto
  next
    case M
    show ?thesis using  $\mu_C$ -bounded-non-degenerated[OF M assms(1,2)] by arith
  qed
qed

```

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

```

lemma  $\mu_C$ -base-0:
  assumes length M  $\leq s$ 
  shows  $\mu_C \ s \ 0 \ M \leq M!0$ 
proof -
  {
    assume s = 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

```

```

lemma finite-bounded-pair-list:
  fixes b :: nat
  shows finite { (ys, xs). length xs < s  $\wedge$  length ys < s  $\wedge$ 
    ( $\forall i < \text{length } xs. xs ! i < b$ )  $\wedge$  ( $\forall i < \text{length } ys. ys ! i < b$ ) }

```

proof –

have $H: \{(ys, xs). \text{length } xs < s \wedge \text{length } ys < s \wedge$
 $(\forall i < \text{length } xs. xs ! i < b) \wedge (\forall i < \text{length } ys. ys ! i < b)\}$
 \subseteq
 $\{xs. \text{length } xs < s \wedge (\forall i < \text{length } xs. xs ! i < b)\} \times$
 $\{xs. \text{length } xs < s \wedge (\forall i < \text{length } xs. xs ! i < b)\}$
by *auto*
moreover have *finite* $\{xs. \text{length } xs < s \wedge (\forall i < \text{length } xs. xs ! i < b)\}$
by (*rule finite-bounded-list*)
ultimately show *?thesis* **by** (*auto simp: finite-subset*)

qed

definition $\nu NOT :: nat \Rightarrow nat \Rightarrow (nat \text{ list} \times nat \text{ list}) \text{ set}$ **where**

$\nu NOT \ s \ base = \{(ys, xs). \text{length } xs < s \wedge \text{length } ys < s \wedge$
 $(\forall i < \text{length } xs. xs ! i < base) \wedge (\forall i < \text{length } ys. ys ! i < base) \wedge$
 $(ys, xs) \in \text{lenlex less-than}\}$

lemma *finite- νNOT* [*simp*]:

finite ($\nu NOT \ s \ base$)

proof –

have $\nu NOT \ s \ base \subseteq \{(ys, xs). \text{length } xs < s \wedge \text{length } ys < s \wedge$
 $(\forall i < \text{length } xs. xs ! i < base) \wedge (\forall i < \text{length } ys. ys ! i < base)\}$
by (*auto simp: νNOT -def*)
moreover have *finite* $\{(ys, xs). \text{length } xs < s \wedge \text{length } ys < s \wedge$
 $(\forall i < \text{length } xs. xs ! i < base) \wedge (\forall i < \text{length } ys. ys ! i < base)\}$
by (*rule finite-bounded-pair-list*)
ultimately show *?thesis* **by** (*auto simp: finite-subset*)

qed

lemma *acyclic- νNOT* : *acyclic* ($\nu NOT \ s \ base$)

apply (*rule acyclic-subset*[*of lenlex less-than $\nu NOT \ s \ base$*])

apply (*rule wf-acyclic*)

by (*auto simp: νNOT -def*)

lemma *wf- νNOT* : *wf* ($\nu NOT \ s \ base$)

by (*rule finite-acyclic-wf*) (*auto simp: acyclic- νNOT*)

end

theory *CDCL-NOT*

imports *List-More Wellfounded-More CDCL-WNOT-Measure Partial-Annotated-Clausal-Logic*

begin

5.2 NOT's CDCL

5.2.1 Auxiliary Lemmas and Measure

We define here some more simplification rules, or rules that have been useful as help for some tactic

lemma *no-dup-cannot-not-lit-and-uminus*:

$\text{no-dup } M \Longrightarrow \text{lit-of } xa = \text{lit-of } x \Longrightarrow x \in \text{set } M \Longrightarrow xa \notin \text{set } M$

by (*metis atm-of-uminus distinct-map inj-on-eq-iff uminus-not-id'*)

lemma *atms-of-ms-single-atm-of*[*simp*]:

$\text{atms-of-ms } \{\text{unmark } L \mid L. P \ L\} = \text{atm-of } \{\text{lit-of } L \mid L. P \ L\}$

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

lemma *atms-of-uminus-lit-atm-of-lit-of*:
 $atms-of \{ \# - lit-of x. x \in \# A \# \} = atm-of ' (lit-of ' (set-mset A))$
unfolding *atms-of-def* **by** (auto simp add: Fun.image-comp)

lemma *atms-of-ms-single-image-atm-of-lit-of*:
 $atms-of-ms (unmark-s A) = atm-of ' (lit-of ' A)$
unfolding *atms-of-ms-def* **by** auto

5.2.2 Initial definitions

The state

We define here an abstraction over operation on the state we are manipulating.

locale *dp11-state-ops* =
fixes
 $trail :: 'st \Rightarrow ('v, unit) \text{ ann-lits}$ **and**
 $clauses_{NOT} :: 'st \Rightarrow 'v \text{ clauses}$ **and**
 $prepend-trail :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow 'st$ **and**
 $tl-trail :: 'st \Rightarrow 'st$ **and**
 $add-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ **and**
 $remove-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$
begin
abbreviation $state_{NOT} :: 'st \Rightarrow ('v, unit) \text{ ann-lit list} \times 'v \text{ clauses}$ **where**
 $state_{NOT} S \equiv (trail S, clauses_{NOT} S)$
end

NOT's state is basically a pair composed of the trail (i.e. the candidate model) and the set of clauses. We abstract this state to convert this state to other states. like Weidenbach's five-tuple.

locale *dp11-state* =
dp11-state-ops
 $trail \ clauses_{NOT} \ prepend-trail \ tl-trail \ add-cls_{NOT} \ remove-cls_{NOT}$ — related to the state
for
 $trail :: 'st \Rightarrow ('v, unit) \text{ ann-lits}$ **and**
 $clauses_{NOT} :: 'st \Rightarrow 'v \text{ clauses}$ **and**
 $prepend-trail :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow 'st$ **and**
 $tl-trail :: 'st \Rightarrow 'st$ **and**
 $add-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ **and**
 $remove-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st$ +
assumes
 $prepend-trail_{NOT}$:
 $state_{NOT} (prepend-trail L st) = (L \# trail st, clauses_{NOT} st)$ **and**
 $tl-trail_{NOT}$:
 $state_{NOT} (tl-trail st) = (tl (trail st), clauses_{NOT} st)$ **and**
 $add-cls_{NOT}$:
 $state_{NOT} (add-cls_{NOT} C st) = (trail st, \{ \# C \# \} + clauses_{NOT} st)$ **and**
 $remove-cls_{NOT}$:
 $state_{NOT} (remove-cls_{NOT} C st) = (trail st, removeAll-mset C (clauses_{NOT} st))$
begin
lemma
 $trail-prepend-trail[simp]$:
 $trail (prepend-trail L st) = L \# trail st$
and
 $trail-tl-trail_{NOT}[simp]$: $trail (tl-trail st) = tl (trail st)$ **and**
 $trail-add-cls_{NOT}[simp]$: $trail (add-cls_{NOT} C st) = trail st$ **and**

trail-remove-cls_{NOT}[simp]: *trail (remove-cls_{NOT} C st) = trail st* **and**

clauses-prepend-trail[simp]:

clauses_{NOT} (prepend-trail L st) = clauses_{NOT} st

and

clauses-tl-trail[simp]: *clauses_{NOT} (tl-trail st) = clauses_{NOT} st* **and**

clauses-add-cls_{NOT}[simp]:

clauses_{NOT} (add-cls_{NOT} C st) = {#C#} + clauses_{NOT} st **and**

clauses-remove-cls_{NOT}[simp]:

clauses_{NOT} (remove-cls_{NOT} C st) = removeAll-mset C (clauses_{NOT} st)

using *prepend-trail_{NOT}*[of L st] *tl-trail_{NOT}*[of st] *add-cls_{NOT}*[of C st] *remove-cls_{NOT}*[of C st]

by (*cases state_{NOT} st; auto*)**+**

We define the following function doing the backtrack in the trail:

function *reduce-trail-to_{NOT}* :: 'a list \Rightarrow 'st \Rightarrow 'st **where**

reduce-trail-to_{NOT} F S =

(if length (trail S) = length F \vee trail S = [] then S else reduce-trail-to_{NOT} F (tl-trail S))

by *fast+*

termination by (*relation measure* ($\lambda(-, S). \text{length (trail S)}$)) *auto*

declare *reduce-trail-to_{NOT}.simps*[simp del]

Then we need several lemmas about the *reduce-trail-to_{NOT}*.

lemma

shows

reduce-trail-to_{NOT}-Nil[simp]: *trail S = [] \implies reduce-trail-to_{NOT} F S = S* **and**

reduce-trail-to_{NOT}-eq-length[simp]: *length (trail S) = length F \implies reduce-trail-to_{NOT} F S = S*

by (*auto simp: reduce-trail-to_{NOT}.simps*)

lemma *reduce-trail-to_{NOT}-length-ne*[simp]:

length (trail S) \neq length F \implies trail S \neq [] \implies

reduce-trail-to_{NOT} F S = reduce-trail-to_{NOT} F (tl-trail S)

by (*auto simp: reduce-trail-to_{NOT}.simps*)

lemma *trail-reduce-trail-to_{NOT}-length-le*:

assumes *length F > length (trail S)*

shows *trail (reduce-trail-to_{NOT} F S) = []*

using *assms by (induction F S rule: reduce-trail-to_{NOT}.induct)*

(simp add: less-imp-diff-less reduce-trail-to_{NOT}.simps)

lemma *trail-reduce-trail-to_{NOT}-Nil*[simp]:

trail (reduce-trail-to_{NOT} [] S) = []

by (*induction [] S rule: reduce-trail-to_{NOT}.induct*)

(simp add: less-imp-diff-less reduce-trail-to_{NOT}.simps)

lemma *clauses-reduce-trail-to_{NOT}-Nil*:

clauses_{NOT} (reduce-trail-to_{NOT} [] S) = clauses_{NOT} S

by (*induction [] S rule: reduce-trail-to_{NOT}.induct*)

(simp add: less-imp-diff-less reduce-trail-to_{NOT}.simps)

lemma *trail-reduce-trail-to_{NOT}-drop*:

trail (reduce-trail-to_{NOT} F S) =

(if length (trail S) \geq length F

then drop (length (trail S) - length F) (trail S)

else [])

apply (*induction F S rule: reduce-trail-to_{NOT}.induct*)

apply (*rename-tac F S, 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-toNOT F S) = F
  using assms by (auto simp: trail-reduce-trail-toNOT-drop)

```

lemma *reduce-trail-to_{NOT}-clauses[simp]*:

```

  clausesNOT (reduce-trail-toNOT F S) = clausesNOT S
  by (induction F S rule: reduce-trail-toNOT.induct)
  (simp add: less-imp-diff-less reduce-trail-toNOT.simps)

```

lemma *trail-eq-reduce-trail-to_{NOT}-eq*:

```

  trail S = trail T  $\implies$  trail (reduce-trail-toNOT F S) = trail (reduce-trail-toNOT F T)
  apply (induction F S arbitrary: T rule: reduce-trail-toNOT.induct)
  by (metis trail-tl-trailNOT reduce-trail-toNOT-eq-length reduce-trail-toNOT-length-ne
      reduce-trail-toNOT-Nil)

```

lemma *trail-reduce-trail-to_{NOT}-add-cl_{NOT}[simp]*:

```

  no-dup (trail S)  $\implies$ 
  trail (reduce-trail-toNOT F (add-clNOT C S)) = trail (reduce-trail-toNOT F S)
  by (rule trail-eq-reduce-trail-toNOT-eq) simp

```

lemma *reduce-trail-to_{NOT}-trail-tl-trail-decomp[simp]*:

```

  trail S = F' @ Decided K # F  $\implies$ 
  trail (reduce-trail-toNOT F (tl-trail S)) = F
  apply (rule reduce-trail-toNOT-skip-beginning[of - tl (F' @ Decided K # [])])
  by (cases F') (auto simp add:tl-append reduce-trail-toNOT-skip-beginning)

```

lemma *reduce-trail-to_{NOT}-length*:

```

  length M = length M'  $\implies$  reduce-trail-toNOT M S = reduce-trail-toNOT M' S
  apply (induction M S rule: reduce-trail-toNOT.induct)
  by (simp add: reduce-trail-toNOT.simps)

```

abbreviation *trail-weight where*

trail-weight S \equiv map (($\lambda l.$ 1 + length l) o snd) (get-all-ann-decomposition (trail S))

As we are defining abstract states, the Isabelle equality about them is too strong: we want the weaker equivalence stating that two states are equal if they cannot be distinguished, i.e. given the getter *trail* and *clauses_{NOT}* do not distinguish them.

definition *state-eq_{NOT}* :: 'st \Rightarrow 'st \Rightarrow bool (**infix** \sim 50) **where**

$S \sim T \longleftrightarrow \text{trail } S = \text{trail } T \wedge \text{clauses}_{\text{NOT}} S = \text{clauses}_{\text{NOT}} T$

lemma *state-eq_{NOT}-ref[simp]*:

```

  S  $\sim$  S
  unfolding state-eqNOT-def by auto

```

lemma *state-eq_{NOT}-sym*:

$S \sim T \longleftrightarrow T \sim S$

unfolding *state-eq_{NOT}-def* **by** *auto*

lemma *state-eq_{NOT}-trans*:

$S \sim T \implies T \sim U \implies S \sim U$

unfolding *state-eq_{NOT}-def* **by** *auto*

lemma

shows

state-eq_{NOT}-trail: $S \sim T \implies \text{trail } S = \text{trail } T$ **and**

state-eq_{NOT}-clauses: $S \sim T \implies \text{clauses}_{\text{NOT}} S = \text{clauses}_{\text{NOT}} T$

unfolding *state-eq_{NOT}-def* **by** *auto*

lemmas *state-simp_{NOT}[simp]* = *state-eq_{NOT}-trail* *state-eq_{NOT}-clauses*

lemma *reduce-trail-to_{NOT}-state-eq_{NOT}-compatible*:

assumes *ST*: $S \sim T$

shows *reduce-trail-to_{NOT}* $F S \sim \text{reduce-trail-to}_{\text{NOT}} F T$

proof –

have *clauses_{NOT}* (*reduce-trail-to_{NOT}* $F S$) = *clauses_{NOT}* (*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

end

Definition of the operation

Each possible is in its own locale.

locale *propagate-ops* =

dpll-state *trail* *clauses_{NOT}* *prepend-trail* *tl-trail* *add-cl_s_{NOT}* *remove-cl_s_{NOT}*

for

trail :: '*st* \Rightarrow ('*v*, *unit*) *ann-lits* **and**

clauses_{NOT} :: '*st* \Rightarrow '*v* *clauses* **and**

prepend-trail :: ('*v*, *unit*) *ann-lit* \Rightarrow '*st* \Rightarrow '*st* **and**

tl-trail :: '*st* \Rightarrow '*st* **and**

add-cl_s_{NOT} :: '*v* *clause* \Rightarrow '*st* \Rightarrow '*st* **and**

remove-cl_s_{NOT} :: '*v* *clause* \Rightarrow '*st* \Rightarrow '*st* +

fixes

propagate-cond :: ('*v*, *unit*) *ann-lit* \Rightarrow '*st* \Rightarrow *bool*

begin

inductive *propagate_{NOT}* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **where**

propagate_{NOT}[intro]: $C + \{\#L\} \in \# \text{clauses}_{\text{NOT}} S \implies \text{trail } S \models_{\text{as}} C \text{Not } C$

$\implies \text{undefined-lit } (\text{trail } S) L$

$\implies \text{propagate-cond } (\text{Propagated } L ()) S$

$\implies T \sim \text{prepend-trail } (\text{Propagated } L ()) S$

$\implies \text{propagate}_{\text{NOT}} S T$

inductive-cases *propagate_{NOT}E[elim]*: *propagate_{NOT}* $S T$

end

locale *decide-ops* =

dpll-state *trail* *clauses_{NOT}* *prepend-trail* *tl-trail* *add-cl_s_{NOT}* *remove-cl_s_{NOT}*

for

```

trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
clausesNOT :: 'st  $\Rightarrow$  'v clauses and
prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st
begin
inductive decideNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
decideNOT[intro]: undefined-lit (trail S) L  $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)
 $\Rightarrow$  T  $\sim$  prepend-trail (Decided L) S
 $\Rightarrow$  decideNOT S T

inductive-cases decideNOTE[elim]: decideNOT S S'
end

locale backjumping-ops =
dpll-state trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
for
trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
clausesNOT :: 'st  $\Rightarrow$  'v clauses and
prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

inductive backjump where
trail S = F' @ Decided K# F
 $\Rightarrow$  T  $\sim$  prepend-trail (Propagated L ()) (reduce-trail-toNOT F S)
 $\Rightarrow$  C  $\in$  # clausesNOT S
 $\Rightarrow$  trail S  $\models_{as}$  CNot C
 $\Rightarrow$  undefined-lit F L
 $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))
 $\Rightarrow$  clausesNOT 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

```

The condition $\text{atm-of } L \in \text{atms-of-mm } (\text{clauses}_{\text{NOT}} S) \cup \text{atm-of ' } (\text{lits-of-l } (\text{trail } S))$ is not implied by the condition $\text{clauses}_{\text{NOT}} S \models_{pm} C' + \{\#L\# \}$ (no negation).

end

5.2.3 DPLL with backjumping

```

locale dpll-with-backjumping-ops =
propagate-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds +
decide-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT +
backjumping-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT backjump-conds
for
trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
clausesNOT :: 'st  $\Rightarrow$  'v clauses and
prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and

```


$add-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$
 $remove-cls_{NOT} :: 'v \text{ clause} \Rightarrow 'st \Rightarrow 'st \text{ and}$
 $inv :: 'st \Rightarrow bool \text{ and}$
 $backjump-conds :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow bool \text{ and}$
 $propagate-conds :: ('v, unit) \text{ ann-lit} \Rightarrow 'st \Rightarrow bool +$
assumes
 $bj\text{-can-jump:}$
 $\bigwedge S \ C \ F' \ K \ F \ L.$
 $inv \ S \Longrightarrow$
 $no\text{-dup} \ (trail \ S) \Longrightarrow$
 $trail \ S = F' @ Decided \ K \ \# \ F \Longrightarrow$
 $C \in \# \ clauses_{NOT} \ S \Longrightarrow$
 $trail \ S \models_{as} CNot \ C \Longrightarrow$
 $undefined\text{-lit} \ F \ L \Longrightarrow$
 $atm\text{-of} \ L \in atm\text{-of}\text{-mm} \ (clauses_{NOT} \ S) \cup atm\text{-of} \ ' \ (lits\text{-of}\text{-l} \ (F' @ Decided \ K \ \# \ F)) \Longrightarrow$
 $clauses_{NOT} \ S \models_{pm} C' + \{\#L\# \} \Longrightarrow$
 $F \models_{as} CNot \ C' \Longrightarrow$
 $\neg no\text{-step} \ backjump \ S$
begin

We cannot add a like condition $atms\text{-of} \ C' \subseteq atm\text{-of}\text{-ms} \ N$ to ensure that we can backjump even if the last decision variable has disappeared from the set of clauses.

The part of the condition $atm\text{-of} \ L \in atm\text{-of} \ ' \ lits\text{-of}\text{-l} \ (F' @ Decided \ K \ \# \ F)$ is important, otherwise you are not sure that you can backtrack.

Definition

We define $dp\text{ll}$ with backjumping:

inductive $dp\text{ll}\text{-bj} :: 'st \Rightarrow 'st \Rightarrow bool \text{ for } S :: 'st \text{ where}$
 $bj\text{-decide}_{NOT}: decide_{NOT} \ S \ S' \Longrightarrow dp\text{ll}\text{-bj} \ S \ S' \mid$
 $bj\text{-propagate}_{NOT}: propagate_{NOT} \ S \ S' \Longrightarrow dp\text{ll}\text{-bj} \ S \ S' \mid$
 $bj\text{-backjump}: backjump \ S \ S' \Longrightarrow dp\text{ll}\text{-bj} \ S \ S'$

lemmas $dp\text{ll}\text{-bj}\text{-induct} = dp\text{ll}\text{-bj}.\text{induct}[\text{split-format}(\text{complete})]$

thm $dp\text{ll}\text{-bj}\text{-induct}[OF \ dp\text{ll}\text{-with-backjumping-ops-axioms}]$

lemma $dp\text{ll}\text{-bj}\text{-all-induct}[\text{consumes } 2, \text{ case-names } decide_{NOT} \ propagate_{NOT} \ backjump]:$

fixes $S \ T :: 'st$

assumes

$dp\text{ll}\text{-bj} \ S \ T \text{ and}$

$inv \ S$

$\bigwedge L \ T. \text{undefined-lit} \ (trail \ S) \ L \Longrightarrow atm\text{-of} \ L \in atm\text{-of}\text{-mm} \ (clauses_{NOT} \ S)$

$\Longrightarrow T \sim \text{prepend-trail} \ (Decided \ L) \ S$

$\Longrightarrow P \ S \ T \text{ and}$

$\bigwedge C \ L \ T. C + \{\#L\# \} \in \# \ clauses_{NOT} \ S \Longrightarrow trail \ S \models_{as} CNot \ C \Longrightarrow \text{undefined-lit} \ (trail \ S) \ L$

$\Longrightarrow T \sim \text{prepend-trail} \ (Propagated \ L \ ()) \ S$

$\Longrightarrow P \ S \ T \text{ and}$

$\bigwedge C \ F' \ K \ F \ L \ C' \ T. C \in \# \ clauses_{NOT} \ S \Longrightarrow F' @ Decided \ K \ \# \ F \models_{as} CNot \ C$

$\Longrightarrow trail \ S = F' @ Decided \ K \ \# \ F$

$\Longrightarrow \text{undefined-lit} \ F \ L$

$\Longrightarrow atm\text{-of} \ L \in atm\text{-of}\text{-mm} \ (clauses_{NOT} \ S) \cup atm\text{-of} \ ' \ (lits\text{-of}\text{-l} \ (F' @ Decided \ K \ \# \ F))$

$\Longrightarrow clauses_{NOT} \ S \models_{pm} C' + \{\#L\# \}$

$\Longrightarrow F \models_{as} CNot \ C'$

$\Longrightarrow T \sim \text{prepend-trail} \ (Propagated \ L \ ()) \ (\text{reduce-trail-to}_{NOT} \ F \ S)$

$\Longrightarrow P \ S \ T$

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* *backjump* E) **using** *assms*(5) $\langle \text{inv } S \rangle$ **by** *simp*

Basic properties

First, some better suited induction principle **lemma** *dpll-bj-clauses*:

assumes *dpll-bj* $S \ T$ **and** *inv* S
shows *clauses*_{NOT} $S = \text{clauses}_{NOT} \ T$
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 $I \models^{sm} \text{clauses}_{NOT} \ S \longleftrightarrow I \models^{sm} \text{clauses}_{NOT} \ T$
using *assms* **by** (*induction* *rule*: *dpll-bj-all-induct*) *auto*

Clauses **lemma** *dpll-bj-atms-of-ms-clauses-inv*:

assumes
 dpll-bj $S \ T$ **and**
 inv S
shows *atms-of-mm* (*clauses*_{NOT} S) = *atms-of-mm* (*clauses*_{NOT} T)
using *assms* **by** (*induction* *rule*: *dpll-bj-all-induct*) *auto*

lemma *dpll-bj-atms-in-trail*:

assumes
 dpll-bj $S \ T$ **and**
 inv S **and**
 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} \ S)$
shows $\text{atm-of } ' (\text{lits-of-l } (\text{trail } T)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} \ S)$
using *assms* **by** (*induction* *rule*: *dpll-bj-all-induct*)
(auto simp: in-plus-implies-atm-of-on-atms-of-ms reduce-trail-to_{NOT}-skip-beginning)

lemma *dpll-bj-atms-in-trail-in-set*:

assumes *dpll-bj* $S \ T$ **and**
 inv S **and**
 $\text{atms-of-mm } (\text{clauses}_{NOT} \ S) \subseteq A$ **and**
 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq A$
shows $\text{atm-of } ' (\text{lits-of-l } (\text{trail } T)) \subseteq A$
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 (clausesNOT S) (get-all-ann-decomposition (trail S))
  shows all-decomposition-implies-m (clausesNOT T) (get-all-ann-decomposition (trail T))
  using assms(1,2)
proof (induction rule:dpll-bj-all-induct)
  case decideNOT
  then show ?case using decomp by auto
next
  case (propagateNOT 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 = clausesNOT S
  obtain a y l where ay: get-all-ann-decomposition ?M' = (a, y) # l
    by (cases get-all-ann-decomposition ?M') fastforce+
  then have M': ?M' = y @ a using get-all-ann-decomposition-decomp[of ?M'] by auto
  have M: get-all-ann-decomposition (trail S) = (a, tl y) # l
    using ay undef by (cases get-all-ann-decomposition (trail S)) auto
  have y0: y = (Propagated L ()) # (tl y)
    using ay undef by (auto simp add: M)
  from arg-cong[OF this, of set] have y[simp]: set y = insert (Propagated L ()) (set (tl y))
    by simp
  have tr-S: trail S = tl y @ a
    using arg-cong[OF M', of tl] y0 M get-all-ann-decomposition-decomp by force
  have a-Un-N-M: unmark-l a ∪ set-mset ?N ⊨ps unmark-l (tl y)
    using decomp ay unfolding all-decomposition-implies-def by (simp add: M) +

  moreover have unmark-l a ∪ set-mset ?N ⊨p {#L#} (is ?I ⊨p -)
  proof (rule true-clss-clss-plus-CNot)
    show ?I ⊨p C + {#L#}
      using propa propagateNOT.prems by (auto dest!: true-clss-clss-in-imp-true-clss-clss)
  next
    have unmark-l ?M' ⊨ps CNot C
      using (trail S ⊨as CNot C) undef by (auto simp add: true-annots-true-clss-clss)
    have a1: unmark-l a ∪ unmark-l (tl y) ⊨ps CNot C
      using propagateNOT.hyps(2) tr-S true-annots-true-clss-clss
      by (force simp add: image-Un sup-commute)
    then have unmark-l a ∪ set-mset (clausesNOT S) ⊨ps unmark-l a ∪ unmark-l (tl y)
      using a-Un-N-M true-clss-clss-def by blast
    then show unmark-l a ∪ set-mset (clausesNOT S) ⊨ps CNot C
      using a1 by (meson true-clss-clss-left-right true-clss-clss-union-and
        true-clss-clss-union-l-r)
  qed
  ultimately have unmark-l a ∪ set-mset ?N ⊨ps unmark-l ?M'
    unfolding M' by (auto simp add: all-in-true-clss-clss image-Un)
  then show ?case
    using decomp T M undef unfolding ay all-decomposition-implies-def by (auto simp add: ay)
next
  case (backjump C F' K F L D T) note confl = this(2) and tr = this(3) and undef = this(4) and
    L = this(5) and N-C = this(6) and vars-D = this(5) and T = this(8)
  have decomp: all-decomposition-implies-m (clausesNOT S) (get-all-ann-decomposition F)
    using decomp unfolding tr all-decomposition-implies-def
    by (metis (no-types, lifting) get-all-ann-decomposition.simps(1)
      get-all-ann-decomposition-never-empty hd-Cons-tl insert-iff list.sel(3) list.set(2)
      tl-get-all-ann-decomposition-skip-some)

  obtain a b li where F: get-all-ann-decomposition F = (a, b) # li
    by (cases get-all-ann-decomposition F) auto
  have F = b @ a

```

```

  using get-all-ann-decomposition-decomp[of F a b] F by auto
have a-N-b:unmark-l a  $\cup$  set-mset (clausesNOT S)  $\models_{ps}$  unmark-l b
  using decomp unfolding all-decomposition-implies-def by (auto simp add: F)

have F-D: unmark-l F  $\models_{ps}$  CNot D
  using  $\langle F \models_{as} CNot D \rangle$  by (simp add: true-annots-true-clss-clss)
then have unmark-l a  $\cup$  unmark-l b  $\models_{ps}$  CNot D
  unfolding  $\langle F = b @ a \rangle$  by (simp add: image-Un sup commute)
have a-N-CNot-D: unmark-l a  $\cup$  set-mset (clausesNOT S)  $\models_{ps}$  CNot D  $\cup$  unmark-l b
  apply (rule true-clss-clss-left-right)
  using a-N-b F-D unfolding  $\langle F = b @ a \rangle$  by (auto simp add: image-Un ac-simps)

have a-N-D-L: unmark-l a  $\cup$  set-mset (clausesNOT S)  $\models_p$  D+{#L#}
  by (simp add: N-C)
have unmark-l a  $\cup$  set-mset (clausesNOT S)  $\models_p$  {#L#}
  using a-N-D-L a-N-CNot-D by (blast intro: true-clss-clss-plus-CNot)
then show ?case
  using decomp T tr undef unfolding all-decomposition-implies-def by (auto simp add: F)
qed

```

Termination

Using a proper measure lemma length-get-all-ann-decomposition-append-Decided:

```

length (get-all-ann-decomposition (F' @ Decided K # F)) =
  length (get-all-ann-decomposition F')
+ length (get-all-ann-decomposition (Decided K # F))
- 1

```

by (induction F' rule: ann-lit-list-induct) auto

lemma take-length-get-all-ann-decomposition-decided-sandwich:

```

take (length (get-all-ann-decomposition F))
  (map (f o snd) (rev (get-all-ann-decomposition (F' @ Decided K # F))))
=
  map (f o snd) (rev (get-all-ann-decomposition F))

```

proof (induction F' rule: ann-lit-list-induct)

```

case Nil
then show ?case by auto

```

next

```

case (Decided K)
then show ?case by (simp add: length-get-all-ann-decomposition-append-Decided)

```

next

```

case (Propagated L m F') note IH = this(1)
obtain a b l where F': get-all-ann-decomposition (F' @ Decided K # F) = (a, b) # l
  by (cases get-all-ann-decomposition (F' @ Decided K # F)) auto
have length (get-all-ann-decomposition F) - length l = 0
  using length-get-all-ann-decomposition-append-Decided[of F' K F]
  unfolding F' by (cases get-all-ann-decomposition F') auto
then show ?case
  using IH by (simp add: F')

```

qed

lemma length-get-all-ann-decomposition-length:

```

length (get-all-ann-decomposition M)  $\leq$  1 + length M
by (induction M rule: ann-lit-list-induct) auto

```

lemma *length-in-get-all-ann-decomposition-bounded*:
assumes $i: i \in \text{set } (\text{trail-weight } S)$
shows $i \leq \text{Suc } (\text{length } (\text{trail } S))$
proof –
obtain $a \ b$ **where**
 $(a, b) \in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S))$ **and**
 $ib: i = \text{Suc } (\text{length } b)$
using i **by** *auto*
then obtain c **where** $\text{trail } S = c @ b @ a$
using *get-all-ann-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-ann-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* :: $'b \text{ literal multiset set} \Rightarrow 'a \text{ list} \Rightarrow \text{nat}$ **where**
 $\text{unassigned-lit } N \ M \equiv \text{card } (\text{atms-of-ms } N) - \text{length } M$

lemma *dp11-bj-trail-mes-increasing-prop*:

fixes $M :: ('v, \text{unit}) \text{ ann-lits}$ **and** $N :: 'v \text{ clauses}$

assumes

$\text{dp11-bj } S \ T$ **and**

$\text{inv } S$ **and**

$NA: \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \subseteq \text{atms-of-ms } A$ **and**

$MA: \text{atm-of } ' \text{ lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d}: \text{no-dup } (\text{trail } S)$ **and**

$\text{finite}: \text{finite } A$

shows $\mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$

$> \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$

using *assms(1,2)*

proof (*induction rule: dp11-bj-all-induct*)

case ($\text{propagate}_{NOT} \ C \ L$) **note** $CLN = \text{this}(1)$ **and** $MC = \text{this}(2)$ **and** $\text{undef-L} = \text{this}(3)$ **and** $T = \text{this}(4)$

have $\text{incl}: \text{atm-of } ' \text{ lits-of-l } (\text{Propagated } L \ ()) \# \text{trail } S \subseteq \text{atms-of-ms } A$

using $\text{propagate}_{NOT} \ \text{dp11-bj-atms-in-trail-in-set} \ \text{bj-propagate}_{NOT} \ NA \ MA \ CLN$

by (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

have $\text{no-dup}: \text{no-dup } (\text{Propagated } L \ ()) \# \text{trail } S$

using $\text{defined-lit-map } n\text{-d } \text{undef-L}$ **by** *auto*

obtain $a \ b \ l$ **where** $M: \text{get-all-ann-decomposition } (\text{trail } S) = (a, b) \# l$

by (*cases get-all-ann-decomposition (trail S) auto*)

have $b\text{-le-M}: \text{length } b \leq \text{length } (\text{trail } S)$

using $\text{get-all-ann-decomposition-decomp[of trail S]}$ **by** (*simp add: M*)

have $\text{finite } (\text{atms-of-ms } A)$ **using** finite **by** *simp*

then have $\text{length } (\text{Propagated } L \ ()) \# \text{trail } S \leq \text{card } (\text{atms-of-ms } A)$

using $\text{incl } \text{finite}$ **unfolding** $\text{no-dup-length-eq-card-atm-of-lits-of-l[OF no-dup]}$

```

    by (simp add: card-mono)
  then have latm: unassigned-lit A b = Suc (unassigned-lit A (Propagated L d # b))
    using b-le-M by auto
  then show ?case using T undef-L by (auto simp: latm M  $\mu_C$ -cons)
next
case (decideNOT L) note undef-L = this(1) and MC = this(2) and T = this(3)
have incl: atm-of ' lits-of-l (Decided L # (trail S))  $\subseteq$  atms-of-ms A
  using dpll-bj-atms-in-trail-in-set bj-decideNOT decideNOT.decideNOT[OF decideNOT.hyps] NA MA
MC
  by auto

have no-dup: no-dup (Decided L # (trail S))
  using defined-lit-map n-d undef-L by auto
obtain a b l where M: get-all-ann-decomposition (trail S) = (a, b) # l
  by (cases get-all-ann-decomposition (trail S)) auto

then have length (Decided L # (trail S))  $\leq$  card (atms-of-ms A)
  using incl finite unfolding no-dup-length-eq-card-atm-of-lits-of-l[OF no-dup]
  by (simp add: card-mono)
show ?case using T undef-L by (simp add:  $\mu_C$ -cons)
next
case (backjump C F' K F L C' T) note undef-L = this(4) and MC = this(1) and tr-S = this(3)
and
  L = this(5) and T = this(8)
have incl: atm-of ' lits-of-l (Propagated L () # F)  $\subseteq$  atms-of-ms A
  using dpll-bj-atms-in-trail-in-set NA MA L by (auto simp: tr-S)

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-ann-decomposition (trail S) = (a, b) # l
  by (cases get-all-ann-decomposition (trail S)) auto
have b-le-M: length b  $\leq$  length (trail S)
  using get-all-ann-decomposition-decomp[of trail S] by (simp add: M)
have fin-atms-A: finite (atms-of-ms A) using finite by simp

then have F-le-A: length (Propagated L () # F)  $\leq$  card (atms-of-ms A)
  using incl finite unfolding no-dup-length-eq-card-atm-of-lits-of-l[OF no-dup]
  by (simp add: card-mono)
have tr-S-le-A: length (trail S)  $\leq$  card (atms-of-ms A)
  using n-d MA by (metis fin-atms-A card-mono no-dup-length-eq-card-atm-of-lits-of-l)
obtain a b l where F: get-all-ann-decomposition F = (a, b) # l
  by (cases get-all-ann-decomposition F) auto
then have F = b @ a
  using get-all-ann-decomposition-decomp[of Propagated L () # F a
    Propagated L () # b] by simp
then have latm: unassigned-lit A b = Suc (unassigned-lit A (Propagated L () # b))
  using F-le-A by simp
obtain rem where
  rem: map ( $\lambda a$ . Suc (length (snd a))) (rev (get-all-ann-decomposition (F' @ Decided K # F)))
  = map ( $\lambda a$ . Suc (length (snd a))) (rev (get-all-ann-decomposition F)) @ rem
  using take-length-get-all-ann-decomposition-decided-sandwich[of F  $\lambda a$ . Suc (length a) F' K]
  unfolding o-def by (metis append-take-drop-id)
then have rem: map ( $\lambda a$ . Suc (length (snd a)))
  (get-all-ann-decomposition (F' @ Decided K # F))
  = rev rem @ map ( $\lambda a$ . Suc (length (snd a))) ((get-all-ann-decomposition F))
  by (simp add: rev-map[symmetric] rev-swap)

```

```

have length (rev rem @ map (λa. Suc (length (snd a))) (get-all-ann-decomposition F))
  ≤ Suc (card (atms-of-ms A))
using arg-cong[OF rem, of length] tr-S-le-A
length-get-all-ann-decomposition-length[of F' @ Decided K # F] tr-S by auto
moreover
{ fix i :: nat and xs :: 'a list
  have i < length xs ⇒ length xs - Suc i < length xs
  by auto
  then have H: i < length xs ⇒ rev xs ! i ∈ set xs
  using rev-nth[of i xs] unfolding in-set-conv-nth by (force simp add: in-set-conv-nth)
} note H = this
have ∀ i < length rem. rev rem ! i < card (atms-of-ms A) + 2
  using tr-S-le-A length-in-get-all-ann-decomposition-bounded[of - S] unfolding tr-S
  by (force simp add: o-def rem dest!: H intro: length-get-all-ann-decomposition-length)
ultimately show ?case
  using μC-bounded[of rev rem card (atms-of-ms A)+2 unassigned-lit A l] T undef-L
  by (simp add: rem μC-append μC-cons F tr-S)
qed

```

lemma *dpll-bj-trail-mes-decreasing-prop*:

assumes *dpll*: *dpll-bj S T* **and** *inv*: *inv S* **and**
N-A: *atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A* **and**
M-A: *atm-of ' lits-of-l (trail S) ⊆ atms-of-ms A* **and**
nd: *no-dup (trail S)* **and**
fin-A: *finite A*

shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $\quad < (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$

proof –

```

let ?b = 2 + card (atms-of-ms A)
let ?s = 1 + card (atms-of-ms A)
let ?μ = μC ?s ?b
have M'-A: atm-of ' lits-of-l (trail T) ⊆ atms-of-ms A
  by (meson M-A N-A dpll dpll-bj-atms-in-trail-in-set inv)
have nd': no-dup (trail T)
  using ⟨dpll-bj S T⟩ dpll-bj-no-dup nd inv by blast
{ fix i :: nat and xs :: 'a list
  have i < length xs ⇒ length xs - Suc i < length xs
  by auto
  then have H: i < length xs ⇒ xs ! i ∈ set xs
  using rev-nth[of i xs] unfolding in-set-conv-nth by (force simp add: in-set-conv-nth)
} note H = this

```

```

have l-M-A: length (trail S) ≤ card (atms-of-ms A)
  by (simp add: fin-A M-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd)
have l-M'-A: length (trail T) ≤ card (atms-of-ms A)
  by (simp add: fin-A M'-A card-mono no-dup-length-eq-card-atm-of-lits-of-l nd')
have l-trail-weight-M: length (trail-weight T) ≤ 1 + card (atms-of-ms A)
  using l-M'-A length-get-all-ann-decomposition-length[of trail T] by auto
have bounded-M: ∀ i < length (trail-weight T). (trail-weight T)! i < card (atms-of-ms A) + 2
  using length-in-get-all-ann-decomposition-bounded[of - T] l-M'-A
  by (metis (no-types, lifting) H Nat.le-trans add-2-eq-Suc' not-le not-less-eq-eq)

```

```

from dpll-bj-trail-mes-increasing-prop[OF dpll inv N-A M-A nd fin-A]
have μC ?s ?b (trail-weight S) < μC ?s ?b (trail-weight T) by simp

```

moreover from μ_C -bounded[*OF bounded-M l-trail-weight-M*]
have μ_C ?s ?b (trail-weight *T*) \leq ?b \wedge ?s **by** *auto*
ultimately show ?thesis **by** *linarith*
qed

lemma *wf-dpll-bj*:
assumes *fin*: finite *A*
shows *wf* $\{(T, S). \text{dpll-bj } S \text{ } T$
 $\wedge \text{atms-of-mm } (\text{clauses}_{NOT} \text{ } S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{no-dup } (\text{trail } S) \wedge \text{inv } S\}$
(is *wf* ?*A*)
proof (rule *wf-bounded-measure*[of -
 $\lambda \cdot. (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\lambda S. \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)]$)
fix *a b* :: 'st
let ?b = 2 + card (atms-of-ms *A*)
let ?s = 1 + card (atms-of-ms *A*)
let ? μ = μ_C ?s ?b
assume *ab*: (*b*, *a*) \in ?*A*

have *fin-A*: finite (atms-of-ms *A*)
using *fin* **by** *auto*
have
dpll-bj: *dpll-bj* *a b* **and**
N-A: atms-of-mm (clauses_{NOT} *a*) \subseteq atms-of-ms *A* **and**
M-A: atm-of ' lits-of-l (trail *a*) \subseteq atms-of-ms *A* **and**
nd: no-dup (trail *a*) **and**
inv: inv *a*
using *ab* **by** *auto*

have *M'-A*: atm-of ' lits-of-l (trail *b*) \subseteq atms-of-ms *A*
by (meson *M-A N-A* $\langle \text{dpll-bj } a \text{ } b \rangle \text{dpll-bj-atms-in-trail-in-set inv}$)
have *nd'*: no-dup (trail *b*)
using $\langle \text{dpll-bj } a \text{ } b \rangle \text{dpll-bj-no-dup nd inv}$ **by** *blast*
{ fix *i* :: nat **and** *xs* :: 'a list
have *i* < length *xs* \implies length *xs* - Suc *i* < length *xs*
by *auto*
then have *H*: *i* < length *xs* \implies *xs* ! *i* \in set *xs*
using rev-nth[of *i xs*] **unfolding** in-set-conv-nth **by** (force simp add: in-set-conv-nth)
} **note** *H* = *this*

have *l-M-A*: length (trail *a*) \leq card (atms-of-ms *A*)
by (simp add: *fin-A M-A* card-mono no-dup-length-eq-card-atm-of-lits-of-l *nd*)
have *l-M'-A*: length (trail *b*) \leq card (atms-of-ms *A*)
by (simp add: *fin-A M'-A* card-mono no-dup-length-eq-card-atm-of-lits-of-l *nd'*)
have *l-trail-weight-M*: length (trail-weight *b*) \leq 1 + card (atms-of-ms *A*)
using *l-M'-A* length-get-all-ann-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-ann-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 μ_C ?s ?b (trail-weight *a*) < μ_C ?s ?b (trail-weight *b*) **by** *simp*
moreover from μ_C -bounded[*OF bounded-M l-trail-weight-M*]
have μ_C ?s ?b (trail-weight *b*) \leq ?b \wedge ?s **by** *auto*

ultimately show $?b \wedge ?s \leq ?b \wedge ?s \wedge$
 $\mu_C ?s ?b \text{ (trail-weight } b) \leq ?b \wedge ?s \wedge$
 $\mu_C ?s ?b \text{ (trail-weight } a) < \mu_C ?s ?b \text{ (trail-weight } b)$
 by *blast*
 qed

Normal Forms

We prove that given a normal form of DPLL, with some structural invariants, then 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* rule tells us that every variable in N has a value.
2. The assumption $\neg M \models_{as} N$ implies that there is conflict.
3. There is at least one decision in the trail (otherwise, M would be a model of the set of clauses N).
4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

The assumption are saying that we have a finite upper bound A for the literals, that we cannot do any step *no-step dpll-bj* S

theorem *dpll-backjump-final-state:*

fixes $A :: 'v \text{ clause set}$ **and** $S \ T :: 'st$

assumes

atms-of-mm (*clauses*_{NOT} S) \subseteq *atms-of-ms* A **and**

atm-of ' *lits-of-l* (*trail* S) \subseteq *atms-of-ms* A **and**

no-dup (*trail* S) **and**

finite A **and**

inv: *inv* S **and**

n-s: *no-step dpll-bj* S **and**

decomp: *all-decomposition-implies-m* (*clauses*_{NOT} S) (*get-all-ann-decomposition* (*trail* S))

shows *unsatisfiable* (*set-mset* (*clauses*_{NOT} S))

\vee (*trail* $S \models_{asm}$ *clauses*_{NOT} $S \wedge$ *satisfiable* (*set-mset* (*clauses*_{NOT} S)))

proof –

let $?N = \text{set-mset} (\text{clauses}_{NOT} S)$

let $?M = \text{trail } S$

consider

(*sat*) *satisfiable* $?N$ **and** $?M \models_{as} ?N$

| (*sat'*) *satisfiable* $?N$ **and** $\neg ?M \models_{as} ?N$

| (*unsat*) *unsatisfiable* $?N$

by *auto*

then show *?thesis*

proof *cases*

case *sat'* **note** $\text{sat} = \text{this}(1)$ **and** $M = \text{this}(2)$

obtain C **where** $C \in ?N$ **and** $\neg ?M \models_a C$ **using** M **unfolding** *true-annots-def* **by** *auto*

obtain $I :: 'v \text{ literal set}$ **where**

$I \models_s ?N$ **and**

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

tot: *total-over-m* $I ?N$ **and**

atm-I-N: *atm-of* ' $I \subseteq \text{atms-of-ms } ?N$

```

using sat unfolding satisfiable-def-min by auto
let  $?I = I \cup \{P \mid P. P \in \text{ lits-of-}l \text{ } ?M \wedge \text{ atm-of } P \notin \text{ atm-of ' } I\}$ 
let  $?O = \{\text{unmark } L \mid L. \text{ is-decided } L \wedge L \in \text{ set } ?M \wedge \text{ atm-of } (\text{lit-of } L) \notin \text{ atms-of-ms } ?N\}$ 
have  $\text{cons-}I'$ : consistent-interp  $?I$ 
  using cons using  $\langle \text{no-dup } ?M \rangle$  unfolding consistent-interp-def
  by (auto simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def
    dest!: no-dup-cannot-not-lit-and-uminus)
have  $\text{tot-}I'$ : total-over-m  $?I$   $(?N \cup \text{unmark-}l \text{ } ?M)$ 
  using tot atm-I-N unfolding total-over-m-def total-over-set-def
  by (fastforce simp: image-iff lits-of-def)
have  $\{P \mid P. P \in \text{ lits-of-}l \text{ } ?M \wedge \text{ atm-of } P \notin \text{ atm-of ' } I\} \models_s ?O$ 
  using  $\langle I \models_s ?N \rangle$  atm-I-N by (auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def)
then have  $I'-N$ :  $?I \models_s ?N \cup ?O$ 
  using  $\langle I \models_s ?N \rangle$  true-clss-union-increase by force
have  $\text{tot}'$ : total-over-m  $?I$   $(?N \cup ?O)$ 
  using atm-I-N tot unfolding total-over-m-def total-over-set-def
  by (force simp: lits-of-def elim!: is-decided-ex-Decided)

have  $\text{atms-}N\text{-}M$ :  $\text{atms-of-ms } ?N \subseteq \text{ atm-of ' } \text{ lits-of-}l \text{ } ?M$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $l :: 'v$  where
     $l\text{-}N$ :  $l \in \text{ atms-of-ms } ?N$  and
     $l\text{-}M$ :  $l \notin \text{ atm-of ' } \text{ lits-of-}l \text{ } ?M$ 
  by auto
  have undefined-lit  $?M$   $(\text{Pos } l)$ 
    using  $l\text{-}M$  by (metis Decided-Propagated-in-iff-in-lits-of-l
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
  from  $\text{bj-decide}_{NOT}[\text{OF decide}_{NOT}[\text{OF this}]]$  show False
    using  $l\text{-}N$   $n\text{-}s$  by (metis literal.sel(1) state-eq_{NOT}-ref)
qed
have  $?M \models_{as} C\text{Not } C$ 
  apply (rule all-variables-defined-not-imply-cnot)
  using  $\langle C \in \text{ set-mset } (\text{clauses}_{NOT} S) \rangle \langle \neg \text{ trail } S \models_a C \rangle$ 
     $\text{atms-}N\text{-}M$  by (auto dest: atms-of-atms-of-ms-mono)
have  $\exists l \in \text{ set } ?M. \text{ is-decided } l$ 
proof (rule ccontr)
  let  $?O = \{\text{unmark } L \mid L. \text{ is-decided } L \wedge L \in \text{ set } ?M \wedge \text{ atm-of } (\text{lit-of } L) \notin \text{ atms-of-ms } ?N\}$ 
  have  $\vartheta[\text{iff}]: \bigwedge I. \text{ total-over-m } I \text{ } (?N \cup ?O \cup \text{unmark-}l \text{ } ?M)$ 
     $\longleftrightarrow \text{ total-over-m } I \text{ } (?N \cup \text{unmark-}l \text{ } ?M)$ 
  unfolding total-over-set-def total-over-m-def atms-of-ms-def by blast
  assume  $\neg ?thesis$ 
  then have  $[\text{simp}]: \{\text{unmark } L \mid L. \text{ is-decided } L \wedge L \in \text{ set } ?M\}$ 
     $= \{\text{unmark } L \mid L. \text{ is-decided } L \wedge L \in \text{ set } ?M \wedge \text{ atm-of } (\text{lit-of } L) \notin \text{ atms-of-ms } ?N\}$ 
  by auto
  then have  $?N \cup ?O \models_{ps} \text{unmark-}l \text{ } ?M$ 
    using all-decomposition-implies-propagated-lits-are-implied  $[\text{OF decomp}]$  by auto

  then have  $?I \models_s \text{unmark-}l \text{ } ?M$ 
    using  $\text{cons-}I' \text{ } I'\text{-}N \text{ tot-}I' \langle I \models_s ?N \cup ?O \rangle$  unfolding  $\vartheta$  true-clss-clss-def by blast
  then have  $\text{ lits-of-}l \text{ } ?M \subseteq ?I$ 
    unfolding true-clss-def lits-of-def by auto
  then have  $?M \models_{as} ?N$ 
    using  $I'\text{-}N \langle C \in ?N \rangle \langle \neg ?M \models_a C \rangle \text{cons-}I' \text{atms-}N\text{-}M$ 
    by (meson  $\langle \text{trail } S \models_{as} C\text{Not } C \rangle$  consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
      true-annots-def true-clss-mono-set-mset-l true-clss-def)

```

```

    then show False using M by fast
qed
from List.split-list-first-propE[OF this] obtain K :: 'v literal and
  F F' :: ('v, unit) ann-lits where
  M-K: ?M = F' @ Decided K # F and
  nm:  $\forall f \in \text{set } F'. \neg \text{is-decided } f$ 
  unfolding is-decided-def by metis
let ?K = Decided K :: ('v, unit) ann-lit
have ?K ∈ set ?M
  unfolding M-K by auto
let ?C = image-mset lit-of {#L ∈ #mset ?M. is-decided L ∧ L ≠ ?K #} :: 'v clause
let ?C' = set-mset (image-mset ( $\lambda L. 'v \text{ literal. } \{ \#L \# \}$ ) (?C + unmark ?K))
have ?N ∪ {unmark L | L. is-decided L ∧ L ∈ set ?M} ⊨ps unmark-l ?M
  using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have C': ?C' = {unmark L | L. is-decided L ∧ L ∈ set ?M}
  unfolding M-K by standard force+
ultimately have N-C-M: ?N ∪ ?C' ⊨ps unmark-l ?M
  by auto
have N-M-False: ?N ∪ ( $\lambda L. \text{unmark } L$ ) ' (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-l ?M = ?N ∪ unmark-l ?M
      unfolding M-K by auto
    show ?thesis
      using true-clss-clss-left-right[OF N-C-M, of {{#}}] N-M-False unfolding A by auto
  qed
have ?N ⊨p image-mset uminus ?C + {#-K #}
  unfolding true-clss-clss-def true-clss-clss-def total-over-m-def
proof (intro allI impI)
  fix I
  assume
    tot: total-over-set I (atms-of-ms (?N ∪ {image-mset uminus ?C + {#-K #}))) and
    cons: consistent-interp I and
    I ⊨s ?N
  have (K ∈ I ∧ -K ∉ I) ∨ (-K ∈ I ∧ K ∉ I)
    using cons tot unfolding consistent-interp-def by (cases K) auto
  have {a ∈ set (trail S). is-decided a ∧ a ≠ Decided K} =
    set (trail S) ∩ {L. is-decided L ∧ L ≠ Decided K}
  by auto
  then have tot': total-over-set I
    (atm-of ' lit-of ' (set ?M ∩ {L. is-decided L ∧ L ≠ Decided K}))
    using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
  { fix x :: ('v, unit) ann-lit
    assume
      a3: lit-of x ∉ I and
      a1: x ∈ set ?M and
      a4: is-decided x and
      a5: x ≠ Decided K
    then have Pos (atm-of (lit-of x)) ∈ I ∨ Neg (atm-of (lit-of x)) ∈ I
      using a5 a4 tot' a1 unfolding total-over-set-def atms-of-s-def by blast

```

```

moreover have f6: Neg (atm-of (lit-of x)) = - Pos (atm-of (lit-of x))
  by simp
ultimately have - lit-of x ∈ I
  using f6 a3 by (metis (no-types) atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
    literal.sel(1))
} note H = this

have  $\neg I \models_s ?C'$ 
  using  $\langle ?N \cup ?C' \models_{ps} \{\{\#\}\} \rangle \text{ tot cons } \langle I \models_s ?N \rangle$ 
  unfolding true-clss-clss-def total-over-m-def
  by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of)
then show  $I \models \text{image-mset uminus } ?C + \{\# - K\# \}$ 
  unfolding true-clss-def true-cl-def using  $\langle (K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I) \rangle$ 
  by (auto dest!: H)
qed
moreover have  $F \models_{as} CNot (\text{image-mset uminus } ?C)$ 
  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-decided L \wedge L \neq Decided K\#})]
     $\langle C \in ?N \rangle \text{ n-s } \langle ?M \models_{as} CNot C \rangle$  bj-backjump inv (no-dup (trail S)) unfolding M-K by auto
  then show ?thesis by fast
qed auto
qed

end — End of dpll-with-backjumping-ops

locale dpll-with-backjumping =
  dpll-with-backjumping-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT inv
  backjump-conds propagate-conds
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  inv :: 'st  $\Rightarrow$  bool and
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool
  +
  assumes dpll-bj-inv:  $\bigwedge S T. \text{dpll-bj } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$ 
begin

lemma rtrancpl-dpll-bj-inv:
  assumes dpll-bj** S T and inv S
  shows inv T
  using assms by (induction rule: rtrancpl-induct)
  (auto simp add: dpll-bj-no-dup intro: dpll-bj-inv)

lemma rtrancpl-dpll-bj-no-dup:
  assumes dpll-bj** S T and inv S
  and no-dup (trail S)
  shows no-dup (trail T)
  using assms by (induction rule: rtrancpl-induct)
  (auto simp add: dpll-bj-no-dup dest: rtrancpl-dpll-bj-inv dpll-bj-inv)

```

lemma *rtranclp-dpll-bj-atms-of-ms-clauses-inv*:

assumes

*dpll-bj** S T and inv S*

shows *atms-of-mm (clauses_{NOT} S) = atms-of-mm (clauses_{NOT} T)*

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

(auto dest: rtranclp-dpll-bj-inv dpll-bj-atms-of-ms-clauses-inv)

lemma *rtranclp-dpll-bj-atms-in-trail*:

assumes

*dpll-bj** S T and*

inv S and

atm-of ' (lits-of-l (trail S)) ⊆ atms-of-mm (clauses_{NOT} S)

shows *atm-of ' (lits-of-l (trail T)) ⊆ atms-of-mm (clauses_{NOT} T)*

using *assms apply (induction rule: rtranclp-induct)*

using *dpll-bj-atms-in-trail dpll-bj-atms-of-ms-clauses-inv rtranclp-dpll-bj-inv by auto*

lemma *rtranclp-dpll-bj-sat-iff*:

assumes *dpll-bj** S T and inv S*

shows *I ⊨_{sm} clauses_{NOT} S ⟷ I ⊨_{sm} clauses_{NOT} T*

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

(auto dest!: dpll-bj-sat-iff simp: rtranclp-dpll-bj-inv)

lemma *rtranclp-dpll-bj-atms-in-trail-in-set*:

assumes

*dpll-bj** S T and*

inv S

atms-of-mm (clauses_{NOT} S) ⊆ A and

atm-of ' (lits-of-l (trail S)) ⊆ A

shows *atm-of ' (lits-of-l (trail T)) ⊆ A*

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

(auto dest: rtranclp-dpll-bj-inv

simp: dpll-bj-atms-in-trail-in-set rtranclp-dpll-bj-atms-of-ms-clauses-inv rtranclp-dpll-bj-inv)

lemma *rtranclp-dpll-bj-all-decomposition-implies-inv*:

assumes

*dpll-bj** S T and*

inv S

all-decomposition-implies-m (clauses_{NOT} S) (get-all-ann-decomposition (trail S))

shows *all-decomposition-implies-m (clauses_{NOT} T) (get-all-ann-decomposition (trail T))*

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

(auto intro: dpll-bj-all-decomposition-implies-inv simp: rtranclp-dpll-bj-inv)

lemma *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*:

{(T, S). dpll-bj⁺⁺ S T

∧ atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A ∧ atm-of ' lits-of-l (trail S) ⊆ atms-of-ms A

∧ no-dup (trail S) ∧ inv S}

⊆ {(T, S). dpll-bj S T ∧ atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A

∧ atm-of ' lits-of-l (trail S) ⊆ atms-of-ms A ∧ no-dup (trail S) ∧ inv S}⁺

(is ?A ⊆ ?B⁺)

proof *standard*

fix *x*

assume *x-A: x ∈ ?A*

obtain *S T::'st where*

x[simp]: x = (T, S) by (cases x) auto

have

$dpll\text{-}bj^{++} \ S \ T$ **and**
 $atms\text{-}of\text{-}mm \ (clauses_{NOT} \ S) \subseteq atms\text{-}of\text{-}ms \ A$ **and**
 $atm\text{-}of \ ' \ lits\text{-}of\text{-}l \ (trail \ S) \subseteq atms\text{-}of\text{-}ms \ A$ **and**
 $no\text{-}dup \ (trail \ S)$ **and**
 $inv \ S$
using $x\text{-}A$ **by** *auto*
then show $x \in ?B^+$ **unfolding** x
proof (*induction rule: tranclp-induct*)
case *base*
then show $?case$ **by** *auto*
next
case ($step \ T \ U$) **note** $step = this(1)$ **and** $ST = this(2)$ **and** $IH = this(3)[OF \ this(4-7)]$
and $N\text{-}A = this(4)$ **and** $M\text{-}A = this(5)$ **and** $nd = this(6)$ **and** $inv = this(7)$

have [*simp*]: $atms\text{-}of\text{-}mm \ (clauses_{NOT} \ S) = atms\text{-}of\text{-}mm \ (clauses_{NOT} \ T)$
using $step \ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv \ tranclp\text{-}into\text{-}rtranclp \ inv$ **by** *fastforce*
have $no\text{-}dup \ (trail \ T)$
using $local.step \ nd \ rtranclp\text{-}dpll\text{-}bj\text{-}no\text{-}dup \ tranclp\text{-}into\text{-}rtranclp \ inv$ **by** *fastforce*
moreover have $atm\text{-}of \ ' \ (lits\text{-}of\text{-}l \ (trail \ T)) \subseteq atms\text{-}of\text{-}ms \ A$
by ($metis \ inv \ M\text{-}A \ N\text{-}A \ local.step \ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail\text{-}in\text{-}set$
 $tranclp\text{-}into\text{-}rtranclp$)
moreover have $inv \ T$
using $inv \ local.step \ rtranclp\text{-}dpll\text{-}bj\text{-}inv \ tranclp\text{-}into\text{-}rtranclp$ **by** *fastforce*
ultimately have $(U, \ T) \in ?B$ **using** $ST \ N\text{-}A \ M\text{-}A \ inv$ **by** *auto*
then show $?case$ **using** IH **by** (*rule tranclp-into-trancl2*)
qed
qed

lemma *wf-tranclp-dpll-bj*:
assumes $fin: \text{finite } A$
shows $wf \ \{(T, \ S). \ dpll\text{-}bj^{++} \ S \ T$
 $\wedge atms\text{-}of\text{-}mm \ (clauses_{NOT} \ S) \subseteq atms\text{-}of\text{-}ms \ A \wedge atm\text{-}of \ ' \ lits\text{-}of\text{-}l \ (trail \ S) \subseteq atms\text{-}of\text{-}ms \ A$
 $\wedge no\text{-}dup \ (trail \ S) \wedge inv \ S\}$
using $wf\text{-}tranclp[OF \ wf\text{-}dpll\text{-}bj[OF \ fin]] \ rtranclp\text{-}dpll\text{-}bj\text{-}inv\text{-}incl\text{-}dpll\text{-}bj\text{-}inv\text{-}trancl$
by (*rule wf-subset*)

lemma *dpll-bj-sat-ext-iff*:
 $dpll\text{-}bj \ S \ T \implies inv \ S \implies I \models_{sextm \ clauses_{NOT}} S \longleftrightarrow I \models_{sextm \ clauses_{NOT}} T$
by (*simp add: dpll-bj-clauses*)

lemma *rtranclp-dpll-bj-sat-ext-iff*:
 $dpll\text{-}bj^{**} \ S \ T \implies inv \ S \implies I \models_{sextm \ clauses_{NOT}} S \longleftrightarrow I \models_{sextm \ clauses_{NOT}} T$
by (*induction rule: rtranclp-induct*) (*simp-all add: rtranclp-dpll-bj-inv dpll-bj-sat-ext-iff*)

theorem *full-dpll-backjump-final-state*:
fixes $A :: 'v \text{ clause set}$ **and** $S \ T :: 'st$
assumes
 $full: \text{full } dpll\text{-}bj \ S \ T$ **and**
 $atms\text{-}S: atms\text{-}of\text{-}mm \ (clauses_{NOT} \ S) \subseteq atms\text{-}of\text{-}ms \ A$ **and**
 $atms\text{-}trail: atm\text{-}of \ ' \ lits\text{-}of\text{-}l \ (trail \ S) \subseteq atms\text{-}of\text{-}ms \ A$ **and**
 $n\text{-}d: no\text{-}dup \ (trail \ S)$ **and**
 $finite \ A$ **and**
 $inv: inv \ S$ **and**
 $decomp: all\text{-}decomposition\text{-}implies\text{-}m \ (clauses_{NOT} \ S) \ (get\text{-}all\text{-}ann\text{-}decomposition \ (trail \ S))$
shows $unsatisfiable \ (set\text{-}mset \ (clauses_{NOT} \ S))$
 $\vee (trail \ T \models_{asm \ clauses_{NOT}} S \wedge satisfiable \ (set\text{-}mset \ (clauses_{NOT} \ S)))$

proof –

have st : $dpll\text{-}bj^{**} \ S \ T$ **and** $no\text{-}step \ dpll\text{-}bj \ T$
using $full \ unfolding \ full\text{-}def$ **by** $fast+$
moreover **have** $atms\text{-}of\text{-}mm \ (clauses_{NOT} \ T) \subseteq atms\text{-}of\text{-}ms \ A$
using $atms\text{-}S \ inv \ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv \ st$ **by** $blast$
moreover **have** $atm\text{-}of \ ' \ lits\text{-}of\text{-}l \ (trail \ T) \subseteq atms\text{-}of\text{-}ms \ A$
using $atms\text{-}S \ atms\text{-}trail \ inv \ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail\text{-}in\text{-}set \ st$ **by** $auto$
moreover **have** $no\text{-}dup \ (trail \ T)$
using $n\text{-}d \ inv \ rtranclp\text{-}dpll\text{-}bj\text{-}no\text{-}dup \ st$ **by** $blast$
moreover **have** inv : $inv \ T$
using $inv \ rtranclp\text{-}dpll\text{-}bj\text{-}inv \ st$ **by** $blast$
moreover
have $decomp$: $all\text{-}decomposition\text{-}implies\text{-}m \ (clauses_{NOT} \ T) \ (get\text{-}all\text{-}ann\text{-}decomposition \ (trail \ T))$
using $\langle inv \ S \rangle \ decomp \ rtranclp\text{-}dpll\text{-}bj\text{-}all\text{-}decomposition\text{-}implies\text{-}inv \ st$ **by** $blast$
ultimately **have** $unsatisfiable \ (set\text{-}mset \ (clauses_{NOT} \ T))$
 $\vee \ (trail \ T \models_{asm} clauses_{NOT} \ T \wedge satisfiable \ (set\text{-}mset \ (clauses_{NOT} \ T)))$
using $\langle finite \ A \rangle \ dpll\text{-}backjump\text{-}final\text{-}state$ **by** $force$
then show $?thesis$
by $(meson \ \langle inv \ S \rangle \ rtranclp\text{-}dpll\text{-}bj\text{-}sat\text{-}iff \ satisfiable\text{-}carac \ st \ true\text{-}annots\text{-}true\text{-}cls)$
qed

corollary $full\text{-}dpll\text{-}backjump\text{-}final\text{-}state\text{-}from\text{-}init\text{-}state$:

fixes $A :: 'v \ clause \ set$ **and** $S \ T :: 'st$
assumes
 $full$: $full \ dpll\text{-}bj \ S \ T$ **and**
 $trail \ S = []$ **and**
 $clauses_{NOT} \ S = N$ **and**
 $inv \ S$
shows $unsatisfiable \ (set\text{-}mset \ N) \vee (trail \ T \models_{asm} N \wedge satisfiable \ (set\text{-}mset \ N))$
using $assms \ full\text{-}dpll\text{-}backjump\text{-}final\text{-}state[of \ S \ T \ set\text{-}mset \ N]$ **by** $auto$

lemma $tranclp\text{-}dpll\text{-}bj\text{-}trail\text{-}mes\text{-}decreasing\text{-}prop$:

assumes $dpll$: $dpll\text{-}bj^{++} \ S \ T$ **and** inv : $inv \ S$ **and**
 $N\text{-}A$: $atms\text{-}of\text{-}mm \ (clauses_{NOT} \ S) \subseteq atms\text{-}of\text{-}ms \ A$ **and**
 $M\text{-}A$: $atm\text{-}of \ ' \ lits\text{-}of\text{-}l \ (trail \ S) \subseteq atms\text{-}of\text{-}ms \ A$ **and**
 $n\text{-}d$: $no\text{-}dup \ (trail \ S)$ **and**
 $fin\text{-}A$: $finite \ A$
shows $(2 + card \ (atms\text{-}of\text{-}ms \ A)) \wedge (1 + card \ (atms\text{-}of\text{-}ms \ A))$
 $\quad - \mu_C \ (1 + card \ (atms\text{-}of\text{-}ms \ A)) \ (2 + card \ (atms\text{-}of\text{-}ms \ A)) \ (trail\text{-}weight \ T)$
 $\quad < (2 + card \ (atms\text{-}of\text{-}ms \ A)) \wedge (1 + card \ (atms\text{-}of\text{-}ms \ A))$
 $\quad - \mu_C \ (1 + card \ (atms\text{-}of\text{-}ms \ A)) \ (2 + card \ (atms\text{-}of\text{-}ms \ A)) \ (trail\text{-}weight \ S)$
using $dpll$

proof ($induction$)

case $base$

then show $?case$

using $N\text{-}A \ M\text{-}A \ n\text{-}d \ dpll\text{-}bj\text{-}trail\text{-}mes\text{-}decreasing\text{-}prop \ fin\text{-}A \ inv$ **by** $blast$

next

case $(step \ T \ U)$ **note** $st = this(1)$ **and** $dpll = this(2)$ **and** $IH = this(3)$

have $atms\text{-}of\text{-}mm \ (clauses_{NOT} \ S) = atms\text{-}of\text{-}mm \ (clauses_{NOT} \ T)$

using $rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}of\text{-}ms\text{-}clauses\text{-}inv$ **by** $(metis \ dpll\text{-}bj\text{-}clauses \ dpll\text{-}bj\text{-}inv \ inv \ st \ tranclpD)$

then have $N\text{-}A'$: $atms\text{-}of\text{-}mm \ (clauses_{NOT} \ T) \subseteq atms\text{-}of\text{-}ms \ A$

using $N\text{-}A$ **by** $auto$

moreover **have** $M\text{-}A'$: $atm\text{-}of \ ' \ lits\text{-}of\text{-}l \ (trail \ T) \subseteq atms\text{-}of\text{-}ms \ A$

by $(meson \ M\text{-}A \ N\text{-}A \ inv \ rtranclp\text{-}dpll\text{-}bj\text{-}atms\text{-}in\text{-}trail\text{-}in\text{-}set \ st \ dpll \ tranclp.r\text{-}into\text{-}trancl \ tranclp\text{-}into\text{-}rtranclp \ tranclp\text{-}trans)$

```

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 — End of *dpll-with-backjumping*

5.2.4 CDCL

In this section we will now define the conflict driven clause learning above DPLL: we first introduce the rules learn and forget, and the add these rules to the DPLL calculus.

Learn and Forget

Learning adds a new clause where all the literals are already included in the clauses.

```

locale learn-ops =
  dpll-state trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  learn-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive learn :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
learnNOT-rule: clausesNOT S  $\models_{pm}$  C  $\Rightarrow$ 
atms-of C  $\subseteq$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))  $\Rightarrow$ 
learn-cond C S  $\Rightarrow$ 
T  $\sim$  add-clNOT C S  $\Rightarrow$ 
learn S T
inductive-cases learnNOTE: learn S T

```

```

lemma learn- $\mu_C$ -stable:
  assumes learn S T and no-dup (trail S)
  shows  $\mu_C A B (trail\text{-}weight\ S) = \mu_C A B (trail\text{-}weight\ T)$ 
  using assms by (auto elim: learnNOTE)
end

```

Forget removes an information that can be deduced from the context (e.g. redundant clauses, tautologies)

```

locale forget-ops =
  dpll-state trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and

```



```

    remove-clsNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
    forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive forgetNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
forgetNOT:
    removeAll-mset C(clausesNOT S)  $\models_{pm}$  C  $\Rightarrow$ 
    forget-cond C S  $\Rightarrow$ 
    C  $\in \#$  clausesNOT S  $\Rightarrow$ 
    T  $\sim$  remove-clsNOT C S  $\Rightarrow$ 
    forgetNOT S T
inductive-cases forgetNOTE: forgetNOT S T

lemma forget- $\mu_C$ -stable:
    assumes forgetNOT S T
    shows  $\mu_C$  A B (trail-weight S) =  $\mu_C$  A B (trail-weight T)
    using assms by (auto elim!: forgetNOTE)
end

locale learn-and-forgetNOT =
    learn-ops trail clausesNOT prepend-trail tl-trail add-clsNOT remove-clsNOT learn-cond +
    forget-ops trail clausesNOT prepend-trail tl-trail add-clsNOT remove-clsNOT forget-cond
for
    trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
    clausesNOT :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit) ann-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 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

```

Definition of CDCL

```

locale conflict-driven-clause-learning-ops =
    dpll-with-backjumping-ops trail clausesNOT prepend-trail tl-trail add-clsNOT remove-clsNOT
    inv backjump-conds propagate-conds +
    learn-and-forgetNOT trail clausesNOT prepend-trail tl-trail add-clsNOT remove-clsNOT learn-cond
    forget-cond
for
    trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
    clausesNOT :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit) ann-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 and
    inv :: 'st  $\Rightarrow$  bool and
    backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
    propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
    learn-cond forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

```

inductive $cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**
c-dpll-bj: $dpll\text{-}bj\ S\ S' \Longrightarrow cdcl_{NOT}\ S\ S' \mid$
c-learn: $learn\ S\ S' \Longrightarrow cdcl_{NOT}\ S\ S' \mid$
c-forget_{NOT}: $forget_{NOT}\ S\ S' \Longrightarrow cdcl_{NOT}\ S\ S'$

lemma $cdcl_{NOT}\text{-all-induct}[consumes\ 1, case\text{-}names\ dpll\text{-}bj\ learn\ forget_{NOT}]$:

fixes $S\ T :: 'st$

assumes $cdcl_{NOT}\ S\ T$ **and**

$dpll: \bigwedge T. dpll\text{-}bj\ S\ T \Longrightarrow P\ S\ T$ **and**

learning:

$\bigwedge C\ T. clauses_{NOT}\ S \models_{pm} C \Longrightarrow$

$atms\text{-}of\ C \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S)) \Longrightarrow$

$T \sim add\text{-}cls_{NOT}\ C\ S \Longrightarrow$

$P\ S\ T$ **and**

forgetting: $\bigwedge C\ T. removeAll\text{-}mset\ C\ (clauses_{NOT}\ S) \models_{pm} C \Longrightarrow$

$C \in \# clauses_{NOT}\ S \Longrightarrow$

$T \sim remove\text{-}cls_{NOT}\ C\ S \Longrightarrow$

$P\ S\ T$

shows $P\ S\ T$

using $assms(1)$ **by** (*induction rule*: $cdcl_{NOT}.induct$)

(*auto intro*: $assms(2, 3, 4)$ *elim!*: $learn_{NOT}E\ forget_{NOT}E$)**+**

lemma $cdcl_{NOT}\text{-no-dup}$:

assumes

$cdcl_{NOT}\ S\ T$ **and**

$inv\ S$ **and**

$no\text{-}dup\ (trail\ S)$

shows $no\text{-}dup\ (trail\ T)$

using $assms$ **by** (*induction rule*: $cdcl_{NOT}\text{-all-induct}$) (*auto intro*: $dpll\text{-}bj\text{-no-dup}$)

Consistency of the trail lemma $cdcl_{NOT}\text{-consistent}$:

assumes

$cdcl_{NOT}\ S\ T$ **and**

$inv\ S$ **and**

$no\text{-}dup\ (trail\ S)$

shows $consistent\text{-interp}\ (lits\text{-}of\text{-}l\ (trail\ T))$

using $cdcl_{NOT}\text{-no-dup}[OF\ assms]\ distinct\text{-consistent}\text{-interp}$ **by** *fast*

The subtle problem here is that tautologies can be removed, meaning that some variable can disappear of the problem. It is also means that some variable of the trail might not be present in the clauses anymore.

lemma $cdcl_{NOT}\text{-atms-of-ms-clauses-decreasing}$:

assumes $cdcl_{NOT}\ S\ T$ **and** $inv\ S$ **and** $no\text{-}dup\ (trail\ S)$

shows $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ T) \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S))$

using $assms$ **by** (*induction rule*: $cdcl_{NOT}\text{-all-induct}$)

(*auto dest!*: $dpll\text{-}bj\text{-atms-of-ms-clauses-inv}\ set\text{-}mp\ simp\ add: atms\text{-}of\text{-}ms\text{-}def\ Union\text{-}eq$)

lemma $cdcl_{NOT}\text{-atms-in-trail}$:

assumes $cdcl_{NOT}\ S\ T$ **and** $inv\ S$ **and** $no\text{-}dup\ (trail\ S)$

and $atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$

shows $atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ T)) \subseteq atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$

using $assms$ **by** (*induction rule*: $cdcl_{NOT}\text{-all-induct}$) (*auto simp add*: $dpll\text{-}bj\text{-atms-in-trail}$)

lemma $cdcl_{NOT}\text{-atms-in-trail-in-set}$:

```

assumes
   $cdcl_{NOT} S T$  and  $inv S$  and  $no\_dup (trail S)$  and
   $atms\_of\_mm (clauses_{NOT} S) \subseteq A$  and
   $atm\_of \text{ ' } (lits\_of\_l (trail S)) \subseteq A$ 
shows  $atm\_of \text{ ' } (lits\_of\_l (trail T)) \subseteq A$ 
using assms
by (induction rule: cdclNOT-all-induct)
  (simp-all add: dpll-bj-atms-in-trail-in-set dpll-bj-atms-of-ms-clauses-inv)

lemma cdclNOT-all-decomposition-implies:
assumes  $cdcl_{NOT} S T$  and  $inv S$  and  $n\_d[simp]: no\_dup (trail S)$  and
   $all\_decomposition\_implies\_m (clauses_{NOT} S) (get\_all\_ann\_decomposition (trail S))$ 
shows
   $all\_decomposition\_implies\_m (clauses_{NOT} T) (get\_all\_ann\_decomposition (trail T))$ 
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 ( $forget_{NOT} C T$ ) note  $cls\_C = this(1)$  and  $C = this(2)$  and  $T = this(3)$  and  $inv = this(4)$ 
and
   $decomp = this(5)$ 
show ?case
  unfolding all-decomposition-implies-def Ball-def
proof (intro allI, clarify)
  fix  $a b$ 
  assume  $(a, b) \in set (get\_all\_ann\_decomposition (trail T))$ 
  then have  $unmark\_l a \cup set\_mset (clauses_{NOT} S) \models_{ps} unmark\_l b$ 
    using  $decomp T$  by (auto simp add: all-decomposition-implies-def)
  moreover
    have  $a1:C \in set\_mset (clauses_{NOT} S)$ 
      using  $C$  by blast
    have  $clauses_{NOT} T = clauses_{NOT} (remove\_cls_{NOT} C S)$ 
      using  $T state\_eq_{NOT}\text{-}clauses$  by blast
    then have  $set\_mset (clauses_{NOT} T) \models_{ps} set\_mset (clauses_{NOT} S)$ 
      using  $a1$  by (metis (no-types) clauses-remove-clsNOT cls-C insert-Diff order-refl
        set-mset-minus-replicate-mset(1) true-clss-clss-def true-clss-clss-insert)
    ultimately show  $unmark\_l a \cup set\_mset (clauses_{NOT} T) \models_{ps} unmark\_l b$ 
      using true-clss-clss-generalise-true-clss-clss by blast
  qed
qed

```

Extension of models **lemma** *cdcl_{NOT}-bj-sat-ext-iff:*

```

assumes  $cdcl_{NOT} S T$  and  $inv S$  and  $n\_d: no\_dup (trail S)$ 
shows  $I \models_{sextm} clauses_{NOT} S \longleftrightarrow I \models_{sextm} clauses_{NOT} T$ 
using assms
proof (induction rule: cdclNOT-all-induct)
  case dpll-bj
  then show ?case by (simp add: dpll-bj-clauses)
next
  case ( $learn C T$ ) note  $T = this(3)$ 

```

```

{ fix J
  assume
     $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} S$  and
     $I \subseteq J$  and
    tot: total-over-m  $J$  (set-mset ( $\{\#C\# \} + \text{clauses}_{\text{NOT}} S$ )) and
    cons: consistent-interp  $J$ 
  then have  $J \models_{\text{sm}} \text{clauses}_{\text{NOT}} S$  unfolding true-clss-ext-def by auto

  moreover
    with  $\langle \text{clauses}_{\text{NOT}} S \models_{\text{pm}} C \rangle$  have  $J \models C$ 
    using tot cons unfolding true-clss-clss-def by auto
    ultimately have  $J \models_{\text{sm}} \{\#C\# \} + \text{clauses}_{\text{NOT}} S$  by auto
}
then have  $H: I \models_{\text{sextm}} (\text{clauses}_{\text{NOT}} S) \implies I \models_{\text{sext}} \text{insert } C (\text{set-mset } (\text{clauses}_{\text{NOT}} S))$ 
unfolding true-clss-ext-def by auto
show ?case
apply standard
  using  $T$  n-d apply (auto simp add: H)[]
using  $T$  n-d apply simp
by (metis Diff-insert-absorb insert-subset subsetI subset-antisym
  true-clss-ext-decrease-right-remove-r)
next
case (forgetNOT  $C$   $T$ ) note cls-C = this(1) and  $T = \text{this}(3)$ 
{ fix J
  assume
     $I \models_{\text{sext}} \text{set-mset } (\text{clauses}_{\text{NOT}} S) - \{C\}$  and
     $I \subseteq J$  and
    tot: total-over-m  $J$  (set-mset ( $\text{clauses}_{\text{NOT}} S$ )) and
    cons: consistent-interp  $J$ 
  then have  $J \models_{\text{s}} \text{set-mset } (\text{clauses}_{\text{NOT}} S) - \{C\}$ 
    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-clss-def
    by (metis Un-commute forgetNOT.hyps(2) insert-Diff insert-is-Un order-refl
      set-mset-minus-replicate-mset(1))
    ultimately have  $J \models_{\text{sm}} (\text{clauses}_{\text{NOT}} S)$  by (metis insert-Diff-single true-clss-insert)
  }
  then have  $H: I \models_{\text{sext}} \text{set-mset } (\text{clauses}_{\text{NOT}} S) - \{C\} \implies I \models_{\text{sextm}} (\text{clauses}_{\text{NOT}} S)$ 
    unfolding true-clss-ext-def by blast
  show ?case using  $T$  by (auto simp: true-clss-ext-decrease-right-remove-r H)
}
qed

end — end of conflict-driven-clause-learning-ops

```

CDCL with invariant

```

locale conflict-driven-clause-learning =
  conflict-driven-clause-learning-ops +
  assumes cdclNOT-inv:  $\bigwedge S T. \text{cdcl}_{\text{NOT}} S T \implies \text{inv } S \implies \text{inv } T$ 
begin
sublocale dpll-with-backjumping
apply unfold-locales
using cdclNOT.simps cdclNOT-inv by auto

```

lemma *rtrancpl-cdcl_{NOT}-inv*:

*cdcl_{NOT}** S T \implies inv S \implies inv T*

by (induction rule: rtrancpl-induct) (auto simp add: cdcl_{NOT}-inv)

lemma *rtrancpl-cdcl_{NOT}-no-dup*:

*assumes cdcl_{NOT}** S T and inv S*

and no-dup (trail S)

shows no-dup (trail T)

using assms by (induction rule: rtrancpl-induct) (auto intro: cdcl_{NOT}-no-dup rtrancpl-cdcl_{NOT}-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-mm (clauses_{NOT} S) \subseteq A and

atms-trail-S: atm-of (lits-of-l (trail S)) \subseteq A

shows atm-of (lits-of-l (trail T)) \subseteq A \wedge atms-of-mm (clauses_{NOT} T) \subseteq A

using cdcl

proof *(induction rule: 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-mm (clauses_{NOT} U) \subseteq A

using cdcl_{NOT}-atms-of-ms-clauses-decreasing[OF cdcl_{NOT}] IH n-d (inv T) by fast

moreover

have atm-of (lits-of-l (trail U)) \subseteq A

using cdcl_{NOT}-atms-in-trail-in-set[OF cdcl_{NOT}, of A] (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_{NOT} S) (get-all-ann-decomposition (trail S))

shows

all-decomposition-implies-m (clauses_{NOT} T) (get-all-ann-decomposition (trail T))

using assms by (induction)

(auto intro: rtrancpl-cdcl_{NOT}-inv cdcl_{NOT}-all-decomposition-implies rtrancpl-cdcl_{NOT}-no-dup)

lemma *rtrancpl-cdcl_{NOT}-bj-sat-ext-iff*:

*assumes cdcl_{NOT}** S T and inv S and no-dup (trail S)*

shows $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} S \iff I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T$

using assms apply (induction rule: rtrancpl-induct)

using cdcl_{NOT}-bj-sat-ext-iff by (auto intro: rtrancpl-cdcl_{NOT}-inv rtrancpl-cdcl_{NOT}-no-dup)

definition *cdcl_{NOT}-NOT-all-inv where*

cdcl_{NOT}-NOT-all-inv A S \iff (finite A \wedge inv S \wedge atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A
 \wedge atm-of (lits-of-l (trail S)) \subseteq atms-of-ms A \wedge no-dup (trail S))

lemma *cdcl_{NOT}-NOT-all-inv*:

*assumes cdcl_{NOT}** S T and cdcl_{NOT}-NOT-all-inv A S*

shows $cdcl_{NOT-NOT-all-inv} A T$
using *assms* **unfolding** $cdcl_{NOT-NOT-all-inv-def}$
by (*simp add: rtrancpl-cdcl_{NOT-inv} rtrancpl-cdcl_{NOT-no-dup} rtrancpl-cdcl_{NOT-trail-clauses-bound}*)

abbreviation *learn-or-forget* **where**
 $learn-or-forget S T \equiv learn S T \vee forget_{NOT} S T$

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}]* **by** (*blast intro: cdcl_{NOT.c-learn} cdcl_{NOT.c-forget_{NOT}}*)

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 $(2 + card (atms-of-ms A)) \wedge (1 + card (atms-of-ms A))$
 $- \mu_C (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight U)$
 $< (2 + card (atms-of-ms A)) \wedge (1 + card (atms-of-ms A))$
 $- \mu_C (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight S)$
(is ? μ U < ? μ 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** *presburger*

qed

lemma *infinite-cdcl_{NOT}-exists-learn-and-forget-infinite-chain*:

assumes

$\bigwedge i. cdcl_{NOT} (f i) (f (Suc i))$ **and**

inv: cdcl_{NOT-NOT-all-inv} A (f 0)

shows $\exists j. \forall i \geq j. learn-or-forget (f i) (f (Suc i))$

using *assms*

proof (*induction (2 + card (atms-of-ms A)) \wedge (1 + card (atms-of-ms A))*)

$- \mu_C (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight (f 0))$

arbitrary: f

rule: nat-less-induct-case)

case (*Suc n*) **note** $IH = this(1)$ **and** $\mu = this(2)$ **and** $cdcl_{NOT} = this(3)$ **and** $inv = this(4)$

consider

(*dpll-end*) $\exists j. \forall i \geq j. learn-or-forget (f i) (f (Suc i))$

```

| (dpll-more)  $\neg(\exists j. \forall i \geq j. \text{learn-or-forget } (f \ i) \ (f \ (\text{Suc } i)))$ 
by blast
then show ?case
proof cases
  case dpll-end
  then show ?thesis by auto
next
case dpll-more
then have j:  $\exists i. \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ 
  by blast
obtain i where
 $\neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$  and
 $\forall k < i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
proof -
  obtain i0 where  $\neg \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f \ i_0) \ (f \ (\text{Suc } i_0))$ 
    using j by auto
  then have {i.  $i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ }  $\neq \{\}$ 
    by auto
  let ?I = {i.  $i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ }
  let ?i = Min ?I
  have finite ?I
    by auto
  have  $\neg \text{learn } (f \ ?i) \ (f \ (\text{Suc } ?i)) \wedge \neg \text{forget}_{NOT} (f \ ?i) \ (f \ (\text{Suc } ?i))$ 
    using Min-in[OF <finite ?I> <?I  $\neq \{\}$ >] by auto
  moreover have  $\forall k < ?i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
    using Min.coboundedI[of {i.  $i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$ }, simplified]
    by (meson  $\neg \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{NOT} (f \ i_0) \ (f \ (\text{Suc } i_0))$ ) less-imp-le
    dual-order.trans not-le
  ultimately show ?thesis using that by blast
qed
def g  $\equiv \lambda n. f \ (n + \text{Suc } i)$ 
have dpll-bj (f i) (g 0)
  using  $\neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{NOT} (f \ i) \ (f \ (\text{Suc } i))$  cdclNOT cdclNOT.cases
  g-def by auto
{
  fix j
  assume j  $\leq i$ 
  then have learn-or-forget** (f 0) (f j)
    apply (induction j)
    apply simp
    by (metis (no-types, lifting) Suc-leD Suc-le-lessD rtranclp.simps
       $\langle \forall k < i. \text{learn } (f \ k) \ (f \ (\text{Suc } k)) \vee \text{forget}_{NOT} (f \ k) \ (f \ (\text{Suc } k)) \rangle$ )
}
then have learn-or-forget** (f 0) (f i) by blast
then have  $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (g \ 0))$ 
 $< (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$ 
   $- \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } (f \ 0))$ 
  using learn-or-forget-dpll- $\mu_C$ [of f 0 f i g 0 A] inv <dpll-bj (f i) (g 0)>
  unfolding cdclNOT-NOT-all-inv-def by linarith

moreover have cdclNOT-i: cdclNOT** (f 0) (g 0)
  using rtranclp-learn-or-forget-cdclNOT[of f 0 f i] <learn-or-forget** (f 0) (f i)>
  cdclNOT[of i] unfolding g-def by auto
moreover have  $\bigwedge i. \text{cdcl}_{NOT} (g \ i) \ (g \ (\text{Suc } i))$ 

```

```

    using cdclNOT g-def by auto
moreover have cdclNOT-NOT-all-inv A (g 0)
    using inv cdclNOT-i rtrancpl-cdclNOT-trail-clauses-bound g-def cdclNOT-NOT-all-inv by auto
ultimately obtain j where j:  $\bigwedge i. i \geq j \implies \text{learn-or-forget } (g \ i) \ (g \ (\text{Suc } i))$ 
    using IH unfolding  $\mu[\text{symmetric}]$  by presburger
show ?thesis
  proof
    {
      fix k
      assume  $k \geq j + \text{Suc } i$ 
      then have learn-or-forget (f k) (f (Suc k))
        using j[of k-Suc i] unfolding g-def by auto
    }
    then show  $\forall k \geq j + \text{Suc } i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
      by auto
  qed
qed
next
case 0 note H = this(1) and cdclNOT = this(2) and inv = this(3)
show ?case
  proof (rule ccontr)
    assume  $\neg ?case$ 
    then have j:  $\exists i. \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i) \ (f \ (\text{Suc } i))$ 
      by blast
    obtain i where
       $\neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i) \ (f \ (\text{Suc } i))$  and
       $\forall k < i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
    proof -
      obtain i0 where  $\neg \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i_0) \ (f \ (\text{Suc } i_0))$ 
        using j by auto
      then have  $\{i. i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i) \ (f \ (\text{Suc } i))\} \neq \{\}$ 
        by auto
      let ?I =  $\{i. i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i) \ (f \ (\text{Suc } i))\}$ 
      let ?i = Min ?I
      have finite ?I
        by auto
      have  $\neg \text{learn } (f \ ?i) \ (f \ (\text{Suc } ?i)) \wedge \neg \text{forget}_{\text{NOT}} (f \ ?i) \ (f \ (\text{Suc } ?i))$ 
        using Min-in[OF (finite ?I) (?I  $\neq \{\}$ )] by auto
      moreover have  $\forall k < ?i. \text{learn-or-forget } (f \ k) \ (f \ (\text{Suc } k))$ 
        using Min.coboundedI[of  $\{i. i \leq i_0 \wedge \neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i) \ (f \ (\text{Suc } i))\}$ , simplified]
        by (meson  $\neg \text{learn } (f \ i_0) \ (f \ (\text{Suc } i_0)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i_0) \ (f \ (\text{Suc } i_0))$ ) less-imp-le
        dual-order.trans not-le
      ultimately show ?thesis using that by blast
    qed
  have dpll-bj (f i) (f (Suc i))
    using  $\neg \text{learn } (f \ i) \ (f \ (\text{Suc } i)) \wedge \neg \text{forget}_{\text{NOT}} (f \ i) \ (f \ (\text{Suc } i))$  cdclNOT cdclNOT.cases
    by blast
  {
    fix j
    assume  $j \leq i$ 
    then have learn-or-forget** (f 0) (f j)
      apply (induction j)
      apply simp
      by (metis (no-types, lifting) Suc-leD Suc-le-lessD rtrancpl.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 learn-or-forget** (f 0) (f i) by blast

then show False
  using learn-or-forget-dpll- $\mu_C$ [of f 0 f i f (Suc i) A] inv 0
  ⟨dpll-bj (f i) (f (Suc i))⟩ unfolding cdclNOT-NOT-all-inv-def by linarith
qed
qed

```

lemma wf-cdcl_{NOT}-no-learn-and-forget-infinite-chain:

```

assumes
  no-infinite-lf:  $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i)))$ 
shows wf {(T, S). cdclNOT S T  $\wedge$  cdclNOT-NOT-all-inv A S}
  (is wf {(T, S). cdclNOT S T  $\wedge$  ?inv S})
unfolding wf-iff-no-infinite-down-chain
proof (rule ccontr)
  assume  $\neg \neg (\exists f. \forall i. (f (Suc i), f i) \in \{(T, S). \text{cdcl}_{NOT} S T \wedge ?inv S\})$ 
  then obtain f where
     $\forall i. \text{cdcl}_{NOT} (f i) (f (Suc i)) \wedge ?inv (f i)$ 
  by fast
  then have  $\exists j. \forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i))$ 
    using infinite-cdclNOT-exists-learn-and-forget-infinite-chain[of f] by meson
  then show False using no-infinite-lf by blast
qed

```

lemma inv-and-tranclp-cdcl_{NOT}-tranclp-cdcl_{NOT}-and-inv:

```

cdclNOT++ S T  $\wedge$  cdclNOT-NOT-all-inv A S  $\longleftrightarrow (\lambda S T. \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT}$ -NOT-all-inv A
S)++ S T
(is ?A  $\wedge$  ?I  $\longleftrightarrow$  ?B)
proof
  assume ?A  $\wedge$  ?I
  then have ?A and ?I by blast+
  then show ?B
    apply induction
    apply (simp add: tranclp.r-into-trancl)
    by (subst tranclp.simps) (auto intro: cdclNOT-NOT-all-inv tranclp-into-rtranclp)
next
  assume ?B
  then have ?A by induction auto
  moreover have ?I using ⟨?B⟩ tranclpD by fastforce
  ultimately show ?A  $\wedge$  ?I by blast
qed

```

lemma wf-tranclp-cdcl_{NOT}-no-learn-and-forget-infinite-chain:

```

assumes
  no-infinite-lf:  $\bigwedge f j. \neg (\forall i \geq j. \text{learn-or-forget } (f i) (f (Suc i)))$ 
shows wf {(T, S). cdclNOT++ S T  $\wedge$  cdclNOT-NOT-all-inv A S}
using wf-trancl[OF wf-cdclNOT-no-learn-and-forget-infinite-chain[OF no-infinite-lf]]
apply (rule wf-subset)
by (auto simp: trancl-set-tranclp inv-and-tranclp-cdclNOT-tranclp-cdclNOT-and-inv)

```

lemma cdcl_{NOT}-final-state:

```

assumes
  n-s: no-step cdclNOT S and
  inv: cdclNOT-NOT-all-inv A S and
  decomp: all-decomposition-implies-m (clausesNOT S) (get-all-ann-decomposition (trail S))

```

shows *unsatisfiable* (*set-mset* (*clauses*_{NOT} *S*))
 \vee (*trail* *S* \models_{asm} *clauses*_{NOT} *S* \wedge *satisfiable* (*set-mset* (*clauses*_{NOT} *S*)))

proof –

have *n-s'*: *no-step dpll-bj S*
using *n-s* **by** (*auto simp: cdcl*_{NOT}.*simps*)
show *?thesis*
apply (*rule dpll-backjump-final-state*[*of S A*])
using *inv decomp n-s'* **unfolding** *cdcl*_{NOT}-*NOT-all-inv-def* **by** *auto*

qed

lemma *full-cdcl*_{NOT}-*final-state*:

assumes

full: *full cdcl*_{NOT} *S T* **and**

inv: *cdcl*_{NOT}-*NOT-all-inv A S* **and**

n-d: *no-dup* (*trail S*) **and**

decomp: *all-decomposition-implies-m* (*clauses*_{NOT} *S*) (*get-all-ann-decomposition* (*trail S*))

shows *unsatisfiable* (*set-mset* (*clauses*_{NOT} *T*))

\vee (*trail T* \models_{asm} *clauses*_{NOT} *T* \wedge *satisfiable* (*set-mset* (*clauses*_{NOT} *T*)))

proof –

have *st*: *cdcl*_{NOT}** *S T* **and** *n-s*: *no-step cdcl*_{NOT} *T*

using *full* **unfolding** *full-def* **by** *blast+*

have *n-s'*: *cdcl*_{NOT}-*NOT-all-inv A T*

using *cdcl*_{NOT}-*NOT-all-inv inv st* **by** *blast*

moreover have *all-decomposition-implies-m* (*clauses*_{NOT} *T*) (*get-all-ann-decomposition* (*trail T*))

using *cdcl*_{NOT}-*NOT-all-inv-def decomp inv rtranclp-cdcl*_{NOT}-*all-decomposition-implies st* **by** *auto*

ultimately show *?thesis*

using *cdcl*_{NOT}-*final-state n-s* **by** *blast*

qed

end — end of *conflict-driven-clause-learning*

Termination

To prove termination we need to restrict learn and forget. Otherwise we could forget and relearn the exact same clause over and over. A first idea is to forbid removing clauses that can be used to backjump. This does not change the rules of the calculus. A second idea is to “merge” backjump and learn: that way, though closer to implementation, needs a change of the rules, since the backjump-rule learns the clause used to backjump.

Restricting learn and forget

locale *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt* =

*dpll-state trail clauses*_{NOT} *prepend-trail tl-trail add-cl*_{NOT} *remove-cl*_{NOT} +
*conflict-driven-clause-learning trail clauses*_{NOT} *prepend-trail tl-trail add-cl*_{NOT} *remove-cl*_{NOT}
inv backjump-conds propagate-conds

$\lambda C S. \text{distinct-mset } C \wedge \neg \text{tautology } C \wedge \text{learn-restrictions } C S \wedge$

$(\exists F K d F' C' L. \text{trail } S = F' @ \text{Decided } K \# F \wedge C = C' + \{\#L\} \wedge F \models_{as} C \text{Not } C'$
 $\wedge C' + \{\#L\} \notin \# \text{ clauses}_{NOT} S)$

$\lambda C S. \neg (\exists F' F K d L. \text{trail } S = F' @ \text{Decided } K \# F \wedge F \models_{as} C \text{Not } (\text{remove1-mset } L C))$
 $\wedge \text{forget-restrictions } C S$

for

trail :: '*st* \Rightarrow ('*v*, *unit*) *ann-lits* **and**

*clauses*_{NOT} :: '*st* \Rightarrow '*v* *clauses* **and**

prepend-trail :: ('*v*, *unit*) *ann-lit* \Rightarrow '*st* \Rightarrow '*st* **and**

tl-trail :: '*st* \Rightarrow '*st* **and**

```

add-clNOT :: 'v clause ⇒ 'st ⇒ 'st and
remove-clNOT :: 'v clause ⇒ 'st ⇒ 'st and
inv :: 'st ⇒ bool and
backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
propagate-conds :: ('v, unit) ann-lit ⇒ '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:  $\bigwedge T. \text{dpll-bj } S \ T \implies P \ S \ T$  and
learning:
 $\bigwedge C \ F \ K \ F' \ C' \ L \ T. \text{clauses}_{NOT} \ S \models_{pm} C \implies$ 
 $\text{atms-of } C \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} \ S) \cup \text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \implies$ 
 $\text{distinct-mset } C \implies$ 
 $\neg \text{tautology } C \implies$ 
 $\text{learn-restrictions } C \ S \implies$ 
 $\text{trail } S = F' @ \text{Decided } K \ \# \ F \implies$ 
 $C = C' + \{\#L\# \} \implies$ 
 $F \models_{as} CNot \ C' \implies$ 
 $C' + \{\#L\# \} \notin \# \text{clauses}_{NOT} \ S \implies$ 
 $T \sim \text{add-cl}_{NOT} \ C \ S \implies$ 
 $P \ S \ T$  and
forgetting:  $\bigwedge C \ T. \text{removeAll-mset } C \ (\text{clauses}_{NOT} \ S) \models_{pm} C \implies$ 
 $C \in \# \text{clauses}_{NOT} \ S \implies$ 
 $\neg(\exists F' \ F \ K \ L. \text{trail } S = F' @ \text{Decided } K \ \# \ F \wedge F \models_{as} CNot \ (C - \{\#L\# \})) \implies$ 
 $T \sim \text{remove-cl}_{NOT} \ C \ S \implies$ 
 $\text{forget-restrictions } C \ S \implies$ 
 $P \ S \ T$ 
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:
cdclNOT** 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 (clausesNOT T - clausesNOT S)
 $\subseteq$  simple-clss (atms-of-mm (clausesNOT S)  $\cup$  atm-of ' lits-of-l (trail S))
proof
fix C assume C: C  $\in$  set-mset (clausesNOT T - clausesNOT S)
have distinct-mset C  $\neg$ tautology C using learn C n-d by (elim learnNOTE; auto)+
then have C  $\in$  simple-clss (atms-of C)
  using distinct-mset-not-tautology-implies-in-simple-clss by blast

```

moreover have $\text{atms-of } C \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of ' lits-of-l } (\text{trail } S)$
using $\text{learn } C \text{ n-d by } (\text{elim learn}_{NOT} E) (\text{auto simp: atms-of-ms-def atms-of-def image-Un true-annots-CNot-all-atms-defined})$
moreover have $\text{finite } (\text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of ' lits-of-l } (\text{trail } S))$
by auto
ultimately show $C \in \text{simple-clss } (\text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of ' lits-of-l } (\text{trail } S))$
using $\text{simple-clss-mono by } (\text{metis (no-types) insert-subset mk-disjoint-insert})$
qed

definition $\text{conflicting-bj-clss } S \equiv$
 $\{C + \{\#L\# \} \mid C \text{ L. } C + \{\#L\# \} \in \# \text{ clauses}_{NOT} S \wedge \text{distinct-mset } (C + \{\#L\# \})$
 $\wedge \neg \text{tautology } (C + \{\#L\# \})$
 $\wedge (\exists F' K F. \text{trail } S = F' @ \text{Decided } K \# F \wedge F \models_{as} CNot C)\}$

lemma $\text{conflicting-bj-clss-remove-cl}_{NOT}[\text{simp}]$:
 $\text{conflicting-bj-clss } (\text{remove-cl}_{NOT} C S) = \text{conflicting-bj-clss } S - \{C\}$
unfolding $\text{conflicting-bj-clss-def by fastforce}$

lemma $\text{conflicting-bj-clss-remove-cl}_{NOT}'[\text{simp}]$:
 $T \sim \text{remove-cl}_{NOT} C S \implies \text{conflicting-bj-clss } T = \text{conflicting-bj-clss } S - \{C\}$
unfolding $\text{conflicting-bj-clss-def by fastforce}$

lemma $\text{conflicting-bj-clss-add-cl}_{NOT}\text{-state-eq}$:
assumes
 $T: T \sim \text{add-cl}_{NOT} C' S$ **and**
 $n\text{-d: no-dup } (\text{trail } S)$
shows $\text{conflicting-bj-clss } T$
 $= \text{conflicting-bj-clss } S$
 $\cup (\text{if } \exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$
 $\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Decided } K \# F \wedge F \models_{as} CNot C)$
 $\text{then } \{C'\} \text{ else } \{\})$

proof –
def $P \equiv \lambda C L T. \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \}) \wedge$
 $(\exists F' K F. \text{trail } T = F' @ \text{Decided } K \# F \wedge F \models_{as} CNot C)$
have $\text{conf: } \bigwedge T. \text{conflicting-bj-clss } T = \{C + \{\#L\# \} \mid C L. C + \{\#L\# \} \in \# \text{ clauses}_{NOT} T \wedge P C L T\}$
unfolding $\text{conflicting-bj-clss-def } P\text{-def by auto}$
have $P\text{-S-T: } \bigwedge C L. P C L T = P C L S$
using $T \text{ n-d unfolding } P\text{-def by auto}$
have $P: \text{conflicting-bj-clss } T = \{C + \{\#L\# \} \mid C L. C + \{\#L\# \} \in \# \text{ clauses}_{NOT} S \wedge P C L T\} \cup$
 $\{C + \{\#L\# \} \mid C L. C + \{\#L\# \} \in \# \{\#C'\# \} \wedge P C L T\}$
using $T \text{ n-d unfolding conf by auto}$
moreover have $\{C + \{\#L\# \} \mid C L. C + \{\#L\# \} \in \# \text{ clauses}_{NOT} S \wedge P C L T\} = \text{conflicting-bj-clss } S$
using $T \text{ n-d unfolding } P\text{-def conflicting-bj-clss-def by auto}$
moreover have $\{C + \{\#L\# \} \mid C L. C + \{\#L\# \} \in \# \{\#C'\# \} \wedge P C L T\} =$
 $(\text{if } \exists C L. C' = C + \{\#L\# \} \wedge P C L S \text{ then } \{C'\} \text{ else } \{\})$
using $n\text{-d } T \text{ by (force simp: } P\text{-S-T)}$
ultimately show $?thesis \text{ unfolding } P\text{-def by presburger}$
qed

lemma $\text{conflicting-bj-clss-add-cl}_{NOT}$:
 $\text{no-dup } (\text{trail } S) \implies$
 $\text{conflicting-bj-clss } (\text{add-cl}_{NOT} C' S)$
 $= \text{conflicting-bj-clss } S$
 $\cup (\text{if } \exists C L. C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$

$\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Decided } K \# F \wedge F \models_{as} C \text{Not } C)$
 then $\{C'\}$ else $\{\}$)
using *conflicting-bj-clss-add-clss_{NOT}-state-eq* **by** *auto*

lemma *conflicting-bj-clss-incl-clauses*:
conflicting-bj-clss $S \subseteq \text{set-mset } (\text{clauses}_{NOT} S)$
unfolding *conflicting-bj-clss-def* **by** *auto*

lemma *finite-conflicting-bj-clss[simp]*:
finite (*conflicting-bj-clss* S)
using *conflicting-bj-clss-incl-clauses[of S]* *rev-finite-subset* **by** *blast*

lemma *learn-conflicting-increasing*:
no-dup (*trail* S) \implies *learn* $S T \implies$ *conflicting-bj-clss* $S \subseteq$ *conflicting-bj-clss* T
apply (*elim learn_{NOT}E*)
by (*subst conflicting-bj-clss-add-clss_{NOT}-state-eq*[*of T*]) *auto*

abbreviation *conflicting-bj-clss-yet* $b S \equiv$
 $3 \wedge b - \text{card } (\text{conflicting-bj-clss } S)$

abbreviation $\mu_L :: \text{nat} \Rightarrow 'st \Rightarrow \text{nat} \times \text{nat}$ **where**
 $\mu_L b S \equiv (\text{conflicting-bj-clss-yet } b S, \text{card } (\text{set-mset } (\text{clauses}_{NOT} S)))$

lemma *remove1-mset-single-add-if*:
remove1-mset $L (C + \{\#L'\#\}) = (\text{if } L = L' \text{ then } C \text{ else } \text{remove1-mset } L C + \{\#L'\#\})$
by (*auto simp: multiset-eq-iff*)

lemma *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*:
assumes *forget_{NOT}* $S T$
shows *conflicting-bj-clss* $S =$ *conflicting-bj-clss* T
using *assms* **apply** (*elim forget_{NOT}E*)
apply *rule*
apply (*subst conflicting-bj-clss-remove-clss_{NOT}'*[*of T*], *simp*)
apply (*fastforce simp: conflicting-bj-clss-def remove1-mset-single-add-if split: if-splits*)
apply *fastforce*
done

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 $\text{card } (\text{set-mset } (\text{clauses}_{NOT} S)) > 0$
using *forget_{NOT}* **by** (*elim forget_{NOT}E*) (*auto simp: size-mset-removeAll-mset-le-iff card-gt-0-iff*)
then have $\text{card } (\text{set-mset } (\text{clauses}_{NOT} T)) < \text{card } (\text{set-mset } (\text{clauses}_{NOT} S))$
using *forget_{NOT}* **by** (*elim forget_{NOT}E*) (*auto simp: size-mset-removeAll-mset-le-iff*)
then show *?thesis*
unfolding *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*[*OF forget_{NOT}*]
by *auto*
qed

lemma *set-condition-or-split*:
 $\{a. (a = b \vee Q a) \wedge S a\} = (\text{if } S b \text{ then } \{b\} \text{ else } \{\}) \cup \{a. Q a \wedge S a\}$
by *auto*

lemma *set-insert-neq*:
 $A \neq \text{insert } a A \iff a \notin A$

by auto

lemma *learn- μ_L -decrease*:

assumes *learnST*: *learn* *S* *T* **and** *n-d*: *no-dup* (*trail* *S*) **and**
A: *atms-of-mm* (*clauses*_{NOT} *S*) \cup *atm-of* ‘*lits-of-l* (*trail* *S*) \subseteq *A* **and**
fin-A: *finite* *A*
shows (μ_L (*card* *A*) *T*, μ_L (*card* *A*) *S*) \in *less-than* $\langle *lex* \rangle$ *less-than*

proof –

have [*simp*]: (*atms-of-mm* (*clauses*_{NOT} *T*) \cup *atm-of* ‘*lits-of-l* (*trail* *T*))
 $=$ (*atms-of-mm* (*clauses*_{NOT} *S*) \cup *atm-of* ‘*lits-of-l* (*trail* *S*))
using *learnST* *n-d* **by** (*elim learn*_{NOT} *E*) *auto*

then have *card* (*atms-of-mm* (*clauses*_{NOT} *T*) \cup *atm-of* ‘*lits-of-l* (*trail* *T*))
 $=$ *card* (*atms-of-mm* (*clauses*_{NOT} *S*) \cup *atm-of* ‘*lits-of-l* (*trail* *S*))
by (*auto intro!*: *card-mono*)

then have \exists : ($\exists::nat$) \wedge *card* (*atms-of-mm* (*clauses*_{NOT} *T*) \cup *atm-of* ‘*lits-of-l* (*trail* *T*))
 $=$ $\exists \wedge$ *card* (*atms-of-mm* (*clauses*_{NOT} *S*) \cup *atm-of* ‘*lits-of-l* (*trail* *S*))
by (*auto intro*: *power-mono*)

moreover have *conflicting-bj-clss* *S* \subseteq *conflicting-bj-clss* *T*
using *learnST* *n-d* **by** (*simp add*: *learn-conflicting-increasing*)

moreover have *conflicting-bj-clss* *S* \neq *conflicting-bj-clss* *T*
using *learnST*

proof (*elim learn*_{NOT} *E*, *goal-cases*)

case (*1 C*) **note** *clss-S* = *this*(*1*) **and** *atms-C* = *this*(*2*) **and** *inv* = *this*(*3*) **and** *T* = *this*(*4*)

then obtain *F K F' C' L* **where**

tr-S: *trail* *S* = *F' @ Decided* *K # F* **and**

C: *C* = *C' + {#L#}* **and**

F: *F* \models_{as} *CNot* *C'* **and**

C-S: *C' + {#L#}* \notin *clauses*_{NOT} *S*

by *blast*

moreover have *distinct-mset* *C* \neg *tautology* *C* **using** *inv* **by** *blast+*

ultimately have *C' + {#L#}* \in *conflicting-bj-clss* *T*

using *T n-d unfolding* *conflicting-bj-clss-def* **by** *fastforce*

moreover have *C' + {#L#}* \notin *conflicting-bj-clss* *S*

using *C-S unfolding* *conflicting-bj-clss-def* **by** *auto*

ultimately show *?case* **by** *blast*

qed

moreover have *fin-T*: *finite* (*conflicting-bj-clss* *T*)

using *learnST* **by** *induction* (*auto simp add*: *conflicting-bj-clss-add-clss*_{NOT})

ultimately have *card* (*conflicting-bj-clss* *T*) \geq *card* (*conflicting-bj-clss* *S*)

using *card-mono* **by** *blast*

moreover

have *fin'*: *finite* (*atms-of-mm* (*clauses*_{NOT} *T*) \cup *atm-of* ‘*lits-of-l* (*trail* *T*))
by *auto*

have *1*: *atms-of-ms* (*conflicting-bj-clss* *T*) \subseteq *atms-of-mm* (*clauses*_{NOT} *T*)
unfolding *conflicting-bj-clss-def* *atms-of-ms-def* **by** *auto*

have *2*: $\bigwedge x. x \in$ *conflicting-bj-clss* *T* $\implies \neg$ *tautology* *x* \wedge *distinct-mset* *x*
unfolding *conflicting-bj-clss-def* **by** *auto*

have *T*: *conflicting-bj-clss* *T*

\subseteq *simple-clss* (*atms-of-mm* (*clauses*_{NOT} *T*) \cup *atm-of* ‘*lits-of-l* (*trail* *T*))

by *standard* (*meson* *1 2 fin'* ‘*finite* (*conflicting-bj-clss* *T*)’ *simple-clss-mono*
distinct-mset-set-def *simplified-in-simple-clss* *subsetCE* *sup.coboundedI1*)

moreover

then have $\#$: $\exists \wedge$ *card* (*atms-of-mm* (*clauses*_{NOT} *T*) \cup *atm-of* ‘*lits-of-l* (*trail* *T*))
 \geq *card* (*conflicting-bj-clss* *T*)

```

  by (meson Nat.le-trans simple-clss-card simple-clss-finite card-mono fin')
have atms-of-mm (clausesNOT T)  $\cup$  atm-of ' lits-of-l (trail T)  $\subseteq$  A
  using learnNOTE[OF learnST] A by simp
then have 3  $\wedge$  (card A)  $\geq$  card (conflicting-bj-clss T)
  using # fin-A by (meson simple-clss-card simple-clss-finite
    simple-clss-mono calculation(2) card-mono dual-order.trans)
ultimately show ?thesis
  using psubset-card-mono[OF fin-T ]
  unfolding less-than-iff lex-prod-def by clarify
  (meson (conflicting-bj-clss S  $\neq$  conflicting-bj-clss T)
    (conflicting-bj-clss S  $\subseteq$  conflicting-bj-clss T)
    diff-less-mono2 le-less-trans not-le psubsetI)
qed

```

We have to assume the following:

- *inv S*: the invariant holds in the initial state.
- *A* is a (finite *finite A*) superset of the literals in the trail *atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A* and in the clauses *atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A*. This can be the set of all the literals in the starting set of clauses.
- *no-dup (trail S)*: no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} where

$\mu_{CDCL} A T \equiv ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T),$
 $\quad \text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) T, \text{card} (\text{set-mset} (\text{clauses}_{NOT} T)))$

lemma *cdcl_{NOT}-decreasing-measure*:

```

assumes
  cdclNOT S T and
  inv: inv S and
  atm-clss: atms-of-mm (clausesNOT S)  $\subseteq$  atms-of-ms A and
  atm-lits: atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-ms A and
  n-d: no-dup (trail S) and
  fin-A: finite A
shows ( $\mu_{CDCL} A T, \mu_{CDCL} A S$ )
   $\in$  less-than <*lex*> (less-than <*lex*> less-than)
  using assms(1)
proof induction
  case (c-dpll-bj T)
  from dpll-bj-trail-mes-decreasing-prop[OF this(1) inv atm-clss atm-lits n-d fin-A]
  show ?case unfolding  $\mu_{CDCL}$ -def
    by (meson in-lex-prod less-than-iff)
next
  case (c-learn T) note learn = this(1)
  then have S: trail S = trail T
    using inv atm-clss atm-lits n-d fin-A
    by (elim learnNOTE) auto
  show ?case
    using learn- $\mu_L$ -decrease[OF learn n-d, of atms-of-ms A] atm-clss atm-lits fin-A n-d
    unfolding S  $\mu_{CDCL}$ -def by auto
next
  case (c-forgetNOT T) note forgetNOT = this(1)
  have trail S = trail T using forgetNOT by induction auto

```

then show *?case*
using *forget- μ_L -decrease*[*OF forget_{NOT}*] **unfolding** *μ_{CDCL} -def* **by** *auto*
qed

lemma *wf-cdcl_{NOT}-restricted-learning*:

assumes *finite A*

shows *wf {(T, S).*

(atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A \wedge atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A
 \wedge no-dup (trail S)
 \wedge inv S)
 \wedge cdcl_{NOT} S T }

by (*rule wf-wf-if-measure'*[*of less-than <*lex*> (less-than <*lex*> less-than)*])
(auto intro: cdcl_{NOT}-decreasing-measure[*OF - - - - assms*])

definition $\mu_C' :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_C' A T \equiv \mu_C (1 + \text{card (atms-of-ms A)}) (2 + \text{card (atms-of-ms A)}) (\text{trail-weight T})$

definition $\mu_{CDCL}' :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_{CDCL}' A T \equiv$

$((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T) * (1 + 3^{\text{card (atms-of-ms A)}}) * 2$
 $+ \text{conflicting-bj-clss-yet (card (atms-of-ms A)) T} * 2$
 $+ \text{card (set-mset (clauses_{NOT} T))}$

lemma *cdcl_{NOT}-decreasing-measure'*:

assumes

cdcl_{NOT} S T and

inv: inv S and

atms-clss: atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A and

atms-trail: atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A and

n-d: no-dup (trail S) and

fin-A: finite A

shows $\mu_{CDCL}' A T < \mu_{CDCL}' A S$

using *assms(1)*

proof (*induction rule: cdcl_{NOT}-learn-all-induct*)

case (*dpll-bj T*)

then have $(2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T$

$< (2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A S$

using *dpll-bj-trail-mes-decreasing-prop fin-A inv n-d atms-clss atms-trail*

unfolding μ_C' -def **by** *blast*

then have *XX*: $((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T) + 1$

$\leq (2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A S$

by *auto*

from *mult-le-mono1*[*OF this, of 1 + 3^{card (atms-of-ms A)}*]

have $((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T) * (1 + 3^{\text{card (atms-of-ms A)}}) + (1 + 3^{\text{card (atms-of-ms A)}})$

$\leq ((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A S)$

$* (1 + 3^{\text{card (atms-of-ms A)}})$

unfolding *Nat.add-mult-distrib*

by *presburger*

moreover

have *cl-T-S*: $\text{clauses}_{NOT} T = \text{clauses}_{NOT} S$

using *dpll-bj.hyps inv dpll-bj-clauses* **by** *auto*

have *conflicting-bj-clss-yet (card (atms-of-ms A)) S* $< 1 + 3^{\text{card (atms-of-ms A)}}$

by *simp*

ultimately have $((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T)$


```

    * (1 + 3 ^ 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 μC'DCL'-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)
  using C'
  by (metis (no-types, hide-lams) Un-subset-iff simple-clss-mono
    contra-subsetD dist distinct-mset-not-tautology-implies-in-simple-clss
    dual-order.trans atms-C atms-clss atms-trail tauto)
  moreover have conflicting-bj-clss S ⊆ simple-clss (atms-of-ms A)
  proof
    fix x :: 'v clause
    assume x ∈ conflicting-bj-clss S
    then have x ∈ # clausesNOT S ∧ distinct-mset x ∧ ¬ tautology x
    unfolding conflicting-bj-clss-def 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
      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)
moreover have [simp]: conflicting-bj-clss (add-clNOT C S) = conflicting-bj-clss S ∪ {C}
  using dist tauto F-C by (subst conflicting-bj-clss-add-clNOT[OF n-d]) (force simp: C' tr-S n-d)
ultimately have [simp]: conflicting-bj-clss-yet (card (atms-of-ms A)) S
  = Suc (conflicting-bj-clss-yet (card (atms-of-ms A)) (add-clNOT C S))
  by simp

```

have 1: $\text{clauses}_{NOT} T = \text{clauses}_{NOT} (\text{add-cl}_{NOT} C S)$ **using** T **by** *auto*
have 2: $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) T$
 $= \text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S)$
using T **unfolding** $\text{conflicting-bj-clss-def}$ **by** *auto*
have 3: $\mu_C' A T = \mu_C' A (\text{add-cl}_{NOT} C S)$
using T **unfolding** $\mu_C'\text{-def}$ **by** *auto*
have $((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A (\text{add-cl}_{NOT} C S))$
 $* (1 + 3 \wedge \text{card} (\text{atms-of-ms } A)) * 2$
 $= ((2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A)) - \mu_C' A S)$
 $* (1 + 3 \wedge \text{card} (\text{atms-of-ms } A)) * 2$
using $n\text{-d}$ **unfolding** $\mu_C'\text{-def}$ **by** *auto*
moreover
have $\text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) (\text{add-cl}_{NOT} C S)$
 $* 2$
 $+ \text{card} (\text{set-mset} (\text{clauses}_{NOT} (\text{add-cl}_{NOT} C S)))$
 $< \text{conflicting-bj-clss-yet} (\text{card} (\text{atms-of-ms } A)) S * 2$
 $+ \text{card} (\text{set-mset} (\text{clauses}_{NOT} S))$
by (*simp add: C' C-new n-d*)
ultimately show $?case$ **unfolding** $\mu_{CDCL}'\text{-def 1 2 3}$ **by** *presburger*
next
case ($\text{forget}_{NOT} C T$) **note** $T = \text{this}(4)$
have [*simp*]: $\mu_C' A (\text{remove-cl}_{NOT} C S) = \mu_C' A S$
unfolding $\mu_C'\text{-def}$ **by** *auto*
have $\text{forget}_{NOT} S T$
apply (*rule forget_{NOT}.intros*) **using** forget_{NOT} **by** *auto*
then have $\text{conflicting-bj-clss } T = \text{conflicting-bj-clss } S$
using *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched* **by** *blast*
moreover have $\text{card} (\text{set-mset} (\text{clauses}_{NOT} T)) < \text{card} (\text{set-mset} (\text{clauses}_{NOT} S))$
by (*metis T card-Diff1-less clauses-remove-cl_{NOT} finite-set-mset forget_{NOT}.hyps(2)*
order-refl set-mset-minus-replicate-mset(1) state-eq_{NOT}-clauses)
ultimately show $?case$ **unfolding** $\mu_{CDCL}'\text{-def}$
using $T \mu_C' A (\text{remove-cl}_{NOT} C S) = \mu_C' A S$ **by** (*metis (no-types) add-le-cancel-left*
 $\mu_C'\text{-def not-le state-eq_{NOT}-trail})
qed$

lemma $\text{cdcl}_{NOT}\text{-clauses-bound}$:

assumes
 $\text{cdcl}_{NOT} S T$ **and**
 $\text{inv } S$ **and**
 $\text{atms-of-mm} (\text{clauses}_{NOT} S) \subseteq A$ **and**
 $\text{atm-of } (\text{lits-of-l } (\text{trail } S)) \subseteq A$ **and**
 $n\text{-d: no-dup } (\text{trail } S)$ **and**
 $\text{fin-A}[\text{simp}]: \text{finite } A$
shows $\text{set-mset} (\text{clauses}_{NOT} T) \subseteq \text{set-mset} (\text{clauses}_{NOT} S) \cup \text{simple-clss } A$
using *assms*
proof (*induction rule: cdcl_{NOT}-learn-all-induct*)
case *dpll-bj*
then show $?case$ **using** *dpll-bj-clauses* **by** *simp*
next
case forget_{NOT}
then show $?case$ **using** $\text{clauses-remove-cl}_{NOT}$ **unfolding** $\text{state-eq}_{NOT}\text{-def}$ **by** *auto*
next
case ($\text{learn } C F K d F' C' L$) **note** $\text{atms-C} = \text{this}(2)$ **and** $\text{dist} = \text{this}(3)$ **and** $\text{tauto} = \text{this}(4)$ **and**
 $T = \text{this}(10)$ **and** $\text{atms-clss-S} = \text{this}(12)$ **and** $\text{atms-trail-S} = \text{this}(13)$
have $\text{atms-of } C \subseteq A$
using $\text{atms-C atms-clss-S atms-trail-S}$ **by** *fast*

then have *simple-clss* (*atms-of* *C*) \subseteq *simple-clss* *A*
by (*simp add: simple-clss-mono*)
then have *C* \in *simple-clss* *A*
using *finite dist tauto* **by** (*auto dest: distinct-mset-not-tautology-implies-in-simple-clss*)
then show ?*case* **using** *T n-d* **by** *auto*
qed

lemma *rtrancpl-cdcl_{NOT}-clauses-bound*:

assumes
*cdcl_{NOT}** S T* **and**
inv S **and**
atms-of-mm (clauses_{NOT} S) \subseteq A **and**
atm-of '(lits-of-l (trail S)) \subseteq A **and**
n-d: no-dup (trail S) **and**
finite: finite A
shows *set-mset (clauses_{NOT} T) \subseteq set-mset (clauses_{NOT} S) \cup simple-clss A*
using *assms(1-5)*
proof *induction*
case *base*
then show ?*case* **by** *simp*
next
case (*step T U*) **note** *st = this(1)* **and** *cdcl_{NOT} = this(2)* **and** *IH = this(3)[OF this(4-7)]* **and**
inv = this(4) **and** *atms-clss-S = this(5)* **and** *atms-trail-S = this(6)* **and** *finite-clss-S = this(7)*
have *inv T*
using *rtrancpl-cdcl_{NOT}-inv st inv* **by** *blast*
moreover have *atms-of-mm (clauses_{NOT} T) \subseteq A* **and** *atm-of '(lits-of-l (trail T)) \subseteq A*
using *rtrancpl-cdcl_{NOT}-trail-clauses-bound[OF st] inv atms-clss-S atms-trail-S n-d* **by** *auto*
moreover have *no-dup (trail T)*
using *rtrancpl-cdcl_{NOT}-no-dup[OF st (inv S) n-d]* **by** *simp*
ultimately have *set-mset (clauses_{NOT} U) \subseteq set-mset (clauses_{NOT} T) \cup simple-clss A*
using *cdcl_{NOT} finite n-d* **by** (*auto simp: cdcl_{NOT}-clauses-bound*)
then show ?*case* **using** *IH* **by** *auto*
qed

lemma *rtrancpl-cdcl_{NOT}-card-clauses-bound*:

assumes
*cdcl_{NOT}** S T* **and**
inv S **and**
atms-of-mm (clauses_{NOT} S) \subseteq A **and**
atm-of '(lits-of-l (trail S)) \subseteq A **and**
n-d: no-dup (trail S) **and**
finite: finite A
shows *card (set-mset (clauses_{NOT} T)) \leq card (set-mset (clauses_{NOT} S)) + 3 \wedge (card A)*
using *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-mm (clauses_{NOT} S) \subseteq A **and**
atm-of '(lits-of-l (trail S)) \subseteq A **and**
n-d: no-dup (trail S) **and**
finite: finite A
shows *card {C | C. C \in # clauses_{NOT} T \wedge (tautology C \vee \neg distinct-mset C)}*

$\leq \text{card } \{C \mid C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } A)$
 (is card ?T ≤ card ?S + -)
 using rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] finite
proof –
 have ?T ⊆ ?S ∪ simple-clss A
 using rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] by force
 then have card ?T ≤ card (?S ∪ 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
 NA: atms-of-mm (clauses_{NOT} S) ⊆ A and
 MA: atm-of '(lits-of-l (trail S)) ⊆ A and
 n-d: no-dup (trail S) and
 finite: finite A
shows card (set-mset (clauses_{NOT} T))
 $\leq \text{card } \{C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } A)$
 (is card ?T ≤ card ?S + -)
 using rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] finite
proof –
 have $\bigwedge x. x \in \# \text{ clauses}_{NOT} T \implies \neg \text{tautology } x \implies \text{distinct-mset } x \implies x \in \text{simple-clss } A$
 using rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] by (metis (no-types, hide-lams) Un-iff NA
 atms-of-atms-of-ms-mono simple-clss-mono contra-subsetD subset-trans
 distinct-mset-not-tautology-implies-in-simple-clss)
 then have set-mset (clauses_{NOT} T) ⊆ ?S ∪ simple-clss A
 using rtrancpl-cdcl_{NOT}-clauses-bound[OF assms] by auto
 then have card(set-mset (clauses_{NOT} T)) ≤ card (?S ∪ 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 $\mu_{CDCL}'\text{-bound} :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_{CDCL}'\text{-bound } A \ S =$
 $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * (1 + 3 \wedge \text{card } (\text{atms-of-ms } A)) * 2$
 $+ 2 * 3 \wedge (\text{card } (\text{atms-of-ms } A))$
 $+ \text{card } \{C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge (\text{card } (\text{atms-of-ms } A))$

lemma $\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}$ [simp]:

$\mu_{CDCL}'\text{-bound } A \ (\text{reduce-trail-to}_{NOT} M \ S) = \mu_{CDCL}'\text{-bound } A \ S$
unfolding $\mu_{CDCL}'\text{-bound-def}$ by auto

lemma rtrancpl-cdcl_{NOT}- $\mu_{CDCL}'\text{-bound-reduce-trail-to}_{NOT}$:

assumes
 cdcl_{NOT}** S T and
 inv S and
 atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A and
 atm-of '(lits-of-l (trail S)) ⊆ atms-of-ms A and
 n-d: no-dup (trail S) and
 finite: finite (atms-of-ms A) and
 U: U ~ 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 *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 $\text{card } (\text{set-mset } (\text{clauses}_{NOT} U))$
 $\leq \text{card } \{C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\} + 3 \wedge \text{card } (\text{atms-of-ms } A)$
 using *rtranclp-cdcl_{NOT}-card-simple-clauses-bound*[*OF assms(1-6)*] *U* by *auto*
ultimately show *?thesis*
 unfolding $\mu_{CDCL}'\text{-def}$ $\mu_{CDCL}'\text{-bound-def}$ by *linarith*
qed

lemma *rtranclp-cdcl_{NOT}- μ_{CDCL}' -bound*:

assumes
 $\text{cdcl}_{NOT}^{**} S T$ **and**
 $\text{inv } S$ **and**
 $\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**
 $n\text{-d: no-dup } (\text{trail } S)$ **and**
 $\text{finite: finite } (\text{atms-of-ms } A)$
 shows $\mu_{CDCL}' A T \leq \mu_{CDCL}'\text{-bound } A S$
proof –
 have $\mu_{CDCL}' A (\text{reduce-trail-to}_{NOT} (\text{trail } T) T) = \mu_{CDCL}' A T$
 unfolding $\mu_{CDCL}'\text{-def}$ $\mu_C'\text{-def}$ *conflicting-bj-clss-def* by *auto*
 then show *?thesis* using *rtranclp-cdcl_{NOT}- μ_{CDCL}' -bound-reduce-trail-to_{NOT}*[*OF assms, of - trail T*]
 $\text{state-eq}_{NOT}\text{-ref}$ by *fastforce*
qed

lemma *rtranclp- μ_{CDCL}' -bound-decreasing*:

assumes
 $\text{cdcl}_{NOT}^{**} S T$ **and**
 $\text{inv } S$ **and**
 $\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-of } '(\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-ms } A$ **and**
 $n\text{-d: no-dup } (\text{trail } S)$ **and**
 $\text{finite[simp]: finite } (\text{atms-of-ms } A)$
 shows $\mu_{CDCL}'\text{-bound } A T \leq \mu_{CDCL}'\text{-bound } A S$
proof –
 have $\{C. C \in \# \text{ clauses}_{NOT} T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$
 $\subseteq \{C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)\}$ (**is** $?T \subseteq ?S$)
proof (*rule Set.subsetI*)
 fix *C* **assume** $C \in ?T$
 then have *C-T*: $C \in \# \text{ clauses}_{NOT} T$ **and** *t-d*: $\text{tautology } C \vee \neg \text{distinct-mset } C$
 by *auto*
 then have $C \notin \text{simple-clss } (\text{atms-of-ms } A)$
 by (*auto dest: simple-clssE*)
 then show $C \in ?S$
 using *C-T rtranclp-cdcl_{NOT}-clauses-bound*[*OF assms*] *t-d* by *force*
qed

```

then have card { $C. C \in \# \text{ clauses}_{NOT} T \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)$ }  $\leq$ 
  card { $C. C \in \# \text{ clauses}_{NOT} S \wedge (\text{tautology } C \vee \neg \text{distinct-mset } C)$ }
by (simp add: card-mono)
then show ?thesis
  unfolding  $\mu_{CDCL}$ '-bound-def by auto
qed

end — end of conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt

```

5.2.5 CDCL with restarts

Definition

```

locale restart-ops =
  fixes
    cdclNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
    restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool
  begin
inductive cdclNOT-raw-restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  cdclNOT  $S T \Longrightarrow$  cdclNOT-raw-restart  $S T$  |
  restart  $S T \Longrightarrow$  cdclNOT-raw-restart  $S T$ 

end

locale conflict-driven-clause-learning-with-restarts =
  conflict-driven-clause-learning trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds learn-cond forget-cond
  for
    trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
    clausesNOT :: 'st  $\Rightarrow$  'v clauses and
    prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
    tl-trail :: 'st  $\Rightarrow$  'st and
    add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
    inv :: 'st  $\Rightarrow$  bool and
    backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
    propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
    learn-cond forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool
  begin

lemma cdclNOT-iff-cdclNOT-raw-restart-no-restarts:
  cdclNOT  $S T \longleftrightarrow$  restart-ops.cdclNOT-raw-restart cdclNOT ( $\lambda - . \text{False}$ )  $S T$ 
  (is ? $C S T \longleftrightarrow$  ? $R S T$ )
proof
  fix  $S T$ 
  assume ? $C S T$ 
  then show ? $R S T$  by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
next
  fix  $S T$ 
  assume ? $R S T$ 
  then show ? $C S T$ 
    apply (cases rule: restart-ops.cdclNOT-raw-restart.cases)
    using ⟨? $R S T$ ⟩ by fast+
qed

lemma cdclNOT-cdclNOT-raw-restart:

```

```

cdclNOT S T  $\implies$  restart-ops.cdclNOT-raw-restart cdclNOT restart S T
by (simp add: restart-ops.cdclNOT-raw-restart.intros(1))
end

```

Increasing restarts

To add restarts we need some assumptions on the predicate (called *cdcl*_{NOT} here):

- a function *f* that is strictly monotonic. The first step is actually only used as a restart to clean the state (e.g. to ensure that the trail is empty). Then we assume that $(1::'a) \leq f\ n$ for $(1::'a) \leq n$: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...
- a measure μ : it should decrease under the assumptions *bound-inv*, whenever a *cdcl*_{NOT} or a *restart* is done. A parameter is given to μ : for conflict- driven clause learning, it is an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.
- we also assume that the measure decrease after any *cdcl*_{NOT} step.
- an invariant on the states *cdcl*_{NOT-inv} that also holds after restarts.
- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ -*bound* taking the same parameter as μ and the initial state of the considered *cdcl*_{NOT} chain.

```

locale cdclNOT-increasing-restarts-ops =
  restart-ops cdclNOT restart for
    restart :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
    cdclNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool +
fixes
  f :: nat  $\Rightarrow$  nat and
  bound-inv :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  bool and
   $\mu$  :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat and
  cdclNOT-inv :: 'st  $\Rightarrow$  bool and
   $\mu$ -bound :: 'bound  $\Rightarrow$  'st  $\Rightarrow$  nat
assumes
  f: unbounded f and
  f-ge-1:  $\bigwedge n. n \geq 1 \implies f\ n \neq 0$  and
  bound-inv:  $\bigwedge A\ S\ T. \text{cdcl}_{NOT-inv}\ S \implies \text{bound-inv}\ A\ S \implies \text{cdcl}_{NOT}\ S\ T \implies \text{bound-inv}\ A\ T$  and
  cdclNOT-measure:  $\bigwedge A\ S\ T. \text{cdcl}_{NOT-inv}\ S \implies \text{bound-inv}\ A\ S \implies \text{cdcl}_{NOT}\ S\ T \implies \mu\ A\ T < \mu$ 
  A S and
  measure-bound2:  $\bigwedge A\ T\ U. \text{cdcl}_{NOT-inv}\ T \implies \text{bound-inv}\ A\ T \implies \text{cdcl}_{NOT}^{**}\ T\ U$ 
     $\implies \mu\ A\ U \leq \mu\text{-bound}\ A\ T$  and
  measure-bound4:  $\bigwedge A\ T\ U. \text{cdcl}_{NOT-inv}\ T \implies \text{bound-inv}\ A\ T \implies \text{cdcl}_{NOT}^{**}\ T\ U$ 
     $\implies \mu\text{-bound}\ A\ U \leq \mu\text{-bound}\ A\ T$  and
  cdclNOT-restart-inv:  $\bigwedge A\ U\ V. \text{cdcl}_{NOT-inv}\ U \implies \text{restart}\ U\ V \implies \text{bound-inv}\ A\ U \implies \text{bound-inv}$ 
  A V
and
  exists-bound:  $\bigwedge R\ S. \text{cdcl}_{NOT-inv}\ R \implies \text{restart}\ R\ S \implies \exists A. \text{bound-inv}\ A\ S$  and
  cdclNOT-inv:  $\bigwedge S\ T. \text{cdcl}_{NOT-inv}\ S \implies \text{cdcl}_{NOT}\ S\ T \implies \text{cdcl}_{NOT-inv}\ T$  and
  cdclNOT-inv-restart:  $\bigwedge S\ T. \text{cdcl}_{NOT-inv}\ S \implies \text{restart}\ S\ T \implies \text{cdcl}_{NOT-inv}\ T$ 
begin

```

lemma *cdcl*_{NOT-cdcl}_{NOT-inv}:

```

assumes
  ( $cdcl_{NOT} \sim n$ )  $S$   $T$  and
   $cdcl_{NOT-inv}$   $S$ 
shows  $cdcl_{NOT-inv}$   $T$ 
using assms by (induction  $n$  arbitrary:  $T$ ) (auto intro:bound-inv  $cdcl_{NOT-inv}$ )

lemma  $cdcl_{NOT-bound-inv}$ :
assumes
  ( $cdcl_{NOT} \sim n$ )  $S$   $T$  and
   $cdcl_{NOT-inv}$   $S$ 
  bound-inv  $A$   $S$ 
shows bound-inv  $A$   $T$ 
using assms by (induction  $n$  arbitrary:  $T$ ) (auto intro:bound-inv  $cdcl_{NOT}$ - $cdcl_{NOT-inv}$ )

lemma  $rtrancp-cdcl_{NOT-cdcl_{NOT-inv}}$ :
assumes
   $cdcl_{NOT}^{**}$   $S$   $T$  and
   $cdcl_{NOT-inv}$   $S$ 
shows  $cdcl_{NOT-inv}$   $T$ 
using assms by induction (auto intro:  $cdcl_{NOT-inv}$ )

lemma  $rtrancp-cdcl_{NOT-bound-inv}$ :
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  $rtrancp-cdcl_{NOT}$ - $cdcl_{NOT-inv}$ )

lemma  $cdcl_{NOT-comp-n-le}$ :
assumes
  ( $cdcl_{NOT} \sim (Suc\ n)$ )  $S$   $T$  and
  bound-inv  $A$   $S$ 
   $cdcl_{NOT-inv}$   $S$ 
shows  $\mu\ A\ T < \mu\ A\ S - n$ 
using assms
proof (induction  $n$  arbitrary:  $T$ )
  case 0
    then show ?case using  $cdcl_{NOT-measure}$  by auto
  next
    case ( $Suc\ n$ ) note  $IH = this(1)[OF - this(3)\ this(4)]$  and  $S-T = this(2)$  and  $b-inv = this(3)$  and
     $c-inv = this(4)$ 
    obtain  $U :: 'st$  where  $S-U: (cdcl_{NOT} \sim (Suc\ n))\ S\ U$  and  $U-T: cdcl_{NOT}\ U\ T$  using  $S-T$  by auto
    then have  $\mu\ A\ U < \mu\ A\ S - n$  using  $IH[of\ U]$  by simp
    moreover
      have bound-inv  $A$   $U$ 
      using  $S-U\ b-inv\ cdcl_{NOT-bound-inv}\ c-inv$  by blast
      then have  $\mu\ A\ T < \mu\ A\ U$  using  $cdcl_{NOT-measure}[OF - -\ U-T]\ S-U\ c-inv\ cdcl_{NOT-cdcl_{NOT-inv}}$ 
    by auto
    ultimately show ?case by linarith
  qed

lemma  $wf-cdcl_{NOT}$ :
   $wf\ \{(T, S).\ cdcl_{NOT}\ S\ T \wedge cdcl_{NOT-inv}\ S \wedge bound-inv\ A\ S\}$  (is  $wf\ ?A$ )
apply (rule  $wfP-if-measure2[of\ -\ -\ \mu\ A]$ )
using  $cdcl_{NOT-comp-n-le}[of\ 0\ -\ A]$  by auto

```


lemma *rtrancpl-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: rtrancpl-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* $cdcl_{NOT-bound-inv}\ rtrancpl-imp-relpowp\ st\ step.premis$)
moreover have $cdcl_{NOT-inv} T$
using $c-inv\ rtrancpl-cdcl_{NOT-cdcl_{NOT-inv}\ st}$ **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 $\neg(cdcl_{NOT} \sim^m) S T$
using *assms* $cdcl_{NOT-comp-n-le}[of\ m-1\ S\ T\ A]$ **by** *fastforce*

- $f\ n < m$ ensures that at least one step has been done.

inductive *cdcl_{NOT}-restart* **where**
restart-step: $(cdcl_{NOT} \sim^m) S T \implies m \geq f\ n \implies restart\ T\ U$
 $\implies cdcl_{NOT-restart}\ (S, n)\ (U, Suc\ n) \mid$
restart-full: $full1\ cdcl_{NOT}\ S\ T \implies cdcl_{NOT-restart}\ (S, n)\ (T, Suc\ n)$

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*:
 $cdcl_{NOT-restart}\ S\ T \implies cdcl_{NOT-raw-restart}^{**}\ (fst\ S)\ (fst\ T)$
proof (*induction rule: cdcl_{NOT}-restart.induct*)
case (*restart-step* $m\ S\ T\ n\ U$)
then have $cdcl_{NOT}^{**}\ S\ T$ **by** (*meson* $relpowp-imp-rtrancpl$)
then have $cdcl_{NOT-raw-restart}^{**}\ S\ T$ **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 $\langle restart\ T\ U \rangle\ cdcl_{NOT-raw-restart.intros}(2)$ **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 rtranclp-unfold fstI full1-def rtranclp-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 rtranclp-cdcl_{NOT}-cdcl_{NOT}-inv*)

done

lemma *rtranclp-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 *rtranclp-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 +*

dpll-state trail clauses_{NOT} prepend-trail tl-trail add-cl_s_{NOT} remove-cl_s_{NOT}

for

trail :: '*st* \Rightarrow ('*v*, *unit*) *ann-lits* **and**

clauses_{NOT} :: '*st* \Rightarrow '*v* *clauses* **and**

prepend-trail :: ('*v*, *unit*) *ann-lit* \Rightarrow '*st* \Rightarrow '*st* **and**

tl-trail :: '*st* \Rightarrow '*st* **and**

add-cl_s_{NOT} :: '*v* *clause* \Rightarrow '*st* \Rightarrow '*st* **and**

remove-cl_s_{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 ⇒ 'st ⇒ nat and
cdclNOT :: 'st ⇒ 'st ⇒ bool and
cdclNOT-inv :: 'st ⇒ bool and
μ-bound :: 'bound ⇒ 'st ⇒ nat +
assumes
  measure-bound:  $\bigwedge A\ T\ V\ n. \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A\ T$ 
     $\implies \text{cdcl}_{NOT}\text{-restart } (T, n) (V, \text{Suc } n) \implies \mu\ A\ V \leq \mu\text{-bound } A\ T$  and
  cdclNOT-raw-restart-μ-bound:
    cdclNOT-restart (T, a) (V, b)  $\implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A\ T$ 
     $\implies \mu\text{-bound } A\ V \leq \mu\text{-bound } A\ T$ 
begin

lemma rtrancp-cdclNOT-raw-restart-μ-bound:
  cdclNOT-restart** (T, a) (V, b)  $\implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A\ T$ 
   $\implies \mu\text{-bound } A\ V \leq \mu\text{-bound } A\ T$ 
  apply (induction rule: rtrancp-induct2)
  apply simp
  by (metis cdclNOT-raw-restart-μ-bound dual-order.trans fst-conv
    rtrancp-cdclNOT-with-restart-bound-inv rtrancp-cdclNOT-with-restart-cdclNOT-inv)

lemma cdclNOT-raw-restart-measure-bound:
  cdclNOT-restart (T, a) (V, b)  $\implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A\ T$ 
   $\implies \mu\ A\ V \leq \mu\text{-bound } A\ T$ 
  apply (cases rule: cdclNOT-restart.cases)
  apply simp
  using measure-bound relpowp-imp-rtrancp apply fastforce
  by (metis full-def full-unfold measure-bound2 prod.inject)

lemma rtrancp-cdclNOT-raw-restart-measure-bound:
  cdclNOT-restart** (T, a) (V, b)  $\implies \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A\ T$ 
   $\implies \mu\ A\ V \leq \mu\text{-bound } A\ T$ 
  apply (induction rule: rtrancp-induct2)
  apply (simp add: measure-bound2)
  by (metis dual-order.trans fst-conv measure-bound2 r-into-rtrancp rtrancp.rtrancp-refl
    rtrancp-cdclNOT-with-restart-bound-inv rtrancp-cdclNOT-with-restart-cdclNOT-inv
    rtrancp-cdclNOT-raw-restart-μ-bound)

lemma wf-cdclNOT-restart:
  wf {(T, S). cdclNOT-restart S T ∧ cdclNOT-inv (fst S)} (is wf ?A)
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain g where
    g:  $\bigwedge i. \text{cdcl}_{NOT}\text{-restart } (g\ i) (g\ (\text{Suc } i))$  and
    cdclNOT-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
    cdclNOT-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 (λi. f (snd (g i)))
  using f unfolding bounded-def by (metis add commute f less-or-eq-imp-le snd-g
    not-bounded-nat-exists-larger not-le 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: cdclNOT-restart.cases)
    apply auto[]
    using g[of Suc i] f-ge-1 apply (cases rule: cdclNOT-restart.cases)
    apply (auto simp add: full1-def full-def dest: H dest: tranclpD)
    using H Suc-leI leD by blast
} note H = this
obtain A where bound-inv A (fst (g 1))
  using g[of 0] cdclNOT-inv-g[of 0] apply (cases rule: cdclNOT-restart.cases)
  apply (metis One-nat-def cdclNOT-inv exists-bound fst-conv relpowp-imp-rtranclp
    rtranclp-induct)
  using H[of 1] unfolding full1-def by (metis One-nat-def Suc-eq-plus1 diff-is-0-eq' diff-zero
    f-ge-1 fst-conv le-add2 relpowp-E2 snd-conv)
let ?j =  $\mu$ -bound A (fst (g 1)) + 1
obtain j where
  j:  $f (\text{snd } (g \ j)) > ?j$  and  $j > 1$ 
  using unbounded-f-g not-bounded-nat-exists-larger by blast
{
  fix i j
  have cdclNOT-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 rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl)
} note cdclNOT-restart = this
have cdclNOT-inv (fst (g (Suc 0)))
  by (simp add: cdclNOT-inv-g)
have cdclNOT-restart** (fst (g 1), snd (g 1)) (fst (g j), snd (g j))
  using <j> 1> by (simp add: cdclNOT-restart)
have  $\mu \ A \ (\text{fst } (g \ j)) \leq \mu\text{-bound } A \ (\text{fst } (g \ 1))$ 
  apply (rule rtranclp-cdclNOT-raw-restart-measure-bound)
  using <cdclNOT-restart** (fst (g 1), snd (g 1)) (fst (g j), snd (g j))> apply blast
  apply (simp add: cdclNOT-inv-g)
  using <bound-inv A (fst (g 1))> apply simp
done
then have  $\mu \ A \ (\text{fst } (g \ j)) \leq ?j$ 
  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))>
  rtranclp-cdclNOT-with-restart-bound-inv by auto
obtain T m where
  cdclNOT-m:  $(\text{cdcl}_{NOT} \rightsquigarrow m) (\text{fst } (g \ j)) T$  and
  f-m:  $f (\text{snd } (g \ j)) \leq m$ 
  using H[of j] by blast
have ?j < m
  using f-m j Nat.le-trans by linarith

then show False
  using < $\mu \ A \ (\text{fst } (g \ j)) \leq \mu\text{-bound } A \ (\text{fst } (g \ 1))$ >
  cdclNOT-comp-bounded[OF inv cdclNOT-inv-g, of ] cdclNOT-inv-g cdclNOT-m
  <?j < m> by auto
qed

```

lemma *cdcl_{NOT}-restart-steps-bigger-than-bound*:
assumes
cdcl_{NOT}-restart S T **and**
bound-inv A (*fst* S) **and**
cdcl_{NOT}-inv (*fst* S) **and**
 f (*snd* S) $>$ μ -*bound* A (*fst* S)
shows *full1 cdcl_{NOT}* (*fst* S) (*fst* T)
using *assms*
proof (*induction rule: cdcl_{NOT}-restart.induct*)
case *restart-full*
then show ?*case* **by** *auto*
next
case (*restart-step* m S T n U) **note** $st = \text{this}(1)$ **and** $f = \text{this}(2)$ **and** $\text{bound-inv} = \text{this}(4)$ **and**
 $\text{cdcl}_{NOT}\text{-inv} = \text{this}(5)$ **and** $\mu = \text{this}(6)$
then obtain m' **where** $m: m = \text{Suc } m'$ **by** (*cases* m) *auto*
have μ A $S - m' = 0$
using f *bound-inv* *cdcl_{NOT}-inv* μ m *rtrancp-cdcl_{NOT}-raw-restart-measure-bound* **by** *fastforce*
then have *False* **using** *cdcl_{NOT}-comp-n-le*[*of* m' S T A] *restart-step unfolding* m **by** *simp*
then show ?*case* **by** *fast*
qed

lemma *rtrancp-cdcl_{NOT}-with-inv-inv-rtrancp-cdcl_{NOT}*:
assumes
inv: cdcl_{NOT}-inv S **and**
binv: bound-inv A S
shows $(\lambda S T. \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT}\text{-inv } S \wedge \text{bound-inv } A S)^{**} S T \longleftrightarrow \text{cdcl}_{NOT}^{**} S T$
(is ? $A^{**} S T \longleftrightarrow ?B^{**} S T$ **)**
apply (*rule iffI*)
using *rtrancp-mono*[*of* ? A ? B] **apply** *blast*
apply (*induction rule: rtrancp-induct*)
using *inv binv apply simp*
by (*metis (mono-tags, lifting) binv inv rtrancp.simps rtrancp-cdcl_{NOT}-bound-inv rtrancp-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** $T: \text{cdcl}_{NOT} (\text{fst } S) T$
by *blast*
then obtain U **where** $U: \text{full } (\lambda S T. \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT}\text{-inv } S \wedge \text{bound-inv } A S) T U$
using *wf-exists-normal-form-full*[*OF* *wf-cdcl_{NOT}*, *of* A T] **by** *auto*
moreover have *inv-T: cdcl_{NOT}-inv* T
using $\langle \text{cdcl}_{NOT} (\text{fst } S) T \rangle \text{cdcl}_{NOT}\text{-inv inv}$ **by** *blast*
moreover have *b-inv-T: bound-inv* A T
using $\langle \text{cdcl}_{NOT} (\text{fst } S) T \rangle \text{binv bound-inv inv}$ **by** *blast*
ultimately have *full cdcl_{NOT}* T U
using *rtrancp-cdcl_{NOT}-with-inv-inv-rtrancp-cdcl_{NOT}* *rtrancp-cdcl_{NOT}-bound-inv rtrancp-cdcl_{NOT}-cdcl_{NOT}-inv* **unfolding** *full-def* **by** *blast*
then have *full1 cdcl_{NOT}* (*fst* S) U
using T *full-full1* **by** *metis*

```

    then show False by (metis n-s prod.collapse restart-full)
qed

end

```

5.2.6 Merging backjump and learning

```

locale cdclNOT-merge-bj-learn-ops =
  decide-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT +
  forget-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT forget-cond +
  propagate-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit) ann-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  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

```

We have a new backjump that combines the backjumping on the trail and the learning of the used clause (called C'' below)

```

inductive backjump-l where
backjump-l: trail S = F' @ Decided K # F
   $\Rightarrow$  no-dup (trail S)
   $\Rightarrow$   $T \sim \text{prepend-trail } (\text{Propagated } L \ ()) \ (\text{reduce-trail-to}_{NOT} \ F \ (\text{add-cl}_{NOT} \ C'' \ S))$ 
   $\Rightarrow$   $C \in \# \text{ clauses}_{NOT} \ S$ 
   $\Rightarrow$  trail S  $\models_{as} CNot \ C$ 
   $\Rightarrow$  undefined-lit F L
   $\Rightarrow$   $atm\text{-of } L \in atm\text{-of-mm } (\text{clauses}_{NOT} \ S) \cup atm\text{-of } ' \ (\text{lits-of-l } (\text{trail } S))$ 
   $\Rightarrow$   $\text{clauses}_{NOT} \ S \models_{pm} C' + \{\#L\# \}$ 
   $\Rightarrow$   $C'' = C' + \{\#L\# \}$ 
   $\Rightarrow$   $F \models_{as} CNot \ C'$ 
   $\Rightarrow$  backjump-l-cond C C' L S T
   $\Rightarrow$  backjump-l S T

```

Avoid (meaningless) simplification in the theorem generated by *inductive-cases*:

```

declare reduce-trail-toNOT-length-ne[simp del] Set.Un-iff[simp del] Set.insert-iff[simp del]
inductive-cases backjump-lE: backjump-l S T
thm backjump-lE
declare reduce-trail-toNOT-length-ne[simp] Set.Un-iff[simp] Set.insert-iff[simp]

```

```

inductive cdclNOT-merged-bj-learn :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where
cdclNOT-merged-bj-learn-decideNOT: decideNOT S S'  $\Rightarrow$  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 clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds forget-cond
   $\lambda C\ C'\ L'\ S\ T.$  backjump-l-cond  $C\ C'\ L'\ S\ T$ 
   $\wedge$  distinct-mset  $(C' + \{\#L'\#\}) \wedge \neg$ tautology  $(C' + \{\#L'\#\})$ 
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool and
  backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  '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' @ Decided K # F
   $\Rightarrow$  C  $\in$  # clausesNOT S
   $\Rightarrow$  trail S  $\models_{as}$  CNot C
   $\Rightarrow$  undefined-lit F L
   $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (F' @ Decided K # F))
   $\Rightarrow$  clausesNOT S  $\models_{pm}$  C' + {#L'#}
   $\Rightarrow$  F  $\models_{as}$  CNot 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 :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
where
  backjump-conds  $\equiv$   $\lambda C\ C'\ L'\ S\ T.$  distinct-mset  $(C' + \{\#L'\#\}) \wedge \neg$ tautology  $(C' + \{\#L'\#\})$ 

Without additional knowledge on backjump-l-cond, it is impossible to have the same invariant.

sublocale dpll-with-backjumping-ops trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds
proof (unfold-locales, goal-cases)
case 1
{ fix S S'
assume bj: backjump-l S S' and no-dup (trail S)
then obtain F' K F L C' C D where
  S': S'  $\sim$  prepend-trail (Propagated L ()) (reduce-trail-toNOT F (add-clNOT D S))
and
  tr-S: trail S = F' @ Decided K # F and
  C: C  $\in$  # clausesNOT S and
  tr-S-C: trail S  $\models_{as}$  CNot C and
  undef-L: undefined-lit F L and

```

```

atm-L:
  atm-of L ∈ insert (atm-of K) (atms-of-mm (clausesNOT S) ∪ atm-of ‘ (lits-of-l F' ∪ lits-of-l F))
  and
  cls-S-C': clausesNOT S ⊨pm C' + {#L#} and
  F-C': F ⊨as CNot C' and
  dist: distinct-mset (C' + {#L#}) and
  not-tauto: ¬ tautology (C' + {#L#}) and
  cond: backjump-l-cond C C' L S S'
  D = C' + {#L#}
  by (elim backjump-lE) metis
interpret backjumping-ops trail clausesNOT prepend-trail tl-trail add-clsNOT remove-clsNOT
backjump-conds
  by unfold-locales
have ∃ T. backjump S T
apply rule
apply (rule backjump.intros)
  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 tr-S apply simp
  using cls-S-C' apply simp
  using F-C' apply simp
  using dist not-tauto cond apply simp
done
}
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 clausesNOT prepend-trail tl-trail add-clsNOT remove-clsNOT
  propagate-conds forget-cond backjump-l-cond inv
for
  trail :: 'st ⇒ ('v, unit) ann-lits and
  clausesNOT :: 'st ⇒ 'v clauses and
  prepend-trail :: ('v, unit) ann-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-clsNOT :: 'v clause ⇒ 'st ⇒ 'st and
  remove-clsNOT :: 'v clause ⇒ 'st ⇒ 'st and
  propagate-conds :: ('v, unit) ann-lit ⇒ 'st ⇒ bool and
  forget-cond :: 'v clause ⇒ 'st ⇒ bool and
  backjump-l-cond :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
  inv :: 'st ⇒ bool
begin

sublocale conflict-driven-clause-learning-ops trail clausesNOT prepend-trail tl-trail add-clsNOT
  remove-clsNOT inv backjump-conds propagate-conds
λC -. distinct-mset C ∧ ¬tautology C
forget-cond
by unfold-locales
end

locale cdclNOT-merge-bj-learn =

```



```

cdclNOT-merge-bj-learn-proxy2 trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
propagate-conds forget-cond backjump-l-cond inv
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  propagate-conds :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  forget-cond :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool +
assumes
  dpll-merge-bj-inv:  $\bigwedge S T. \text{dp}_{ll}\text{-bj } S T \Rightarrow \text{inv } S \Rightarrow \text{inv } T$  and
  learn-inv:  $\bigwedge S T. \text{learn } S T \Rightarrow \text{inv } S \Rightarrow \text{inv } T$ 
begin

sublocale
  conflict-driven-clause-learning trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds
   $\lambda C -. \text{distinct-mset } C \wedge \neg \text{tautology } C$ 
  forget-cond
apply unfold-locales
using cdclNOT-merged-bj-learn-forgetNOT cdcl-merged-inv learn-inv
by (auto simp add: cdclNOT.simps dpll-merge-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  $\exists C' L D. \text{learn } S (\text{add-cl}_{NOT} D S)$ 
 $\wedge D = (C' + \{\#L\# \})$ 
 $\wedge \text{backjump } (\text{add-cl}_{NOT} D S) T$ 
 $\wedge \text{atms-of } (C' + \{\#L\# \}) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of } ' (\text{lits-of-l } (\text{trail } S))$ 
proof –
obtain C F' K F L l C' D where
  tr-S: trail S = F' @ Decided K # F and
  T: T  $\sim$  prepend-trail (Propagated L l) (reduce-trail-toNOT F (add-clNOT D S)) and
  C-clS: C  $\in \#$  clausesNOT 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-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S)) and
  clss-C: clausesNOT S  $\models_{pm}$  D and
  D: D = C' + {#L#}
  F  $\models_{as}$  CNot C' and
  distinct: distinct-mset D and
  not-tauto:  $\neg$  tautology D
using bt inv by (elim backjump-lE) simp
have atms-C': atms-of C'  $\subseteq$  atm-of ' (lits-of-l F)
by (metis D(2) atms-of-def image-subsetI true-annots-CNot-all-atms-defined)
then have atms-of (C' + {#L#})  $\subseteq$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))
using atm-L tr-S by auto
moreover have learn: learn S (add-clNOT D S)
apply (rule learn.intros)
apply (rule clss-C)
using atms-C' atm-L D apply (fastforce simp add: tr-S in-plus-implies-atm-of-on-atms-of-ms)
apply standard

```

apply (*rule distinct*)
apply (*rule not-tauto*)
apply *simp*
done
moreover have *bj*: *backjump* (*add-cls*_{NOT} *D S*) *T*
apply (*rule backjump.intros*)
using $\langle F \models_{as} CNot\ C' \rangle$ *C-cls-S tr-S-CNot-C undef T distinct not-tauto n-d D*
by (*auto simp*: *tr-S state-eq*_{NOT}-*def simp del*: *state-simp*_{NOT})
ultimately show *?thesis* **using** *D* **by** *blast*
qed

lemma *cdcl*_{NOT}-*merged-bj-learn-is-tranclp-cdcl*_{NOT}:
*cdcl*_{NOT}-*merged-bj-learn* *S T* \implies *inv S* \implies *no-dup* (*trail S*) \implies *cdcl*_{NOT}⁺⁺ *S T*

proof (*induction rule*: *cdcl*_{NOT}-*merged-bj-learn.induct*)

case (*cdcl*_{NOT}-*merged-bj-learn-decide*_{NOT} *T*)
then have *cdcl*_{NOT} *S T*
using *bj-decide*_{NOT} *cdcl*_{NOT}.*simps* **by** *fastforce*
then show *?case* **by** *auto*

next

case (*cdcl*_{NOT}-*merged-bj-learn-propagate*_{NOT} *T*)
then have *cdcl*_{NOT} *S T*
using *bj-propagate*_{NOT} *cdcl*_{NOT}.*simps* **by** *fastforce*
then show *?case* **by** *auto*

next

case (*cdcl*_{NOT}-*merged-bj-learn-forget*_{NOT} *T*)
then have *cdcl*_{NOT} *S T*
using *c-forget*_{NOT} **by** *blast*
then show *?case* **by** *auto*

next

case (*cdcl*_{NOT}-*merged-bj-learn-backjump-l* *T*) **note** *bt* = *this*(1) **and** *inv* = *this*(2) **and**
n-d = *this*(3)
obtain *C'* :: '*v* clause **and** *L* :: '*v* literal **and** *D* :: '*v* clause **where**
f3: *learn S* (*add-cls*_{NOT} *D S*) \wedge
backjump (*add-cls*_{NOT} *D S*) *T* \wedge
atms-of (*C'* + {*#L#*}) \subseteq *atms-of-mm* (*clauses*_{NOT} *S*) \cup *atm-of* '*lits-of-l* (*trail S*) **and**
D: *D* = *C'* + {*#L#*}
using *n-d backjump-l-learn-backjump*[*OF bt inv*] **by** *blast*
then have *f4*: *cdcl*_{NOT} *S* (*add-cls*_{NOT} *D S*)
using *n-d c-learn* **by** *blast*
have *cdcl*_{NOT} (*add-cls*_{NOT} *D S*) *T*
using *f3 n-d bj-backjump c-dpll-bj* **by** *blast*
then show *?case*
using *f4* **by** (*meson tranclp.r-into-trancl tranclp.trancl-into-trancl*)
qed

lemma *rtranclp-cdcl*_{NOT}-*merged-bj-learn-is-rtranclp-cdcl*_{NOT}-*and-inv*:

*cdcl*_{NOT}-*merged-bj-learn*^{**} *S T* \implies *inv S* \implies *no-dup* (*trail S*) \implies *cdcl*_{NOT}^{**} *S T* \wedge *inv T*

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

case *base*
then show *?case* **by** *auto*

next

case (*step T U*) **note** *st* = *this*(1) **and** *cdcl*_{NOT} = *this*(2) **and** *IH* = *this*(3)[*OF this*(4-)] **and**
inv = *this*(4) **and** *n-d* = *this*(5)
have *cdcl*_{NOT}^{**} *T U*
using *cdcl*_{NOT}-*merged-bj-learn-is-tranclp-cdcl*_{NOT}[*OF cdcl*_{NOT}] *IH*
*rtranclp-cdcl*_{NOT}-*no-dup inv n-d* **by** *auto*

then have $cdcl_{NOT}^{**} S U$ using *IH* by *fastforce*
 moreover have $inv U$ using $n-d$ *IH* $\langle cdcl_{NOT}^{**} T U \rangle$ *rtranclp-cdcl_{NOT}-inv* by *blast*
 ultimately show *?case* using *st* by *fast*
 qed

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}*:
 $cdcl_{NOT}\text{-merged-bj-learn}^{**} S T \implies inv S \implies no\text{-dup} (trail S) \implies cdcl_{NOT}^{**} S T$
 using *rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}-and-inv* by *blast*

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-inv*:
 $cdcl_{NOT}\text{-merged-bj-learn}^{**} S T \implies inv S \implies no\text{-dup} (trail S) \implies inv T$
 using *rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}-and-inv* by *blast*

definition $\mu_C' :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow nat$ **where**
 $\mu_C' A T \equiv \mu_C (1 + card (atms\text{-of}\text{-ms } A)) (2 + card (atms\text{-of}\text{-ms } A)) (trail\text{-weight } T)$

definition $\mu_{CDCL}'\text{-merged} :: 'v \text{ clause set} \Rightarrow 'st \Rightarrow nat$ **where**
 $\mu_{CDCL}'\text{-merged } A T \equiv ((2 + card (atms\text{-of}\text{-ms } A)) \wedge (1 + card (atms\text{-of}\text{-ms } A)) - \mu_C' A T) * 2 + card (set\text{-mset} (clauses_{NOT} T))$

lemma *cdcl_{NOT}-decreasing-measure'*:
assumes
cdcl_{NOT}-merged-bj-learn *S T* **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A* **and**
atm-trail: *atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A* **and**
n-d: *no-dup (trail S)* **and**
fin-A: *finite A*
shows $\mu_{CDCL}'\text{-merged } A T < \mu_{CDCL}'\text{-merged } A S$
using *assms(1)*

proof *induction*
case (*cdcl_{NOT}-merged-bj-learn-decide_{NOT} T*)
have *clauses_{NOT} S = clauses_{NOT} T*
using *cdcl_{NOT}-merged-bj-learn-decide_{NOT}.hyps* by *auto*
moreover have
 $(2 + card (atms\text{-of}\text{-ms } A)) \wedge (1 + card (atms\text{-of}\text{-ms } A))$
 $- \mu_C (1 + card (atms\text{-of}\text{-ms } A)) (2 + card (atms\text{-of}\text{-ms } A)) (trail\text{-weight } T)$
 $< (2 + card (atms\text{-of}\text{-ms } A)) \wedge (1 + card (atms\text{-of}\text{-ms } A))$
 $- \mu_C (1 + card (atms\text{-of}\text{-ms } A)) (2 + card (atms\text{-of}\text{-ms } A)) (trail\text{-weight } S)$
apply (*rule dpll-bj-trail-mes-decreasing-prop*)
using *cdcl_{NOT}-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 $\mu_{CDCL}'\text{-merged-def}$ $\mu_C'\text{-def}$ by *simp*

next
case (*cdcl_{NOT}-merged-bj-learn-propagate_{NOT} T*)
have *clauses_{NOT} S = clauses_{NOT} T*
using *cdcl_{NOT}-merged-bj-learn-propagate_{NOT}.hyps*
by (*simp add: bj-propagate_{NOT} inv dpll-bj-clauses*)
moreover have
 $(2 + card (atms\text{-of}\text{-ms } A)) \wedge (1 + card (atms\text{-of}\text{-ms } A))$
 $- \mu_C (1 + card (atms\text{-of}\text{-ms } A)) (2 + card (atms\text{-of}\text{-ms } A)) (trail\text{-weight } T)$
 $< (2 + card (atms\text{-of}\text{-ms } A)) \wedge (1 + card (atms\text{-of}\text{-ms } A))$
 $- \mu_C (1 + card (atms\text{-of}\text{-ms } A)) (2 + card (atms\text{-of}\text{-ms } A)) (trail\text{-weight } S)$
apply (*rule dpll-bj-trail-mes-decreasing-prop*)

```

    using inv n-d atm-clss atm-trail fin-A by (simp-all add: bj-propagateNOT
      cdclNOT-merged-bj-learn-propagateNOT.hyps)
  ultimately show ?case
    unfolding  $\mu_{CDCL}'$ -merged-def  $\mu_C'$ -def by simp
next
case (cdclNOT-merged-bj-learn-forgetNOT T)
have card (set-mset (clausesNOT T)) < card (set-mset (clausesNOT S))
  using ⟨forgetNOT S T⟩ by (metis card-Diff1-less clauses-remove-clNOT finite-set-mset
    forgetNOT.cases linear set-mset-minus-replicate-mset(1) state-eqNOT-def)
moreover
  have trail S = trail T
    using ⟨forgetNOT S T⟩ by (auto elim: forgetNOTE)
  then have
    ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
      -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight T))
    = ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
      -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight S))
    by auto
  ultimately show ?case
    unfolding  $\mu_{CDCL}'$ -merged-def  $\mu_C'$ -def by simp
next
case (cdclNOT-merged-bj-learn-backjump-l T) note bj-l = this(1)
obtain C' L D where
  learn: learn S (add-clNOT D S) and
  bj: backjump (add-clNOT D S) T and
  atms-C: atms-of (C' + {#L#})  $\subseteq$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S)) and
  D: D = C' + {#L#}
  using bj-l inv backjump-l-learn-backjump [of S] n-d atm-clss atm-trail by blast
have card-T-S: card (set-mset (clausesNOT T))  $\leq$  1 + card (set-mset (clausesNOT S))
  using bj-l inv by (force elim!: backjump-lE simp: card-insert-if)
have
  ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
    -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight T))
  < ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
    -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A))
      (trail-weight (add-clNOT D S)))
  apply (rule dpll-bj-trail-mes-decreasing-prop)
    using bj bj-backjump apply blast
    using cdclNOT.c-learn cdclNOT-inv inv learn apply blast
    using atms-C atm-clss atm-trail D apply (simp add: n-d) apply fast
    using atm-trail n-d apply simp
  apply (simp add: n-d)
  using fin-A apply simp
done
then have ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
  -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight T))
  < ((2 + card (atms-of-ms A)) ^ (1 + card (atms-of-ms A))
  -  $\mu_C$  (1 + card (atms-of-ms A)) (2 + card (atms-of-ms A)) (trail-weight S))
  using n-d by auto
then show ?case
  using card-T-S unfolding  $\mu_{CDCL}'$ -merged-def  $\mu_C'$ -def by linarith
qed

lemma wf-cdclNOT-merged-bj-learn:
  assumes
    fin-A: finite A

```

shows $wf \{(T, S).$
 $(inv\ S \wedge atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}$
apply (rule $wfP\text{-if-measure}[of\ -\ -\ \mu_{CDCL}'\text{-merged}\ A]$)
using $cdcl_{NOT}\text{-decreasing-measure}'\ fin\text{-}A$ **by** *simp*

lemma $trancpl\text{-}cdcl_{NOT}\text{-}cdcl_{NOT}\text{-}trancpl$:

assumes

$cdcl_{NOT}\text{-merged-bj-learn}^{++}\ S\ T$ **and**

inv : $inv\ S$ **and**

$atm\text{-}clss$: $atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A$ **and**

$atm\text{-}trail$: $atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$ **and**

$n\text{-}d$: $no-dup\ (trail\ S)$ **and**

$fin\text{-}A[simp]$: $finite\ A$

shows $(T, S) \in \{(T, S).$

$(inv\ S \wedge atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}^+ \text{ (is } - \in ?P^+)$

using *assms(1)*

proof (induction rule: *trancpl-induct*)

case *base*

then show *?case* **using** $n\text{-}d\ atm\text{-}clss\ atm\text{-}trail\ inv$ **by** *auto*

next

case (step $T\ U$) **note** $st = this(1)$ **and** $cdcl_{NOT} = this(2)$ **and** $IH = this(3)$

have $cdcl_{NOT}^{**}\ S\ T$

apply (rule $rtrancpl\text{-}cdcl_{NOT}\text{-merged-bj-learn-is-rtrancpl}\text{-}cdcl_{NOT}$)

using $st\ cdcl_{NOT}\ inv\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail\ inv$ **by** *auto*

have $inv\ T$

apply (rule $rtrancpl\text{-}cdcl_{NOT}\text{-merged-bj-learn}\text{-}inv$)

using $inv\ st\ cdcl_{NOT}\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail\ inv$ **by** *auto*

moreover have $atms-of-mm\ (clauses_{NOT}\ T) \subseteq atms-of-ms\ A$

using $rtrancpl\text{-}cdcl_{NOT}\text{-trail-clauses-bound}[OF\ \langle cdcl_{NOT}^{**}\ S\ T \rangle\ inv\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail]$

by *fast*

moreover have $atm-of\ 'lits-of-l\ (trail\ T) \subseteq atms-of-ms\ A$

using $rtrancpl\text{-}cdcl_{NOT}\text{-trail-clauses-bound}[OF\ \langle cdcl_{NOT}^{**}\ S\ T \rangle\ inv\ n\text{-}d\ atm\text{-}clss\ atm\text{-}trail]$

by *fast*

moreover have $no-dup\ (trail\ T)$

using $rtrancpl\text{-}cdcl_{NOT}\text{-no-dup}[OF\ \langle cdcl_{NOT}^{**}\ S\ T \rangle\ inv\ n\text{-}d]$ **by** *fast*

ultimately have $(U, T) \in ?P$

using $cdcl_{NOT}$ **by** *auto*

then show *?case* **using** IH **by** (*simp add: trancpl-into-trancpl2*)

qed

lemma $wf\text{-}trancpl\text{-}cdcl_{NOT}\text{-merged-bj-learn}$:

assumes $finite\ A$

shows $wf \{(T, S).$

$(inv\ S \wedge atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}^{++}\ S\ T\}$

apply (rule *wf-subset*)

apply (rule $wf\text{-}trancpl[OF\ wf\text{-}cdcl_{NOT}\text{-merged-bj-learn}]$)

using *assms* **apply** *simp*

using $trancpl\text{-}cdcl_{NOT}\text{-}cdcl_{NOT}\text{-}trancpl[OF\ -\ -\ -\ -\ \langle finite\ A \rangle]$ **by** *auto*

lemma $backjump\text{-}no\text{-}step\text{-}backjump\text{-}l$:

$\text{backjump } S \ T \implies \text{inv } S \implies \neg \text{no-step backjump-l } S$
apply (elim backjumpE)
apply (rule bj-merge-can-jump)
apply auto[7]
by blast

lemma $\text{cdcl}_{\text{NOT-merged-bj-learn-final-state}}$:

fixes $A :: 'v \text{ clause set}$ **and** $S \ T :: 'st$

assumes

$n\text{-s}$: $\text{no-step cdcl}_{\text{NOT-merged-bj-learn}} \ S$ **and**

atms-S : $\text{atms-of-mm } (\text{clauses}_{\text{NOT}} \ S) \subseteq \text{atms-of-ms } A$ **and**

atms-trail : $\text{atm-of } ' \text{ lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**

$n\text{-d}$: $\text{no-dup } (\text{trail } S)$ **and**

$\text{finite } A$ **and**

inv : $\text{inv } S$ **and**

decomp : $\text{all-decomposition-implies-m } (\text{clauses}_{\text{NOT}} \ S) \ (\text{get-all-ann-decomposition } (\text{trail } S))$

shows $\text{unsatisfiable } (\text{set-mset } (\text{clauses}_{\text{NOT}} \ S))$

$\vee (\text{trail } S \models_{\text{asm}} \text{clauses}_{\text{NOT}} \ S \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{\text{NOT}} \ S)))$

proof –

let $?N = \text{set-mset } (\text{clauses}_{\text{NOT}} \ S)$

let $?M = \text{trail } S$

consider

$(\text{sat}) \text{ satisfiable } ?N$ **and** $?M \models_{\text{as}} ?N$

| $(\text{sat}') \text{ satisfiable } ?N$ **and** $\neg ?M \models_{\text{as}} ?N$

| $(\text{unsat}) \text{ unsatisfiable } ?N$

by auto

then show $?thesis$

proof cases

case sat' **note** $\text{sat} = \text{this}(1)$ **and** $M = \text{this}(2)$

obtain C **where** $C \in ?N$ **and** $\neg ?M \models_a C$ **using** M **unfolding** true-annots-def **by** auto

obtain $I :: 'v \text{ literal set}$ **where**

$I \models_s ?N$ **and**

cons : $\text{consistent-interp } I$ **and**

tot : $\text{total-over-m } I \ ?N$ **and**

atm-I-N : $\text{atm-of } 'I \subseteq \text{atms-of-ms } ?N$

using sat **unfolding** $\text{satisfiable-def-min}$ **by** auto

let $?I = I \cup \{P \mid P. P \in \text{lits-of-l } ?M \wedge \text{atm-of } P \notin \text{atm-of } 'I\}$

let $?O = \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } ?M \wedge \text{atm-of } (\text{lit-of } L) \notin \text{atms-of-ms } ?N\}$

have cons-I' : $\text{consistent-interp } ?I$

using cons **using** $\langle \text{no-dup } ?M \rangle$ **unfolding** $\text{consistent-interp-def}$

by ($\text{auto simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def}$

$\text{dest!:: no-dup-cannot-not-lit-and-uminus}$)

have tot-I' : $\text{total-over-m } ?I \ (\ ?N \cup \text{unmark-l } ?M)$

using tot atms-of-s-def **unfolding** $\text{total-over-m-def total-over-set-def}$

by ($\text{fastforce simp: image-iff}$)

have $\{P \mid P. P \in \text{lits-of-l } ?M \wedge \text{atm-of } P \notin \text{atm-of } 'I\} \models_s ?O$

using $\langle I \models_s ?N \rangle$ atm-I-N **by** ($\text{auto simp add: atm-of-eq-atm-of true-clss-def lits-of-def}$)

then have $I'\text{-N}$: $?I \models_s ?N \cup ?O$

using $\langle I \models_s ?N \rangle$ $\text{true-clss-union-increase}$ **by** force

have tot' : $\text{total-over-m } ?I \ (\ ?N \cup ?O)$

using atm-I-N tot **unfolding** $\text{total-over-m-def total-over-set-def}$

by ($\text{force simp: lits-of-def elim!:: is-decided-ex-Decided}$)

have atms-N-M : $\text{atms-of-ms } ?N \subseteq \text{atm-of } ' \text{ lits-of-l } ?M$

proof (rule ccontr)

assume $\neg ?thesis$

```

then obtain  $l :: 'v$  where
   $l-N: l \in \text{atms-of-ms } ?N$  and
   $l-M: l \notin \text{atm-of } ' \text{ lits-of-l } ?M$ 
  by auto
have undefined-lit  $?M$  (Pos  $l$ )
  using  $l-M$  by (metis Decided-Propagated-in-iff-in-lits-of-l
    atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set literal.sel(1))
have decideNOT  $S$  (prepend-trail (Decided (Pos  $l$ ))  $S$ )
  by (metis (undefined-lit  $?M$  (Pos  $l$ )) decideNOT.intros  $l-N$  literal.sel(1)
    state-eqNOT-ref)
then show False
  using cdclNOT-merged-bj-learn-decideNOT  $n-s$  by blast
qed

have  $?M \models_{as} CNot\ C$ 
apply (rule all-variables-defined-not-imply-cnot)
  using atms- $N-M$   $\langle C \in ?N \rangle \hookrightarrow ?M \models_a C$  atms-of-atms-of-ms-mono[OF  $\langle C \in ?N \rangle$ ]
  by (auto dest: atms-of-atms-of-ms-mono)
have  $\exists l \in \text{set } ?M. \text{is-decided } l$ 
proof (rule ccontr)
  let  $?O = \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } ?M \wedge \text{atm-of } (\text{lit-of } L) \notin \text{atms-of-ms } ?N\}$ 
  have  $\vartheta[\text{iff}]: \bigwedge I. \text{total-over-m } I \ (\ ?N \cup ?O \cup \text{unmark-l } ?M)$ 
     $\longleftrightarrow \text{total-over-m } I \ (\ ?N \cup \text{unmark-l } ?M)$ 
    unfolding total-over-set-def total-over-m-def atms-of-ms-def by blast
  assume  $\neg ?thesis$ 
  then have [simp]:  $\{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } ?M\}$ 
     $= \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } ?M \wedge \text{atm-of } (\text{lit-of } L) \notin \text{atms-of-ms } ?N\}$ 
    by auto
  then have  $?N \cup ?O \models_{ps} \text{unmark-l } ?M$ 
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] by auto

  then have  $?I \models_s \text{unmark-l } ?M$ 
    using cons- $I'$   $I'-N$  tot- $I'$   $\langle ?I \models_s ?N \cup ?O \rangle$  unfolding  $\vartheta$  true-clss-clss-def by blast
  then have lits-of-l  $?M \subseteq ?I$ 
    unfolding true-clss-def lits-of-def by auto
  then have  $?M \models_{as} ?N$ 
    using  $I'-N$   $\langle C \in ?N \rangle \hookrightarrow ?M \models_a C$  cons- $I'$  atms- $N-M$ 
    by (meson (trail  $S \models_{as} CNot\ C$ ) consistent-CNot-not rev-subsetD sup-ge1 true-annot-def
      true-annots-def true-clss-mono-set-mset-l true-clss-def)
  then show False using  $M$  by fast
qed

from List.split-list-first-propE[OF this] obtain  $K :: 'v$  literal and  $d :: \text{unit}$  and
   $F\ F' :: ('v, \text{unit}) \text{ann-lits}$  where
   $M-K: ?M = F' @ \text{Decided } K \# F$  and
   $nm: \forall f \in \text{set } F'. \neg \text{is-decided } f$ 
  unfolding is-decided-def by (metis (full-types) old.unit.exhaust)
let  $?K = \text{Decided } K :: ('v, \text{unit}) \text{ann-lit}$ 
have  $?K \in \text{set } ?M$ 
  unfolding  $M-K$  by auto
let  $?C = \text{image-mset lit-of } \{\#L \in \#mset\ ?M. \text{is-decided } L \wedge L \neq ?K\# \} :: 'v \text{ clause}$ 
let  $?C' = \text{set-mset } (\text{image-mset } (\lambda L :: 'v \text{ literal}. \{\#L\# \}) \ (\ ?C + \text{unmark } ?K))$ 
have  $?N \cup \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } ?M\} \models_{ps} \text{unmark-l } ?M$ 
  using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
moreover have  $C': ?C' = \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } ?M\}$ 
  unfolding  $M-K$  apply standard
  apply force

```

```

  by auto
ultimately have N-C-M: ?N  $\cup$  ?C'  $\models_{ps}$  unmark-l ?M
  by auto
have N-M-False: ?N  $\cup$  ( $\lambda L.$  unmark L) ' (set ?M)  $\models_{ps}$  {{#}}
  using M  $\langle$  ?M  $\models_{as}$  CNot C  $\rangle$   $\langle$  C  $\in$  ?N  $\rangle$  unfolding true-clss-clss-def true-annots-def Ball-def
  true-annot-def by (metis consistent-CNot-not sup.orderE sup-commute true-clss-def
    true-clss-singleton-lit-of-implies-incl true-clss-union true-clss-union-increase)

have undefined-lit F K using  $\langle$  no-dup ?M  $\rangle$  unfolding M-K by (simp add: defined-lit-map)
moreover
  have ?N  $\cup$  ?C'  $\models_{ps}$  {{#}}
  proof -
    have A: ?N  $\cup$  ?C'  $\cup$  unmark-l ?M = ?N  $\cup$  unmark-l ?M
      unfolding M-K by auto
    show ?thesis
      using true-clss-clss-left-right[OF N-C-M, of {{#}}] N-M-False unfolding A by auto
  qed
have ?N  $\models_p$  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  $\{a \in \text{set } (\text{trail } S). \text{is-decided } a \wedge a \neq \text{Decided } K\} =$ 
       $\text{set } (\text{trail } S) \cap \{L. \text{is-decided } L \wedge L \neq \text{Decided } K\}$ 
      by auto
    then have tot': total-over-set I
      (atm-of ' lit-of ' (set ?M  $\cap$  {L. is-decided L  $\wedge$  L  $\neq$  Decided K}))
      using tot by (auto simp add: atms-of-uminus-lit-atm-of-lit-of)
    { fix x :: ('v, unit) ann-lit
      assume
        a3: lit-of x  $\notin$  I and
        a1: x  $\in$  set ?M and
        a4: is-decided x and
        a5: x  $\neq$  Decided 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-clss-def Bex-def
    using  $\langle$   $(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$   $\rangle$ 
    by (auto dest!: H)

```



```

    qed
  moreover have  $F \models_{as} CNot \ (image-mset \ minus \ ?C)$ 
    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 minus (image-mset lit-of {# L :# mset ?M. is-decided L  $\wedge$  L  $\neq$  Decided 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 clause set and S T :: 'st
  assumes
    full: full cdclNOT-merged-bj-learn S T and
    atms-S: atms-of-mm (clausesNOT S)  $\subseteq$  atms-of-ms A and
    atms-trail: atm-of ' lits-of-l (trail S)  $\subseteq$  atms-of-ms A and
    n-d: no-dup (trail S) and
    finite A and
    inv: inv S and
    decomp: all-decomposition-implies-m (clausesNOT S) (get-all-ann-decomposition (trail S))
  shows unsatisfiable (set-mset (clausesNOT T))
     $\vee (trail \ T \models_{asm} clauses_{NOT} \ T \wedge \text{satisfiable} \ (set-mset \ (clauses_{NOT} \ T)))$ 
proof -
  have st: cdclNOT-merged-bj-learn** S T and n-s: no-step cdclNOT-merged-bj-learn T
    using full unfolding full-def by blast+
  then have st: cdclNOT** S T
    using inv rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT-and-inv n-d by auto
  have atms-of-mm (clausesNOT T)  $\subseteq$  atms-of-ms A and atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A
    using rtranclp-cdclNOT-trail-clauses-bound[OF st inv n-d atms-S atms-trail] by blast+
  moreover have no-dup (trail T)
    using rtranclp-cdclNOT-no-dup inv n-d st by blast
  moreover have inv T
    using rtranclp-cdclNOT-inv inv st by blast
  moreover have all-decomposition-implies-m (clausesNOT T) (get-all-ann-decomposition (trail T))
    using rtranclp-cdclNOT-all-decomposition-implies inv st decomp n-d by blast
  ultimately show ?thesis
    using cdclNOT-merged-bj-learn-final-state[of T A] (finite A) n-s by fast
qed

end

```

5.2.7 Instantiations

In this section, we instantiate the previous locales to ensure that the assumption are not contradictory.

```

locale cdclNOT-with-backtrack-and-restarts =
  conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt
  trail clausesNOT prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds learn-restrictions forget-restrictions
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and

```

```

tl-trail :: 'st ⇒ 'st and
add-clsNOT :: 'v clause ⇒ 'st ⇒ 'st and
remove-clsNOT :: 'v clause ⇒ 'st ⇒ 'st and
inv :: 'st ⇒ bool and
backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
propagate-conds :: ('v, unit) ann-lit ⇒ 'st ⇒ bool and
learn-restrictions forget-restrictions :: 'v clause ⇒ 'st ⇒ bool
+
fixes f :: nat ⇒ nat
assumes
  unbounded: unbounded f and f-ge-1:  $\bigwedge n. n \geq 1 \implies f\ n \geq 1$  and
  inv-restart:  $\bigwedge S\ T. \text{inv } S \implies T \sim \text{reduce-trail-to}_{NOT} ([::'a\ list)\ S \implies \text{inv } T$ 
begin

lemma bound-inv-inv:
  assumes
    inv S and
    n-d: no-dup (trail S) and
    atms-clss-S-A: atms-of-mm (clausesNOT S) ⊆ atms-of-ms A and
    atms-trail-S-A: atm-of ' lits-of-l (trail S) ⊆ atms-of-ms A and
    finite A and
    cdclNOT: cdclNOT S T
  shows
    atms-of-mm (clausesNOT T) ⊆ atms-of-ms A and
    atm-of ' lits-of-l (trail T) ⊆ atms-of-ms A and
    finite A
proof -
  have cdclNOT S T
  using ⟨inv S⟩ cdclNOT by linarith
  then have atms-of-mm (clausesNOT T) ⊆ atms-of-mm (clausesNOT S) ∪ atm-of ' lits-of-l (trail S)
  using ⟨inv S⟩
  by (meson conflict-driven-clause-learning-ops.cdclNOT-atms-of-ms-clauses-decreasing
      conflict-driven-clause-learning-ops-axioms n-d)
  then show atms-of-mm (clausesNOT T) ⊆ atms-of-ms A
  using atms-clss-S-A atms-trail-S-A by blast
next
  show atm-of ' lits-of-l (trail T) ⊆ atms-of-ms A
  by (meson ⟨inv S⟩ atms-clss-S-A atms-trail-S-A cdclNOT cdclNOT-atms-in-trail-in-set n-d)
next
  show finite A
  using ⟨finite A⟩ by simp
qed

sublocale cdclNOT-increasing-restarts-ops λS T. T ∼ reduce-trail-toNOT ([::'a list) S cdclNOT f
  λA S. atms-of-mm (clausesNOT S) ⊆ atms-of-ms A ∧ atm-of ' lits-of-l (trail S) ⊆ atms-of-ms A ∧
  finite A
  μCDCL' λS. inv S ∧ no-dup (trail S)
  μCDCL'-bound
  apply unfold-locales
    apply (simp add: unbounded)
    using f-ge-1 apply force
    using bound-inv-inv apply meson
    apply (rule cdclNOT-decreasing-measure'; simp)
    apply (rule rtranclp-cdclNOT-μCDCL'-bound; simp)
    apply (rule rtranclp-μCDCL'-bound-decreasing; simp)
    apply auto[]

```

```

    apply auto[]
    using cdclNOT-inv cdclNOT-no-dup apply blast
    using inv-restart apply auto[]
done

lemma cdclNOT-with-restart- $\mu_{CDCL}'$ -le- $\mu_{CDCL}'$ -bound:
  assumes
    cdclNOT: cdclNOT-restart (T, a) (V, b) and
    cdclNOT-inv:
      inv T
      no-dup (trail T) and
    bound-inv:
      atms-of-mm (clausesNOT T)  $\subseteq$  atms-of-ms A
      atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A
      finite A
  shows  $\mu_{CDCL}' A V \leq \mu_{CDCL}'$ -bound A T
  using cdclNOT-inv bound-inv
proof (induction rule: cdclNOT-with-restart-induct[OF cdclNOT])
  case (1 m S T n U) note U = this(3)
  show ?case
    apply (rule rtrancplp-cdclNOT- $\mu_{CDCL}'$ -bound-reduce-trail-toNOT[of S T])
      using  $\langle (cdcl_{NOT} \rightsquigarrow m) S T \rangle$  apply (fastforce dest!: relpowp-imp-rtrancplp)
      using 1 by auto
  next
  case (2 S T n) note full = this(2)
  show ?case
    apply (rule rtrancplp-cdclNOT- $\mu_{CDCL}'$ -bound)
      using full 2 unfolding full1-def by force+
qed

lemma cdclNOT-with-restart- $\mu_{CDCL}'$ -bound-le- $\mu_{CDCL}'$ -bound:
  assumes
    cdclNOT: cdclNOT-restart (T, a) (V, b) and
    cdclNOT-inv:
      inv T
      no-dup (trail T) and
    bound-inv:
      atms-of-mm (clausesNOT T)  $\subseteq$  atms-of-ms A
      atm-of ' lits-of-l (trail T)  $\subseteq$  atms-of-ms A
      finite A
  shows  $\mu_{CDCL}'$ -bound A V  $\leq \mu_{CDCL}'$ -bound A T
  using cdclNOT-inv bound-inv
proof (induction rule: cdclNOT-with-restart-induct[OF cdclNOT])
  case (1 m S T n U) note U = this(3)
  have  $\mu_{CDCL}'$ -bound A T  $\leq \mu_{CDCL}'$ -bound A S
    apply (rule rtrancplp- $\mu_{CDCL}'$ -bound-decreasing)
      using  $\langle (cdcl_{NOT} \rightsquigarrow m) S T \rangle$  apply (fastforce dest!: relpowp-imp-rtrancplp)
      using 1 by auto
  then show ?case using U unfolding  $\mu_{CDCL}'$ -bound-def by auto
  next
  case (2 S T n) note full = this(2)
  show ?case
    apply (rule rtrancplp- $\mu_{CDCL}'$ -bound-decreasing)
      using full 2 unfolding full1-def by force+
qed

```

sublocale *cdcl_{NOT}-increasing-restarts* - - - - -
 f
 $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([\]::'a \text{ list}) S$
 $\lambda A S. \text{atms-of-mm} (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}' \text{cdcl}_{NOT}$
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
apply *unfold-locales*
using *cdcl_{NOT}-with-restart- $\mu_{CDCL}'\text{-le-}\mu_{CDCL}'\text{-bound}$* **apply** *simp*
using *cdcl_{NOT}-with-restart- $\mu_{CDCL}'\text{-bound-le-}\mu_{CDCL}'\text{-bound}$* **apply** *simp*
done

lemma *cdcl_{NOT}-restart-all-decomposition-implies*:
assumes *cdcl_{NOT}-restart* $S T$ **and**
 $\text{inv } (\text{fst } S)$ **and**
 $\text{no-dup } (\text{trail } (\text{fst } S))$
 $\text{all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } S)) (\text{get-all-ann-decomposition } (\text{trail } (\text{fst } S)))$
shows
 $\text{all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } T)) (\text{get-all-ann-decomposition } (\text{trail } (\text{fst } T)))$
using *assms* **apply** (*induction*)
using *rtranclp-cdcl_{NOT}-all-decomposition-implies* **by** (*auto dest!: tranclp-into-rtranclp simp: full1-def*)

lemma *rtranclp-cdcl_{NOT}-restart-all-decomposition-implies*:
assumes *cdcl_{NOT}-restart*** $S T$ **and**
 $\text{inv: inv } (\text{fst } S)$ **and**
 $\text{n-d: no-dup } (\text{trail } (\text{fst } S))$ **and**
 $\text{decomp: all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } S)) (\text{get-all-ann-decomposition } (\text{trail } (\text{fst } S)))$
shows
 $\text{all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } T)) (\text{get-all-ann-decomposition } (\text{trail } (\text{fst } T)))$
using *assms(1)*
proof (*induction rule: rtranclp-induct*)
case *base*
then show *?case* **using** *decomp* **by** *simp*
next
case (*step* $T u$) **note** $st = \text{this}(1)$ **and** $r = \text{this}(2)$ **and** $IH = \text{this}(3)$
have $\text{inv } (\text{fst } T)$
using *rtranclp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv[OF st] inv n-d* **by** *blast*
moreover have $\text{no-dup } (\text{trail } (\text{fst } T))$
using *rtranclp-cdcl_{NOT}-with-restart-cdcl_{NOT}-inv[OF st] inv n-d* **by** *blast*
ultimately show *?case*
using *cdcl_{NOT}-restart-all-decomposition-implies* $r IH n-d$ **by** *fast*
qed

lemma *cdcl_{NOT}-restart-sat-ext-iff*:
assumes
 $st: \text{cdcl}_{NOT}\text{-restart } S T$ **and**
 $n-d: \text{no-dup } (\text{trail } (\text{fst } S))$ **and**
 $inv: \text{inv } (\text{fst } S)$
shows $I \models_{\text{sextm}} \text{clauses}_{NOT} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} (\text{fst } T)$
using *assms*
proof (*induction*)
case (*restart-step* $m S T n U$)
then show *?case*

```

    using rtrancpl-cdclNOT-bj-sat-ext-iff n-d by (fastforce dest!: relpoup-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:
fixes S T :: 'st × nat
assumes
  st: cdclNOT-restart** S T and
  n-d: no-dup (trail (fst S)) and
  inv: inv (fst S)
shows I ⊨sextm clausesNOT (fst S) ⟷ I ⊨sextm clausesNOT (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 clause set and S T :: 'st
assumes
  full: full cdclNOT-restart (S, n) (T, m) and
  atms-S: atms-of-mm (clausesNOT S) ⊆ atms-of-ms A and
  atms-trail: atm-of ' lits-of-l (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 (clausesNOT S) (get-all-ann-decomposition (trail S))
shows unsatisfiable (set-mset (clausesNOT S))
  ∨ (lits-of-l (trail T) ⊨sextm clausesNOT S ∧ satisfiable (set-mset (clausesNOT S)))
proof -
have st: cdclNOT-restart** (S, n) (T, m) and
  n-s: no-step cdclNOT-restart (T, m)
  using full unfolding full-def by fast+
have binv-T: atms-of-mm (clausesNOT T) ⊆ atms-of-ms A
  atm-of ' lits-of-l (trail T) ⊆ atms-of-ms A
  using rtrancpl-cdclNOT-with-restart-bound-inv[OF st, of A] inv n-d atms-S atms-trail
  by auto
moreover have inv-T: no-dup (trail T) inv T
  using rtrancpl-cdclNOT-with-restart-cdclNOT-inv[OF st] inv n-d by auto
moreover have all-decomposition-implies-m (clausesNOT T) (get-all-ann-decomposition (trail T))
  using rtrancpl-cdclNOT-restart-all-decomposition-implies[OF st] inv n-d
  decomp by auto
ultimately have T: unsatisfiable (set-mset (clausesNOT T))
  ∨ (trail T ⊨asm clausesNOT T ∧ satisfiable (set-mset (clausesNOT T)))
  using no-step-cdclNOT-restart-no-step-cdclNOT[of (T, m) A] n-s

```

```

  cdclNOT-final-state[of  $T$   $A$ ] unfolding cdclNOT-NOT-all-inv-def by auto
have eq-sat- $S$ - $T$ :  $\bigwedge I. I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T$ 
  using rtrancpl-cdclNOT-restart-sat-ext-iff[ $OF$   $st$ ] inv  $n$ -d  $\text{atms}$ - $S$ 
     $\text{atms}$ -trail by auto
have cons- $T$ : consistent-interp (lits-of-l (trail  $T$ ))
  using inv- $T$ (1) distinct-consistent-interp by blast
consider
  (unsat) unsatisfiable (set-mset (clausesNOT  $T$ ))
  | (sat) trail  $T \models_{\text{asm}} \text{clauses}_{\text{NOT}} T$  and satisfiable (set-mset (clausesNOT  $T$ ))
  using  $T$  by blast
then show ?thesis
proof cases
  case unsat
  then have unsatisfiable (set-mset (clausesNOT  $S$ ))
    using eq-sat- $S$ - $T$  consistent-true-clss-ext-satisfiable true-clss-imp-true-clss-ext
    unfolding satisfiable-def by blast
  then show ?thesis by fast
next
  case sat
  then have lits-of-l (trail  $T$ )  $\models_{\text{sextm}} \text{clauses}_{\text{NOT}} S$ 
    using rtrancpl-cdclNOT-restart-sat-ext-iff[ $OF$   $st$ ] inv  $n$ -d  $\text{atms}$ - $S$ 
     $\text{atms}$ -trail by (auto simp: true-clss-imp-true-clss-ext true-annots-true-clss)
  moreover then have satisfiable (set-mset (clausesNOT  $S$ ))
    using cons- $T$  consistent-true-clss-ext-satisfiable by blast
  ultimately show ?thesis by blast
qed
qed
end — end of cdclNOT-with-backtrack-and-restarts locale

```

The restart does only reset the trail, contrary to Weidenbach's version where forget and restart are always combined. But there is a forget rule.

```

locale cdclNOT-merge-bj-learn-with-backtrack-restarts =
  cdclNOT-merge-bj-learn trail clausesNOT prepend-trail tl-trail add-clssNOT remove-clssNOT
   $\lambda C C' L' S T. \text{distinct-mset } (C' + \{\#L'\# \}) \wedge \text{backjump-l-cond } C C' L' S T$ 
  propagate-conds forget-conds inv
for
  trail :: 'st  $\Rightarrow$  ('v, unit) ann-lits and
  clausesNOT :: 'st  $\Rightarrow$  'v clauses and
  prepend-trail :: ('v, unit) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clssNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clssNOT :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  propagate-conds :: ('v, unit) ann-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  $\Rightarrow$  '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:  $\bigwedge S T. \text{inv } S \Rightarrow T \sim \text{reduce-trail-to}_{\text{NOT}} \ \square \ S \Rightarrow \text{inv } T$ 
begin

```

definition not-simplified-clss :: 'b literal multiset multiset \Rightarrow 'b literal multiset multiset
where
 not-simplified-clss $A \equiv \{\#C \in \# A. C \notin \text{simple-clss } (\text{atms-of-mm } A)\#\}$

lemma *not-simplified-cls-tautology-distinct-mset*:

not-simplified-cls $A = \{\#C \in \# A. \text{tautology } C \vee \neg \text{distinct-mset } C\# \}$

unfolding *not-simplified-cls-def* **by** (rule *filter-mset-cong*) (auto *simp*: *simple-clss-def*)

lemma *simple-clss-or-not-simplified-cls*:

assumes *atms-of-mm* (*clauses*_{NOT} S) \subseteq *atms-of-ms* A **and**

$x \in \# \text{clauses}_{NOT} S$ **and** *finite* A

shows $x \in \text{simple-clss} (\text{atms-of-ms } A) \vee x \in \# \text{not-simplified-cls} (\text{clauses}_{NOT} S)$

proof –

consider

(*simpl*) $\neg \text{tautology } x$ **and** *distinct-mset* x

| (*n-simp*) *tautology* $x \vee \neg \text{distinct-mset } x$

by *auto*

then show ?thesis

proof *cases*

case *simpl*

then have $x \in \text{simple-clss} (\text{atms-of-ms } A)$

by (*meson* *assms* *atms-of-atms-of-ms-mono* *atms-of-ms-finite* *simple-clss-mono*
distinct-mset-not-tautology-implies-in-simple-clss *finite-subset*
subsetCE)

then show ?thesis **by** *blast*

next

case *n-simp*

then have $x \in \# \text{not-simplified-cls} (\text{clauses}_{NOT} S)$

using $\langle x \in \# \text{clauses}_{NOT} S \rangle$ **unfolding** *not-simplified-cls-tautology-distinct-mset* **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-mm* (*clauses*_{NOT} S) \subseteq *atms-of-ms* A **and**

atms-trail: *atm-of* (*lits-of-l* (*trail* S)) \subseteq *atms-of-ms* A **and**

n-d: *no-dup* (*trail* S) **and**

fin-A[*simp*]: *finite* A

shows *set-mset* (*clauses*_{NOT} T) \subseteq *set-mset* (*not-simplified-cls* (*clauses*_{NOT} S))

\cup *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-cls*)

next

case *cdcl_{NOT}-merged-bj-learn-propagate_{NOT}*

then show ?case **using** *dpll-bj-clauses* **by** (*force* *dest*!: *simple-clss-or-not-simplified-cls*)

next

case *cdcl_{NOT}-merged-bj-learn-forget_{NOT}*

then show ?case **using** *clauses-remove-cls_{NOT}* **unfolding** *state-eq_{NOT}-def*

by (*force* *elim*!: *forget_{NOT}E* *dest*: *simple-clss-or-not-simplified-cls*)

next

case (*cdcl_{NOT}-merged-bj-learn-backjump-l* T) **note** $\text{bj} = \text{this}(1)$ **and** $\text{inv} = \text{this}(2)$ **and**
 $\text{atms-clss} = \text{this}(3)$ **and** $\text{atms-trail} = \text{this}(4)$ **and** $\text{n-d} = \text{this}(5)$

have *cdcl_{NOT}*** S T

apply (rule rtrancpl-cdcl_{NOT}-merged-bj-learn-is-rtrancpl-cdcl_{NOT})
using bj inv cdcl_{NOT}-merged-bj-learn.simps n-d **by** blast+
have atm-of ‘(lits-of-l (trail T)) \subseteq atms-of-ms A
using 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-mm (clauses_{NOT} T) \subseteq atms-of-ms A
using 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 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 D **where**
tr-S: trail S = F' @ Decided K # F **and**
T: T \sim prepend-trail (Propagated L l) (reduce-trail-to_{NOT} F (add-cl_{NOT} D S)) **and**
C \in # clauses_{NOT} S **and**
trail S \models_{as} CNot C **and**
undef: undefined-lit F L **and**
clauses_{NOT} S \models_{pm} C' + {#L#} **and**
F \models_{as} CNot C' **and**
D: D = C' + {#L#} **and**
dist: distinct-mset (C' + {#L#}) **and**
tauto: \neg tautology (C' + {#L#}) **and**
backjump-l-cond C C' L S T
using \langle backjump-l S T \rangle **apply** (elim backjump-lE) **by** auto

have atms-of C' \subseteq atm-of ‘(lits-of-l F)
using \langle F \models_{as} CNot C' \rangle **by** (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
atms-of-def image-subset-iff in-CNot-implies-uminus(2))
then have atms-of (C' + {#L#}) \subseteq atms-of-ms A
using T \langle atm-of ‘lits-of-l (trail T) \subseteq atms-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 D **by** (force dest!: simple-clss-or-not-simplified-clss)
qed

lemma cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing:
assumes cdcl_{NOT}-merged-bj-learn S T
shows not-simplified-cl_s (clauses_{NOT} T) \subseteq # not-simplified-cl_s (clauses_{NOT} S)
using assms **apply** induction
prefer 4
unfolding not-simplified-cl_s-tautology-distinct-mset **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 not-simplified-cl_s (clauses_{NOT} T) \subseteq # not-simplified-cl_s (clauses_{NOT} S)
using assms **apply** induction
apply simp
by (drule cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing) auto

lemma rtrancpl-cdcl_{NOT}-merged-bj-learn-clauses-bound:
assumes

$cdcl_{NOT}$ -merged-bj-learn** S T and
 inv S and
 $atms$ -of- mm ($clauses_{NOT}$ S) \subseteq $atms$ -of- ms A and
 atm -of ‘($lits$ -of- l ($trail$ S)) \subseteq $atms$ -of- ms A and
 n - d : no -dup ($trail$ S) and
 $finite[simp]$: $finite$ A
shows set - $mset$ ($clauses_{NOT}$ T) \subseteq set - $mset$ (not - $simplified$ - cls ($clauses_{NOT}$ S))
 \cup $simple$ - $clss$ ($atms$ -of- ms A)
using $assms(1-5)$
proof *induction*
case *base*
then show ?*case* **by** (*auto dest!*: *simple-clss-or-not-simplified-cls*)
next
case (*step* T U) **note** $st = this(1)$ and $cdcl_{NOT} = this(2)$ and $IH = this(3)[OF\ this(4-7)]$ and
 $inv = this(4)$ and $atms$ - $clss$ - $S = this(5)$ and $atms$ - $trail$ - $S = this(6)$ and $finite$ - cls - $S = this(7)$
have st' : $cdcl_{NOT}$ ** S T
using inv $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-is- $rtranclp$ - $cdcl_{NOT}$ -and- inv st n - d **by** *blast*
have inv T
using inv $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn- inv st n - d **by** *blast*
moreover
have $atms$ -of- mm ($clauses_{NOT}$ T) \subseteq $atms$ -of- ms A and
 atm -of ‘ $lits$ -of- l ($trail$ T) \subseteq $atms$ -of- ms A
using $rtranclp$ - $cdcl_{NOT}$ -trail- $clauses$ -bound[*OF* st'] inv $atms$ - $clss$ - S $atms$ - $trail$ - S n - d
by *blast+*
moreover moreover have no -dup ($trail$ T)
using $rtranclp$ - $cdcl_{NOT}$ - no -dup[*OF* ‘ $cdcl_{NOT}$ ** S T ’] inv n - d] **by** *fast*
ultimately have set - $mset$ ($clauses_{NOT}$ U)
 \subseteq set - $mset$ (not - $simplified$ - cls ($clauses_{NOT}$ T)) \cup $simple$ - $clss$ ($atms$ -of- ms A)
using $cdcl_{NOT}$ $finite$ $cdcl_{NOT}$ -merged-bj-learn- $clauses$ -bound
by (*auto intro!*: $cdcl_{NOT}$ -merged-bj-learn- $clauses$ -bound)
moreover have set - $mset$ (not - $simplified$ - cls ($clauses_{NOT}$ T))
 \subseteq set - $mset$ (not - $simplified$ - cls ($clauses_{NOT}$ S))
using $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn-not-simplified-decreasing[*OF* st] **by** *auto*
ultimately show ?*case* **using** IH inv $atms$ - $clss$ - S
by (*auto dest!*: *simple-clss-or-not-simplified-cls*)
qed

abbreviation μ_{CDCL}' -bound **where**
 μ_{CDCL}' -bound A $T \equiv ((2 + card\ (atms$ -of- ms $A)) \wedge (1 + card\ (atms$ -of- ms $A))) * 2$
 $+ card\ (set$ - $mset$ (not - $simplified$ - cls ($clauses_{NOT}$ T)))
 $+ 3 \wedge card\ (atms$ -of- ms $A)$

lemma $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn- $clauses$ -bound-card:

assumes

$cdcl_{NOT}$ -merged-bj-learn** S T and
 inv S and
 $atms$ -of- mm ($clauses_{NOT}$ S) \subseteq $atms$ -of- ms A and
 atm -of ‘($lits$ -of- l ($trail$ S)) \subseteq $atms$ -of- ms A and
 n - d : no -dup ($trail$ S) and
 $finite$: $finite$ A

shows μ_{CDCL}' -merged A $T \leq \mu_{CDCL}'$ -bound A S

proof –

have set - $mset$ ($clauses_{NOT}$ T) \subseteq set - $mset$ (not - $simplified$ - cls ($clauses_{NOT}$ S))
 \cup $simple$ - $clss$ ($atms$ -of- ms A)
using $rtranclp$ - $cdcl_{NOT}$ -merged-bj-learn- $clauses$ -bound[*OF* $assms$] .
moreover have $card$ (set - $mset$ (not - $simplified$ - cls ($clauses_{NOT}$ S))

$\cup \text{simple-clss } (\text{atms-of-ms } A)$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-clss}(\text{clauses}_{NOT} S))) + 3 \wedge \text{card } (\text{atms-of-ms } A)$
by (*meson* *Nat.le-trans* *atms-of-ms-finite* *simple-clss-card* *card-Un-le* *finite* *nat-add-left-cancel-le*)
ultimately have $\text{card } (\text{set-mset } (\text{clauses}_{NOT} T))$
 $\leq \text{card } (\text{set-mset } (\text{not-simplified-clss}(\text{clauses}_{NOT} S))) + 3 \wedge \text{card } (\text{atms-of-ms } A)$
by (*meson* *Nat.le-trans* *atms-of-ms-finite* *simple-clss-finite* *card-mono* *finite-UnI* *finite-set-mset* *local.finite*)
moreover have $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) - \mu_C' A T) * 2$
 $\leq (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A)) * 2$
by *auto*
ultimately show *?thesis unfolding* $\mu_{CDCL}'\text{-merged-def}$ **by** *auto*
qed

sublocale *cdcl_{NOT}-increasing-restarts-ops* $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}) S$
cdcl_{NOT}-merged-bj-learn *f*
 $\lambda A S. \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}'\text{-merged}$
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
apply *unfold-locales*
using *unbounded* **apply** *simp*
using *f-ge-1* **apply** *force*
apply (*blast* *dest!:* *cdcl_{NOT}-merged-bj-learn-is-tranclp-cdcl_{NOT}* *tranclp-into-rtranclp* *rtranclp-cdcl_{NOT}-trail-clauses-bound*)
apply (*simp* *add:* *cdcl_{NOT}-decreasing-measure'*)
using *rtranclp-cdcl_{NOT}-merged-bj-learn-clauses-bound-card* **apply** *blast*
apply (*drule* *rtranclp-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*)
apply (*auto* *simp:* *card-mono* *set-mset-mono*)[]
apply *simp*
apply *auto*[]
using *cdcl_{NOT}-merged-bj-learn-no-dup-inv* *cdcl-merged-inv* **apply** *blast*
apply (*auto* *simp:* *inv-restart*)[]
done

lemma *cdcl_{NOT}-restart- μ_{CDCL}' -merged-le- μ_{CDCL}' -bound:*

assumes
cdcl_{NOT}-restart *T V*
inv (*fst* *T*) **and**
no-dup (*trail* (*fst* *T*)) **and**
atms-of-mm (*clauses_{NOT}* (*fst* *T*)) \subseteq *atms-of-ms* *A* **and**
atm-of ' lits-of-l (*trail* (*fst* *T*)) \subseteq *atms-of-ms* *A* **and**
finite *A*
shows $\mu_{CDCL}'\text{-merged } A \text{ (fst } V) \leq \mu_{CDCL}'\text{-bound } A \text{ (fst } T)$
using *assms*
proof *induction*
case (*restart-full* *S T n*)
show *?case*
unfolding *fst-conv*
apply (*rule* *rtranclp-cdcl_{NOT}-merged-bj-learn-clauses-bound-card*)
using *restart-full* **unfolding** *full1-def* **by** (*force* *dest!:* *tranclp-into-rtranclp*) +
next
case (*restart-step* *m S T n U*) **note** *st = this(1)* **and** *U = this(3)* **and** *inv = this(4)* **and**
n-d = this(5) **and** *atms-clss = this(6)* **and** *atms-trail = this(7)* **and** *finite = this(8)*
then have *st': cdcl_{NOT}-merged-bj-learn*** *S T*

```

  by (blast dest: relpoup-imp-rtrancp)
then have st'': cdclNOT** S T
  using inv n-d apply - by (rule rtrancp-cdclNOT-merged-bj-learn-is-rtrancp-cdclNOT) auto
have inv T
  apply (rule rtrancp-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-mm (clausesNOT T)  $\subseteq$  atms-of-ms A
  using rtrancp-cdclNOT-trail-clauses-bound[OF st''] inv atms-clss atms-trail n-d
  by simp
then have atms-of-mm (clausesNOT U)  $\subseteq$  atms-of-ms A
  using U by simp
have not-simplified-cls (clausesNOT U)  $\subseteq$  # not-simplified-cls (clausesNOT T)
  using  $\langle U \sim \text{reduce-trail-to}_{\text{NOT}} [] T \rangle$  by auto
moreover have not-simplified-cls (clausesNOT T)  $\subseteq$  # not-simplified-cls (clausesNOT S)
  apply (rule rtrancp-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!: relpoup-imp-rtrancp)
ultimately have U-S: not-simplified-cls (clausesNOT U)  $\subseteq$  # not-simplified-cls (clausesNOT S)
  by auto

have (set-mset (clausesNOT U))
 $\subseteq$  set-mset (not-simplified-cls (clausesNOT U))  $\cup$  simple-clss (atms-of-ms A)
  apply (rule rtrancp-cdclNOT-merged-bj-learn-clauses-bound)
  apply simp
  using  $\langle \text{inv } U \rangle$  apply simp
  using  $\langle \text{atms-of-mm (clauses}_{\text{NOT}} U) \subseteq \text{atms-of-ms A} \rangle$  apply simp
  using U apply simp
  using U apply simp
  using finite apply simp
done
then have f1: card (set-mset (clausesNOT U))  $\leq$  card (set-mset (not-simplified-cls (clausesNOT 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 (clausesNOT U))  $\cup$  simple-clss (atms-of-ms A)
 $\subseteq$  set-mset (not-simplified-cls (clausesNOT S))  $\cup$  simple-clss (atms-of-ms A)
  using U-S by auto
then have f2:
  card (set-mset (not-simplified-cls (clausesNOT U))  $\cup$  simple-clss (atms-of-ms A))
 $\leq$  card (set-mset (not-simplified-cls (clausesNOT 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 (clausesNOT S))
 $\cup$  simple-clss (atms-of-ms A))
 $\leq$  card (set-mset (not-simplified-cls (clausesNOT 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 (clausesNOT U))
 $\leq$  card (set-mset (not-simplified-cls (clausesNOT S))) + 3  $\wedge$  card (atms-of-ms A)
  by linarith
then show ?case unfolding  $\mu_{CDCL}'$ -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 $\mu_{CDCL}'\text{-bound } A \text{ (fst } V) \leq \mu_{CDCL}'\text{-bound } A \text{ (fst } T)$
using *assms*(1–3)
proof *induction*
case (*restart-full* S T n)
have *not-simplified-cls* (*clauses_{NOT}* T) $\subseteq\#$ *not-simplified-cls* (*clauses_{NOT}* S)
apply (*rule* *rtrancpl-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*)
using $\langle \text{full1 } \text{cdcl}_{NOT}\text{-merged-bj-learn } S \text{ } T \rangle$ **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** $st = \text{this}(1)$ **and** $U = \text{this}(3)$ **and** $n\text{-d} = \text{this}(4)$ **and**
 $inv = \text{this}(5)$
then have st' : *cdcl_{NOT}-merged-bj-learn*^{**} S T
by (*blast* *dest*: *relopwp-imp-rtrancpl*)
then have st'' : *cdcl_{NOT}*^{**} S T
using inv $n\text{-d}$ **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' $n\text{-d}$ **by** *auto*
then have inv U
using U **by** (*auto* *simp*: *inv-restart*)
have *not-simplified-cls* (*clauses_{NOT}* U) $\subseteq\#$ *not-simplified-cls* (*clauses_{NOT}* T)
using $\langle U \sim \text{reduce-trail-to}_{NOT} [] :: 'a \text{ list} \rangle$ **by** *auto*
moreover have *not-simplified-cls* (*clauses_{NOT}* T) $\subseteq\#$ *not-simplified-cls* (*clauses_{NOT}* S)
apply (*rule* *rtrancpl-cdcl_{NOT}-merged-bj-learn-not-simplified-decreasing*)
using $\langle (\text{cdcl}_{NOT}\text{-merged-bj-learn } \widetilde{\sim} m) S \text{ } T \rangle$ **by** (*auto* *dest*!: *relopwp-imp-rtrancpl*)
ultimately have $U\text{-S}$: *not-simplified-cls* (*clauses_{NOT}* U) $\subseteq\#$ *not-simplified-cls* (*clauses_{NOT}* S)
by *auto*
then show ?*case* **by** (*auto* *simp*: *card-mono set-mset-mono*)
qed

sublocale *cdcl_{NOT}-increasing-restarts* - - - - f
 λS T . $T \sim \text{reduce-trail-to}_{NOT} ([] :: 'a \text{ list})$ S
 λA S . *atms-of-mm* (*clauses_{NOT}* S) \subseteq *atms-of-ms* A
 \wedge *atm-of* ' *lits-of-l* (*trail* S) \subseteq *atms-of-ms* $A \wedge \text{finite } A$
 $\mu_{CDCL}'\text{-merged } \text{cdcl}_{NOT}\text{-merged-bj-learn}$
 λS . inv $S \wedge \text{no-dup } (\text{trail } S)$
 λA T . $((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * 2$
 $+ \text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses}_{NOT} \text{ } T)))$
 $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$
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 $I \models \text{sextm } \text{clauses}_{NOT} \text{ (fst } S) \longleftrightarrow I \models \text{sextm } \text{clauses}_{NOT} \text{ (fst } T)$

```

using assms
proof (induction rule: cdclNOT-restart.induct)
  case (restart-full S T n)
  then have cdclNOT-merged-bj-learn** S T
    by (simp add: tranclp-into-rtranclp full1-def)
  then show ?case
    using rtranclp-cdclNOT-bj-sat-ext-iff restart-full.prems(1,2)
    rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT by auto
next
  case (restart-step m S T n U)
  then have cdclNOT-merged-bj-learn** S T
    by (auto simp: tranclp-into-rtranclp full1-def dest!: relpowp-imp-rtranclp)
  then have  $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T$ 
    using rtranclp-cdclNOT-bj-sat-ext-iff restart-step.prems(1,2)
    rtranclp-cdclNOT-merged-bj-learn-is-rtranclp-cdclNOT by auto
  moreover have  $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} U$ 
    using restart-step.hyps(3) by auto
  ultimately show ?case by auto
qed

lemma rtranclp-cdclNOT-restart-eq-sat-iff:
  assumes
    cdclNOT-restart** S T and
    inv: inv (fst S) and n-d: no-dup(trail (fst S))
  shows  $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } T)$ 
  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-cdclNOT-with-restart-cdclNOT-inv using st inv n-d by blast+
  then have  $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } T) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } U)$ 
    using cdclNOT-restart-eq-sat-iff cdcl by blast
  then show ?case using IH by blast
qed

lemma cdclNOT-restart-all-decomposition-implies-m:
  assumes
    cdclNOT-restart S T and
    inv: inv (fst S) and n-d: no-dup(trail (fst S)) and
    all-decomposition-implies-m (clausesNOT (fst S))
    (get-all-ann-decomposition (trail (fst S)))
  shows all-decomposition-implies-m (clausesNOT (fst T))
    (get-all-ann-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 rtrancp-cdclNOT-cdclNOT-inv[OF st] inv n-d by auto
  then show ?case
    using rtrancp-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 *rtrancp-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 (clausesNOT (fst S))
      (get-all-ann-decomposition (trail (fst S)))
  shows all-decomposition-implies-m (clausesNOT (fst T))
    (get-all-ann-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 rtrancp-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 (clausesNOT (fst S))
      (get-all-ann-decomposition (trail (fst S))) and
    atms-cls: atms-of-mm (clausesNOT (fst S)) ⊆ atms-of-ms A and
    atms-trail: atm-of ' lits-of-l (trail (fst S)) ⊆ atms-of-ms A and
    fin: finite A
  shows unsatisfiable (set-mset (clausesNOT (fst S)))
    ∨ lits-of-l (trail (fst T)) ⊨ sextm clausesNOT (fst S) ∧
    satisfiable (set-mset (clausesNOT (fst S)))
proof -
  have inv-T: inv (fst T) and n-d-T: no-dup (trail (fst T))
    using rtrancp-cdclNOT-with-restart-cdclNOT-inv using full inv n-d unfolding full-def by blast+
  moreover have
    atms-cls-T: atms-of-mm (clausesNOT (fst T)) ⊆ atms-of-ms A and
    atms-trail-T: atm-of ' lits-of-l (trail (fst T)) ⊆ atms-of-ms A
    using rtrancp-cdclNOT-with-restart-bound-inv[of S T A] full atms-cls atms-trail fin inv n-d
    unfolding full-def by blast+
  ultimately have no-step cdclNOT-merged-bj-learn (fst T)
  apply -
  apply (rule no-step-cdclNOT-restart-no-step-cdclNOT[of - A])
    using full unfolding full-def apply simp
  apply simp
  using fin apply simp

```

```

done
moreover have all-decomposition-implies-m (clausesNOT (fst T))
  (get-all-ann-decomposition (trail (fst T)))
  using rtrancpl-cdclNOT-restart-all-decomposition-implies-m[of S T] inv n-d decomp
  full unfolding full-def by auto
ultimately have unsatisfiable (set-mset (clausesNOT (fst T)))
  ∨ trail (fst T) ⊨asm clausesNOT (fst T) ∧ satisfiable (set-mset (clausesNOT (fst T)))
  apply -
  apply (rule cdclNOT-merged-bj-learn-final-state)
  using atms-cls-T atms-trail-T fin n-d-T fin inv-T by blast+
then consider
  (unsat) unsatisfiable (set-mset (clausesNOT (fst T)))
  | (sat) trail (fst T) ⊨asm clausesNOT (fst T) and satisfiable (set-mset (clausesNOT (fst T)))
  by auto
then show unsatisfiable (set-mset (clausesNOT (fst S)))
  ∨ lits-of-l (trail (fst T)) ⊨sextm clausesNOT (fst S) ∧
  satisfiable (set-mset (clausesNOT (fst S)))
proof cases
  case unsat
  then have unsatisfiable (set-mset (clausesNOT (fst S)))
    unfolding satisfiable-def apply auto
    using rtrancpl-cdclNOT-restart-eq-sat-iff[of S T] full inv n-d
    consistent-true-clss-ext-satisfiable true-clss-imp-true-cls-ext
    unfolding satisfiable-def full-def by blast
  then show ?thesis by blast
next
  case sat
  then have lits-of-l (trail (fst T)) ⊨sextm clausesNOT (fst T)
    using true-clss-imp-true-cls-ext by (auto simp: true-annots-true-cls)
  then have lits-of-l (trail (fst T)) ⊨sextm clausesNOT (fst S)
    using rtrancpl-cdclNOT-restart-eq-sat-iff[of S T] full inv n-d unfolding full-def by blast
  moreover then have satisfiable (set-mset (clausesNOT (fst S)))
    using consistent-true-clss-ext-satisfiable distinct-consistent-interp n-d-T by fast
  ultimately show ?thesis by fast
qed
qed

corollary full-cdclNOT-restart-normal-form-init-state:
  assumes
    init-state: trail S = [] clausesNOT S = N and
    full: full cdclNOT-restart (S, 0) T and
    inv: inv S
  shows unsatisfiable (set-mset N)
    ∨ lits-of-l (trail (fst T)) ⊨sextm N ∧ satisfiable (set-mset N)
  using full-cdclNOT-restart-normal-form[of (S, 0) T] assms by auto

end

end
theory DPLL-NOT
imports CDCL-NOT
begin

```

5.3 DPLL as an instance of NOT

5.3.1 DPLL with simple backtrack

We are using a concrete couple instead of an abstract state.

locale *dpll-with-backtrack*

begin

inductive *backtrack* :: ('v, unit) ann-lits × 'v clauses

⇒ ('v, unit) ann-lits × 'v clauses ⇒ bool **where**

backtrack-split (fst S) = (M', L # M) ⇒ is-decided L ⇒ D ∈# snd S

⇒ fst S ⊨_{as} CNot D ⇒ *backtrack* S (Propagated (− (lit-of L)) () # M, snd S)

inductive-cases *backtrackE*[elim]: *backtrack* (M, N) (M', N')

lemma *backtrack-is-backjump*:

fixes M M' :: ('v, unit) ann-lits

assumes

backtrack: *backtrack* (M, N) (M', N') **and**

no-dup: (no-dup ∘ fst) (M, N) **and**

decomp: all-decomposition-implies-m N (get-all-ann-decomposition M)

shows

∃ C F' K F L l C'.

M = F' @ Decided K # F ∧

M' = Propagated L l # F ∧ N = N' ∧ C ∈# N ∧ F' @ Decided K # F ⊨_{as} CNot C ∧

undefined-lit F L ∧ atm-of L ∈ atms-of-mm N ∪ atm-of ' lits-of-l (F' @ Decided K # F) ∧

N ⊨_{pm} C' + {#L#} ∧ F ⊨_{as} CNot C'

proof −

let ?S = (M, N)

let ?T = (M', N')

obtain F F' P L D **where**

b-sp: *backtrack-split* M = (F', L # F) **and**

is-decided L **and**

D ∈# snd ?S **and**

M ⊨_{as} CNot D **and**

bt: *backtrack* ?S (Propagated (− (lit-of L)) P # F, N) **and**

M': M' = Propagated (− (lit-of L)) P # F **and**

[simp]: N' = N

using *backtrackE*[OF *backtrack*] **by** (metis *backtrack* fstI sndI)

let ?K = lit-of L

let ?C = image-mset lit-of {#K ∈# mset M. is-decided K ∧ K ≠ L#} :: 'v clause

let ?C' = set-mset (image-mset single (?C + {#?K#}))

obtain K **where** L: L = Decided K **using** (is-decided L) **by** (cases L) auto

have M: M = F' @ Decided K # F

using *b-sp* **by** (metis L *backtrack-split-list-eq* fst-conv snd-conv)

moreover **have** F' @ Decided K # F ⊨_{as} CNot D

using (M ⊨_{as} CNot D) **unfolding** M .

moreover **have** undefined-lit F (−?K)

using *no-dup* **unfolding** M L **by** (simp add: defined-lit-map)

moreover **have** atm-of (−K) ∈ atms-of-mm N ∪ atm-of ' lits-of-l (F' @ Decided K # F)

by auto

moreover

have set-mset N ∪ ?C' ⊨_{ps} {{#}}

proof −

have A: set-mset N ∪ ?C' ∪ unmark-l M =

set-mset N ∪ unmark-l M


```

    unfolding M L by auto
  have set-mset N  $\cup \{\{\#lit-of L\# \} \mid L. is-decided L \wedge L \in set M\}$ 
     $\models_{ps} unmark-l M$ 
    using all-decomposition-implies-propagated-lits-are-implied[OF decomp] .
  moreover have C':  $?C' = \{\{\#lit-of L\# \} \mid L. is-decided L \wedge L \in set M\}$ 
    unfolding M L apply standard
    apply force
    using IntI by auto
  ultimately have N-C-M: set-mset N  $\cup ?C' \models_{ps} unmark-l M$ 
    by auto
  have set-mset N  $\cup (\lambda L. \{\#lit-of L\# \}) ' (set M) \models_{ps} \{\{\#\}\}$ 
    unfolding true-clss-clss-def
  proof (intro allI impI, goal-cases)
    case (1 I) note tot = this(1) and cons = this(2) and I-N-M = this(3)
    have I  $\models D$ 
      using I-N-M  $\langle D \in \# \text{ snd } ?S \rangle$  unfolding true-clss-def by auto
    moreover have I  $\models_s CNot D$ 
      using  $\langle M \models_{as} CNot D \rangle$  unfolding M by (metis 1(3)  $\langle M \models_{as} CNot D \rangle$ 
        true-annots-true-clss true-clss-mono-set-mset-l true-clss-def
        true-clss-singleton-lit-of-implies-incl true-clss-union)
    ultimately show ?case using cons consistent-CNot-not by blast
  qed
  then show ?thesis
    using true-clss-clss-left-right[OF N-C-M, of  $\{\{\#\}\}$ ] unfolding A by auto
  qed
  have N  $\models_{pm} image-mset uminus ?C + \{\#-?K\# \}$ 
    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 (set-mset N  $\cup \{image-mset uminus ?C + \{\#-?K\# \}\}$ )) and
      cons: consistent-interp I and
      I  $\models_{sm} N$ 
    have  $(K \in I \wedge -K \notin I) \vee (-K \in I \wedge K \notin I)$ 
      using cons tot unfolding consistent-interp-def L by (cases K) auto
    have  $\{a \in set M. is-decided a \wedge a \neq Decided K\} =$ 
       $set M \cap \{L. is-decided L \wedge L \neq Decided K\}$ 
      by auto
    then have
      tI: total-over-set I (atm-of 'lit-of ' (set M  $\cap \{L. is-decided L \wedge L \neq Decided K\}$ ))
      using tot by (auto simp add: L atms-of-uminus-lit-atm-of-lit-of)

  then have H:  $\bigwedge x.$ 
     $lit-of x \notin I \implies x \in set M \implies is-decided x$ 
     $\implies x \neq Decided K \implies -lit-of x \in I$ 
  proof -
    fix x :: ('v, unit) ann-lit
    assume a1:  $x \neq Decided K$ 
    assume a2: is-decided x
    assume a3:  $x \in set M$ 
    assume a4:  $lit-of x \notin I$ 
    have atm-of (lit-of x)  $\in atm-of 'lit-of ' (set M \cap \{m. is-decided m \wedge m \neq Decided K\})$ 
      using a3 a2 a1 by blast
    then have Pos (atm-of (lit-of x))  $\in I \vee Neg (atm-of (lit-of x)) \in I$ 
      using tI unfolding total-over-set-def by blast
  end

```

```

    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 (I  $\models_{sm} N$ )
    unfolding true-clss-clss-def total-over-m-def
    by (simp add: atms-of-uminus-lit-atm-of-lit-of atms-of-ms-single-image-atm-of-lit-of)
    then show  $I \models \text{image-mset } \text{uminus } ?C + \{\#\neg \text{lit-of } L\#\}$ 
    unfolding true-clss-def true-clss-def
    using (K  $\in I \wedge \neg K \notin I$ )  $\vee (\neg K \in I \wedge K \notin I)$ 
    unfolding L by (auto dest!: H)
    qed
    moreover
    have set  $F' \cap \{K. \text{is-decided } K \wedge K \neq L\} = \{\}$ 
    using backtrack-split-fst-not-decided[of - M] b-sp by auto
    then have  $F \models_{as} CNot (\text{image-mset } \text{uminus } ?C)$ 
    unfolding M CNot-def true-annots-def by (auto simp add: L lits-of-def)
    ultimately show ?thesis
    using M' (D  $\in \# \text{snd } ?S$ ) L by force
    qed

lemma backtrack-is-backjump':
  fixes M M' :: ('v, unit) ann-lits
  assumes
    backtrack: backtrack S T and
    no-dup: (no-dup  $\circ$  fst) S and
    decomp: all-decomposition-implies-m (snd S) (get-all-ann-decomposition (fst S))
  shows
     $\exists C F' K F L l C'. \text{fst } S = F' @ \text{Decided } K \# F \wedge$ 
     $T = (\text{Propagated } L l \# F, \text{snd } S) \wedge C \in \# \text{snd } S \wedge \text{fst } S \models_{as} CNot C$ 
     $\wedge \text{undefined-lit } F L \wedge \text{atm-of } L \in \text{atms-of-mm } (\text{snd } S) \cup \text{atm-of 'lits-of-l (fst } S) \wedge$ 
     $\text{snd } S \models_{pm} C' + \{\#L\#\} \wedge F \models_{as} CNot C'$ 
  apply (cases S, cases T)
  using backtrack-is-backjump[of fst S snd S fst T snd T] assms by fastforce

sublocale dp11-state
  fst snd  $\lambda L (M, N). (L \# M, N) \lambda (M, N). (tl M, N)$ 
   $\lambda C (M, N). (M, \{\#C\#\} + N) \lambda C (M, N). (M, \text{removeAll-mset } C N)$ 
  by unfold-locals (auto simp: ac-simps)

sublocale backjumping-ops
  fst snd  $\lambda L (M, N). (L \# M, N) \lambda (M, N). (tl M, N)$ 
   $\lambda C (M, N). (M, \{\#C\#\} + N) \lambda C (M, N). (M, \text{removeAll-mset } C N) \lambda - - S T. \text{backtrack } S T$ 
  by unfold-locals

thm reduce-trail-toNOT-clauses

lemma reduce-trail-toNOT:
  reduce-trail-toNOT F S =
    (if length (fst S)  $\geq$  length F
     then drop (length (fst S) - length F) (fst S)
     else [],
     snd S) (is ?R = ?C)
proof -
  have ?R = (fst ?R, snd ?R)

```

```

    by (cases reduce-trail-toNOT F S) auto
  also have (fst ?R, snd ?R) = ?C
    by (auto simp: trail-reduce-trail-toNOT-drop)
  finally show ?thesis .
qed

lemma backtrack-is-backjump'':
  fixes M M' :: ('v, unit) ann-lits
  assumes
    backtrack: backtrack S T and
    no-dup: (no-dup ∘ fst) S and
    decomp: all-decomposition-implies-m (snd S) (get-all-ann-decomposition (fst S))
  shows backjump S T
proof -
  obtain C F' K F L l C' where
    1: fst S = F' @ Decided K # F and
    2: T = (Propagated L l # F, snd S) and
    3: C ∈ # snd S and
    4: fst S ⊨as CNot C and
    5: undefined-lit F L and
    6: atm-of L ∈ atms-of-mm (snd S) ∪ atm-of ' lits-of-l (fst S) and
    7: snd S ⊨pm C' + {#L#} and
    8: F ⊨as CNot C'
  using backtrack-is-backjump'[OF assms] by force
  show ?thesis
  apply (cases S)
  using backjump.intros[OF 1 - - 4 5 - - 8, of T] 2 backtrack 1 5 3 6 7
  by (auto simp: state-eqNOT-def trail-reduce-trail-toNOT-drop
    reduce-trail-toNOT simp del: state-simpNOT)
qed

```

```

lemma can-do-bt-step:
  assumes
    M: fst S = F' @ Decided K # F and
    C ∈ # snd S and
    C: fst S ⊨as CNot C
  shows ¬ no-step backtrack S
proof -
  obtain L G' G where
    backtrack-split (fst S) = (G', L # G)
  unfolding M by (induction F' rule: ann-lit-list-induct) auto
  moreover then have is-decided L
    by (metis backtrack-split-snd-hd-decided list.distinct(1) list.sel(1) snd-conv)
  ultimately show ?thesis
    using backtrack.intros[of S G' L G C] (C ∈ # snd S) C unfolding M by auto
qed

```

end

```

sublocale dpll-with-backtrack ⊆ dpll-with-backjumping-ops
  fst snd λL (M, N). (L # M, N)
  λ(M, N). (tl M, N) λC (M, N). (M, {#C#} + N) λC (M, N). (M, removeAll-mset C N)
  λ(M, N). no-dup M ∧ all-decomposition-implies-m N (get-all-ann-decomposition M)
  λ- - - S T. backtrack S T
  λ- -. True
  apply unfold-locales

```

by (*metis* (*mono-tags*, *lifting*) *case-prod-beta comp-def dpll-with-backtrack.backtrack-is-backjump*''
dpll-with-backtrack.can-do-bt-step)

sublocale *dpll-with-backtrack* \subseteq *dpll-with-backjumping*

fst snd $\lambda L (M, N). (L \# M, N)$
 $\lambda(M, N). (tl\ M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, removeAll-mset\ C\ N)$
 $\lambda(M, N). no-dup\ M \wedge all-decomposition-implies-m\ N\ (get-all-ann-decomposition\ M)$
 $\lambda- - S\ T. backtrack\ S\ T$
 $\lambda- -. True$
apply *unfold-locales*
using *dpll-bj-no-dup dpll-bj-all-decomposition-implies-inv* **apply** *fastforce*
done

context *dpll-with-backtrack*

begin

lemma *wf-tranclp-dpll-inital-state*:

assumes *fin*: *finite A*
shows *wf* $\{((M'::('v, unit)\ ann-lits, N'::'v\ clauses), ([], N)) | M'\ N'\ N.$
 $dpll-bj^{++}\ ([], N)\ (M', N') \wedge atms-of-mm\ N \subseteq atms-of-ms\ A\}$
using *wf-tranclp-dpll-bj[OF assms(1)]* **by** (*rule wf-subset*) *auto*

corollary *full-dpll-final-state-conclusive*:

fixes *M M' :: ('v, unit)\ ann-lits*
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)\ ann-lits*
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 *distinct-consistent-interp satisfiable-carac' true-annots-true-cls* **by** *blast*
then show *?thesis*
using *full-dpll-final-state-conclusive[OF full]* **by** *auto*

qed

interpretation *conflict-driven-clause-learning-ops*

fst snd $\lambda L (M, N). (L \# M, N)$
 $\lambda(M, N). (tl\ M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, removeAll-mset\ C\ N)$
 $\lambda(M, N). no-dup\ M \wedge all-decomposition-implies-m\ N\ (get-all-ann-decomposition\ M)$
 $\lambda- - S\ T. backtrack\ S\ T$
 $\lambda- -. True\ \lambda- -. False\ \lambda- -. False$
by *unfold-locales*

interpretation *conflict-driven-clause-learning*

fst snd $\lambda L (M, N). (L \# M, N)$
 $\lambda(M, N). (tl\ M, N) \lambda C (M, N). (M, \{\#C\# \} + N) \lambda C (M, N). (M, removeAll-mset\ C\ N)$
 $\lambda(M, N). no-dup\ M \wedge all-decomposition-implies-m\ N\ (get-all-ann-decomposition\ M)$
 $\lambda- - S\ T. backtrack\ S\ T$

$\lambda - . \text{ True } \lambda - . \text{ False } \lambda - . \text{ False}$
apply *unfold-locales*
using *cdcl_{NOT}-all-decomposition-implies cdcl_{NOT}-no-dup* **by** *fastforce*

lemma *cdcl_{NOT}-is-dpll*:
 $cdcl_{NOT} S T \longleftrightarrow dpll\text{-}bj S T$
by (*auto simp: cdcl_{NOT}.simps learn.simps forget_{NOT}.simps*)

Another proof of termination:

lemma *wf* $\{(T, S). dpll\text{-}bj S T \wedge cdcl_{NOT}\text{-}NOT\text{-all-inv } A S\}$
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

5.3.2 Adding restarts

This was mainly a test whether it was possible to instantiate the assumption of the locale.

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 *cdcl_{NOT}-increasing-restarts*
 $fst\ snd\ \lambda L\ (M, N). (L \# M, N)\ \lambda(M, N). (tl\ M, N)$
 $\lambda C\ (M, N). (M, \{\#C\} + N)\ \lambda C\ (M, N). (M, removeAll\text{-}mset\ C\ N)\ f\ \lambda(-, N)\ S. S = ([], N)$
 $\lambda A\ (M, N). atms\text{-}of\text{-}mm\ N \subseteq atms\text{-}of\text{-}ms\ A \wedge atm\text{-}of\ ' lits\text{-}of\text{-}l\ M \subseteq atms\text{-}of\text{-}ms\ A \wedge finite\ A$
 $\wedge all\text{-}decomposition\text{-}implies\text{-}m\ N\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
 $\lambda A\ T. (2 + card\ (atms\text{-}of\text{-}ms\ A)) \wedge (1 + card\ (atms\text{-}of\text{-}ms\ A))$
 $\quad -\ \mu_C\ (1 + card\ (atms\text{-}of\text{-}ms\ A))\ (2 + card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T)\ dpll\text{-}bj$
 $\lambda(M, N). no\text{-}dup\ M \wedge all\text{-}decomposition\text{-}implies\text{-}m\ N\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
 $\lambda A\ -. (2 + card\ (atms\text{-}of\text{-}ms\ A)) \wedge (1 + card\ (atms\text{-}of\text{-}ms\ A))$
apply *unfold-locales*
apply (*rule unbounded*)
using *f-ge-1* **apply** *fastforce*
apply (*smt dpll-bj-all-decomposition-implies-inv dpll-bj-atms-in-trail-in-set*
dpll-bj-clauses id-apply prod.case-eq-if)
apply (*rule dpll-bj-trail-mes-decreasing-prop; auto*)
apply (*rename-tac A T U, case-tac T, simp*)
apply (*rename-tac A T U, case-tac U, simp*)
using *dpll-bj-clauses dpll-bj-all-decomposition-implies-inv dpll-bj-no-dup* **by** *fastforce* +
end
end
theory *DPLL-W*
imports *Main Partial-Clausal-Logic Partial-Annotated-Clausal-Logic List-More Wellfounded-More*
DPLL-NOT
begin

5.4 Weidenbach's DPLL

5.4.1 Rules

type-synonym $'a\ dpll_W\text{-}ann\text{-}lit = ('a, unit)\ ann\text{-}lit$
type-synonym $'a\ dpll_W\text{-}ann\text{-}lits = ('a, unit)\ ann\text{-}lits$

type-synonym $'v \text{ dpll}_W\text{-state} = 'v \text{ dpll}_W\text{-ann-lits} \times 'v \text{ clauses}$

abbreviation $\text{trail} :: 'v \text{ dpll}_W\text{-state} \Rightarrow 'v \text{ dpll}_W\text{-ann-lits}$ **where**
 $\text{trail} \equiv \text{fst}$

abbreviation $\text{clauses} :: 'v \text{ dpll}_W\text{-state} \Rightarrow 'v \text{ clauses}$ **where**
 $\text{clauses} \equiv \text{snd}$

inductive $\text{dpll}_W :: 'v \text{ dpll}_W\text{-state} \Rightarrow 'v \text{ dpll}_W\text{-state} \Rightarrow \text{bool}$ **where**
 $\text{propagate: } C + \{\#L\# \} \in \# \text{ clauses } S \Longrightarrow \text{trail } S \models_{\text{as}} C \text{Not } C \Longrightarrow \text{undefined-lit } (\text{trail } S) \ L$
 $\Longrightarrow \text{dpll}_W \ S \ (\text{Propagated } L \ () \ \# \ \text{trail } S, \text{ clauses } S) \mid$
 $\text{decided: } \text{undefined-lit } (\text{trail } S) \ L \Longrightarrow \text{atm-of } L \in \text{atms-of-mm } (\text{clauses } S)$
 $\Longrightarrow \text{dpll}_W \ S \ (\text{Decided } L \ \# \ \text{trail } S, \text{ clauses } S) \mid$
 $\text{backtrack: } \text{backtrack-split } (\text{trail } S) = (M', L \ \# \ M) \Longrightarrow \text{is-decided } L \Longrightarrow D \in \# \text{ clauses } S$
 $\Longrightarrow \text{trail } S \models_{\text{as}} C \text{Not } D \Longrightarrow \text{dpll}_W \ S \ (\text{Propagated } (- \ (\text{lit-of } L)) \ () \ \# \ M, \text{ clauses } S)$

5.4.2 Invariants

lemma $\text{dpll}_W\text{-distinct-inv:}$

assumes $\text{dpll}_W \ S \ S'$
and $\text{no-dup } (\text{trail } S)$
shows $\text{no-dup } (\text{trail } S')$
using assms

proof ($\text{induct rule: } \text{dpll}_W.\text{induct}$)

case ($\text{decided } L \ S$)

then show $?case$ **using** defined-lit-map **by force**

next

case ($\text{propagate } C \ L \ S$)

then show $?case$ **using** defined-lit-map **by force**

next

case ($\text{backtrack } S \ M' \ L \ M \ D$) **note** $\text{extracted} = \text{this}(1)$ **and** $\text{no-dup} = \text{this}(5)$

show $?case$

using $\text{no-dup backtrack-split-list-eq[of trail } S, \text{ symmetric}]$ **unfolding** extracted **by auto**

qed

lemma $\text{dpll}_W\text{-consistent-interp-inv:}$

assumes $\text{dpll}_W \ S \ S'$
and $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$
and $\text{no-dup } (\text{trail } S)$
shows $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S'))$
using assms

proof ($\text{induct rule: } \text{dpll}_W.\text{induct}$)

case ($\text{backtrack } S \ M' \ L \ M \ D$) **note** $\text{extracted} = \text{this}(1)$ **and** $\text{decided} = \text{this}(2)$ **and** $D = \text{this}(4)$ **and**
 $\text{cons} = \text{this}(5)$ **and** $\text{no-dup} = \text{this}(6)$

have $\text{no-dup}' : \text{no-dup } M$

by ($\text{metis } (\text{no-types}) \text{backtrack-split-list-eq distinct.simps}(2) \text{distinct-append extracted}$
 $\text{list.simps}(9) \text{map-append no-dup snd-conv}$)

then have $\text{insert } (\text{lit-of } L) \ (\text{lits-of-l } M) \subseteq \text{lits-of-l } (\text{trail } S)$

using $\text{backtrack-split-list-eq[of trail } S, \text{ symmetric}]$ **unfolding** extracted **by auto**

then have $\text{cons: consistent-interp } (\text{insert } (\text{lit-of } L) \ (\text{lits-of-l } M))$

using $\text{consistent-interp-subset cons}$ **by blast**

moreover

have $\text{lit-of } L \notin \text{lits-of-l } M$

using $\text{no-dup backtrack-split-list-eq[of trail } S, \text{ symmetric}]$ extracted

unfolding lits-of-def **by force**

moreover

have $\text{atm-of } (-\text{lit-of } L) \notin (\lambda m. \text{atm-of } (\text{lit-of } m)) \text{ ' set } M$

using *no-dup backtrack-split-list-eq*[of trail S , symmetric] **unfolding** *extracted by force*
 then have $\text{lit-of } L \notin \text{lits-of-l } M$
 unfolding *lits-of-def* **by force**
 ultimately show *?case* **by simp**
qed (*auto intro: consistent-add-undefined-lit-consistent*)

lemma *dpll_W-vars-in-snd-inv*:

assumes *dpll_W* $S S'$
 and *atm-of* ' (*lits-of-l* (trail S)) \subseteq *atms-of-mm* (*clauses* S)
 shows *atm-of* ' (*lits-of-l* (trail S')) \subseteq *atms-of-mm* (*clauses* S')
 using *assms*

proof (*induct rule: dpll_W.induct*)

case (*backtrack* $S M' L M D$)

then have *atm-of* (*lit-of* L) \in *atms-of-mm* (*clauses* S)
 using *backtrack-split-list-eq*[of trail S , symmetric] **by auto**

moreover

have *atm-of* ' *lits-of-l* (trail S) \subseteq *atms-of-mm* (*clauses* S)
 using *backtrack*(5) **by simp**

then have $\bigwedge x b. x b \in \text{set } M \implies \text{atm-of } (\text{lit-of } x b) \in \text{atms-of-mm } (\text{clauses } S)$
 using *backtrack-split-list-eq*[symmetric, of trail S] *backtrack.hyps*(1)

unfolding *lits-of-def* **by auto**

ultimately show *?case* **by** (*auto simp : lits-of-def*)

qed (*auto simp: in-plus-implies-atm-of-on-atms-of-ms*)

lemma *atms-of-ms-lit-of-atms-of*: *atms-of-ms* (($\lambda a. \{\# \text{lit-of } a \# \}$) ' c) = *atm-of* ' *lit-of* ' c

unfolding *atms-of-ms-def* **using** *image-iff* **by force**

theorem 2.8.2 page 73 of Weidenbach's book

lemma *dpll_W-propagate-is-conclusion*:

assumes *dpll_W* $S S'$

and *all-decomposition-implies-m* (*clauses* S) (*get-all-ann-decomposition* (trail S))

and *atm-of* ' *lits-of-l* (trail S) \subseteq *atms-of-mm* (*clauses* S)

shows *all-decomposition-implies-m* (*clauses* S') (*get-all-ann-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** $\text{inS} = \text{this}(1)$ **and** $\text{cnot} = \text{this}(2)$ **and** $\text{IH} = \text{this}(4)$ **and** $\text{undef} = \text{this}(3)$ **and** $\text{atms-incl} = \text{this}(5)$

let $?I = \text{set } (\text{map } (\lambda a. \{\# \text{lit-of } a \# \}) (\text{trail } S)) \cup \text{set-mset } (\text{clauses } S)$

have $?I \models_p C + \{\# L \# \}$ **by** (*auto simp add: inS*)

moreover have $?I \models_{ps} \text{CNot } C$ **using** *true-annots-true-clss-cls cnot* **by fastforce**

ultimately have $?I \models_p \{\# L \# \}$ **using** *true-clss-cls-plus-CNot*[of $?I C L$] *inS* **by blast**

{

assume *get-all-ann-decomposition* (trail S) = []

then have *?case* **by blast**

}

moreover {

assume $n: \text{get-all-ann-decomposition } (\text{trail } S) \neq []$

have $1: \bigwedge a b. (a, b) \in \text{set } (\text{tl } (\text{get-all-ann-decomposition } (\text{trail } S)))$

$\implies (\text{unmark-l } a \cup \text{set-mset } (\text{clauses } S)) \models_{ps} \text{unmark-l } b$

using *IH* **unfolding** *all-decomposition-implies-def* **by** (*fastforce simp add: list.set-sel*(2) n)

moreover have $2: \bigwedge a c. \text{hd } (\text{get-all-ann-decomposition } (\text{trail } S)) = (a, c)$

$\implies (\text{unmark-l } a \cup \text{set-mset } (\text{clauses } S)) \models_{ps} (\text{unmark-l } c)$

by (*metis IH all-decomposition-implies-cons-pair all-decomposition-implies-single*)

```

    list.collapse n)
moreover have  $\exists: \bigwedge a \ c. \text{hd} (\text{get-all-ann-decomposition} (\text{trail } S)) = (a, c)$ 
 $\implies (\text{unmark-l } a \cup \text{set-mset} (\text{clauses } S)) \models_p \{\#L\#$ 
proof -
  fix a c
  assume h:  $\text{hd} (\text{get-all-ann-decomposition} (\text{trail } S)) = (a, c)$ 
  have h':  $\text{trail } S = c @ a$  using get-all-ann-decomposition-decomp h by blast
  have I:  $\text{set} (\text{map} (\lambda a. \{\#\text{lit-of } a\#\}) a) \cup \text{set-mset} (\text{clauses } S)$ 
     $\cup \text{unmark-l } c \models_{ps} \text{CNot } C$ 
    using  $\langle ?I \models_{ps} \text{CNot } C \rangle$  unfolding h' by (simp add: Un-commute Un-left-commute)
  have
    atms-of-ms (CNot C)  $\subseteq$  atms-of-ms (set (map ( $\lambda a. \{\#\text{lit-of } a\#\}) a) \cup \text{set-mset} (\text{clauses } S))$ 
    and
    atms-of-ms (unmark-l c)  $\subseteq$  atms-of-ms (set (map ( $\lambda a. \{\#\text{lit-of } a\#\}) a)$ 
       $\cup \text{set-mset} (\text{clauses } S))$ 
    apply (metis CNot-plus Un-subset-iff atms-of-atms-of-ms-mono atms-of-ms-CNot-atms-of-
      atms-of-ms-union inS sup.coboundedI2)
    using inS atms-of-atms-of-ms-mono atms-incl by (fastforce simp: h')

  then have  $\text{unmark-l } a \cup \text{set-mset} (\text{clauses } S) \models_{ps} \text{CNot } C$ 
    using true-clss-clss-left-right[OF - I] h 2 by auto
  then show  $\text{unmark-l } a \cup \text{set-mset} (\text{clauses } S) \models_p \{\#L\#$ 
    by (metis (no-types) Un-insert-right inS insertI1 mk-disjoint-insert inS
      true-clss-clss-in true-clss-clss-plus-CNot)
qed
ultimately have ?case
  by (cases  $\text{hd} (\text{get-all-ann-decomposition} (\text{trail } S))$ )
    (auto simp: all-decomposition-implies-def)
}
ultimately show ?case by auto
next
case (backtrack S M' L M D) note extracted = this(1) and decided = this(2) and D = this(3) and
  cnot = this(4) and cons = this(4) and IH = this(5) and atms-incl = this(6)
have S:  $\text{trail } S = M' @ L \# M$ 
  using backtrack-split-list-eq[of trail S] unfolding extracted by auto
have M':  $\forall l \in \text{set } M'. \neg \text{is-decided } l$ 
  using extracted backtrack-split-fst-not-decided[of - trail S] by simp
have n:  $\text{get-all-ann-decomposition} (\text{trail } S) \neq []$  by auto
then have all-decomposition-implies-m (clauses S) ((L # M, M')
  # tl (get-all-ann-decomposition (trail S)))
  by (metis (no-types) IH extracted get-all-ann-decomposition-backtrack-split list.exhaust-sel)
then have 1:  $\text{unmark-l } (L \# M) \cup \text{set-mset} (\text{clauses } S) \models_{ps} (\lambda a. \{\#\text{lit-of } a\#\}) ' \text{set } M'$ 
  by simp
moreover
  have  $\text{unmark-l } (L \# M) \cup \text{unmark-l } M' \models_{ps} \text{CNot } D$ 
    by (metis (mono-tags, lifting) S Un-commute cons image-Un set-append
      true-annots-true-clss-clss)
  then have 2:  $\text{unmark-l } (L \# M) \cup \text{set-mset} (\text{clauses } S) \cup \text{unmark-l } M'$ 
     $\models_{ps} \text{CNot } D$ 
    by (metis (no-types, lifting) Un-assoc Un-left-commute true-clss-clss-union-l-r)
ultimately
  have  $\text{set} (\text{map} (\lambda a. \{\#\text{lit-of } a\#\}) (L \# M)) \cup \text{set-mset} (\text{clauses } S) \models_{ps} \text{CNot } D$ 
    using true-clss-clss-left-right by fastforce
  then have  $\text{set} (\text{map} (\lambda a. \{\#\text{lit-of } a\#\}) (L \# M)) \cup \text{set-mset} (\text{clauses } S) \models_p \{\#\}$ 
    by (metis (mono-tags, lifting) D Un-def mem-Collect-eq
      true-clss-clss-contradiction-true-clss-clss-false)

```



```

then have IL: unmark-l  $M \cup \text{set-mset}(\text{clauses } S) \models_p \{\# - \text{lit-of } L\# \}$ 
  using true-clss-clss-false-left-right by auto
show ?case unfolding S all-decomposition-implies-def
proof
  fix x P level
  assume x:  $x \in \text{set}(\text{get-all-ann-decomposition}$ 
    ( $\text{fst}(\text{Propagated}(-\text{lit-of } L) P \# M, \text{clauses } S)))$ 
  let ?M' =  $\text{Propagated}(-\text{lit-of } L) P \# M$ 
  let ?hd =  $\text{hd}(\text{get-all-ann-decomposition } ?M')$ 
  let ?tl =  $\text{tl}(\text{get-all-ann-decomposition } ?M')$ 
  have x = ?hd  $\vee x \in \text{set } ?tl$ 
  using x
  by (cases  $\text{get-all-ann-decomposition } ?M'$ )
    auto
  moreover {
    assume x':  $x \in \text{set } ?tl$ 
    have L':  $\text{Decided}(\text{lit-of } L) = L$  using decided by (cases L, auto)
    have  $x \in \text{set}(\text{get-all-ann-decomposition } (M' @ L \# M))$ 
      using x' get-all-ann-decomposition-except-last-choice-equal[ $\text{of } M' \text{ lit-of } L P M$ ]
      L' by (metis (no-types) M' list.set-sel(2) tl-Nil)
    then have case x of (Ls, seen)  $\Rightarrow \text{unmark-l } Ls \cup \text{set-mset}(\text{clauses } S)$ 
       $\models_{ps} \text{unmark-l seen}$ 
      using decided IH by (cases L) (auto simp add: S all-decomposition-implies-def)
  }
  moreover {
    assume x':  $x = ?hd$ 
    have tl:  $\text{tl}(\text{get-all-ann-decomposition } (M' @ L \# M)) \neq []$ 
    proof -
      have f1:  $\bigwedge ms. \text{length}(\text{get-all-ann-decomposition } (M' @ ms))$ 
        =  $\text{length}(\text{get-all-ann-decomposition } ms)$ 
      by (simp add: M' get-all-ann-decomposition-remove-undecided-length)
      have Suc ( $\text{length}(\text{get-all-ann-decomposition } M)) \neq \text{Suc } 0$ 
      by blast
      then show ?thesis
        using f1 decided by (metis (no-types) get-all-ann-decomposition.simps(1) length-tl
          list.sel(3) list.size(3) ann-lit.collapse(1))
    qed
    obtain M0' M0 where
      L0:  $\text{hd}(\text{tl}(\text{get-all-ann-decomposition } (M' @ L \# M))) = (M0, M0')$ 
      by (cases  $\text{hd}(\text{tl}(\text{get-all-ann-decomposition } (M' @ L \# M)))$ )
    have x'':  $x = (M0, \text{Propagated}(-\text{lit-of } L) P \# M0')$ 
      unfolding x' using get-all-ann-decomposition-last-choice tl M' L0
      by (metis decided ann-lit.collapse(1))
    obtain l-get-all-ann-decomposition where
       $\text{get-all-ann-decomposition}(\text{trail } S) = (L \# M, M') \# (M0, M0') \#$ 
      l-get-all-ann-decomposition
      using get-all-ann-decomposition-backtrack-split extracted by (metis (no-types) L0 S
        hd-Cons-tl n tl)
    then have  $M = M0' @ M0$  using get-all-ann-decomposition-hd-hd by fastforce
    then have IL':  $\text{unmark-l } M0 \cup \text{set-mset}(\text{clauses } S)$ 
       $\cup \text{unmark-l } M0' \models_{ps} \{\# - \text{lit-of } L\# \}$ 
      using IL by (simp add: Un-commute Un-left-commute image-Un)
    moreover have H:  $\text{unmark-l } M0 \cup \text{set-mset}(\text{clauses } S)$ 
       $\models_{ps} \text{unmark-l } M0'$ 
      using IH x'' unfolding all-decomposition-implies-def by (metis (no-types, lifting) L0 S
        list.set-sel(1) list.set-sel(2) old.prod.case tl tl-Nil)
  }

```

ultimately have case x of $(Ls, seen) \Rightarrow unmark-l Ls \cup set-mset (clauses S)$
 $\models_{ps} unmark-l seen$
 using *true-clss-clss-left-right unfolding* x'' by *auto*
 }
 ultimately show case x of $(Ls, seen) \Rightarrow$
 $unmark-l Ls \cup set-mset (snd (?M', clauses S))$
 $\models_{ps} unmark-l seen$
 unfolding *snd-conv* by *blast*
 qed
 qed

theorem 2.8.3 page 73 of Weidenbach's book

theorem *dpll_W-propagate-is-conclusion-of-decided*:
 assumes *dpll_W S S'*
 and *all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S))*
 and *atm-of ' lits-of-l (trail S) \subseteq atms-of-mm (clauses S)*
 shows *set-mset (clauses S') \cup { $\{\#lit-of L\#\} \mid L. is-decided L \wedge L \in set (trail S')$ }*
 $\models_{ps} (\lambda a. \{\#lit-of a\#}) ' \bigcup (set ' snd ' set (get-all-ann-decomposition (trail S')))$
 using *all-decomposition-implies-trail-is-implied*[*OF dpll_W-propagate-is-conclusion*[*OF assms*]] .

theorem 2.8.4 page 73 of Weidenbach's book

lemma *only-propagated-vars-unsat*:
 assumes *decided: $\forall x \in set M. \neg is-decided x$*
 and *DN: $D \in N$ and $D: M \models_{as} CNot D$*
 and *inv: all-decomposition-implies N (get-all-ann-decomposition M)*
 and *atm-incl: atm-of ' lits-of-l M \subseteq atms-of-ms N*
 shows *unsatisfiable N*
proof (*rule ccontr*)
 assume $\neg unsatisfiable N$
 then obtain I where
 $I: I \models_s N$ and
 $cons: consistent-interp I$ and
 $tot: total-over-m I N$
 unfolding *satisfiable-def* by *auto*
 then have $I-D: I \models D$
 using *DN unfolding true-clss-def* by *auto*

 have $l0: \{\{\#lit-of L\#\} \mid L. is-decided L \wedge L \in set M\} = \{\}$ using *decided* by *auto*
 have $atms-of-ms (N \cup unmark-l M) = atms-of-ms N$
 using *atm-incl unfolding atms-of-ms-def lits-of-def* by *auto*

 then have $total-over-m I (N \cup (\lambda a. \{\#lit-of a\#}) ' (set M))$
 using *tot unfolding total-over-m-def* by *auto*
 then have $I \models_s (\lambda a. \{\#lit-of a\#}) ' (set M)$
 using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons I*
 unfolding *true-clss-clss-def l0* by *auto*
 then have $IM: I \models_s unmark-l M$ by *auto*
 {
 fix K
 assume $K \in \# D$
 then have $-K \in lits-of-l M$
 by (*auto split: if-split-asm*
 intro: *allE*[*OF D*[*unfolded true-annots-def Ball-def*], of $\{\#-K\#\}$])
 then have $-K \in I$ using IM *true-clss-singleton-lit-of-implies-incl* by *fastforce*
 }
 then have $\neg I \models D$ using *cons unfolding true-clss-def consistent-interp-def* by *auto*

then show *False* **using** *I-D* **by** *blast*
qed

lemma *dp_{ll}_W-same-clauses*:

assumes *dp_{ll}_W S S'*

shows *clauses S = clauses S'*

using *assms* **by** (*induct rule: dp_{ll}_W.induct, auto*)

lemma *rtranc_lp-dp_{ll}_W-inv*:

assumes *rtranc_lp dp_{ll}_W S S'*

and *inv*: *all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S))*

and *atm-incl*: *atm-of ' lits-of-l (trail S) \subseteq atms-of-mm (clauses S)*

and *consistent-interp (lits-of-l (trail S))*

and *no-dup (trail S)*

shows *all-decomposition-implies-m (clauses S') (get-all-ann-decomposition (trail S'))*

and *atm-of ' lits-of-l (trail S') \subseteq atms-of-mm (clauses S')*

and *clauses S = clauses S'*

and *consistent-interp (lits-of-l (trail S'))*

and *no-dup (trail S')*

using *assms*

proof (*induct rule: rtranc_lp-induct*)

case *base*

show

all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S)) **and**

atm-of ' lits-of-l (trail S) \subseteq atms-of-mm (clauses S) **and**

clauses S = clauses S **and**

consistent-interp (lits-of-l (trail S)) **and**

no-dup (trail S) **using** *assms* **by** *auto*

next

case (*step S' S''*) **note** *dp_{ll}_WStar = this(1)* **and** *IH = this(3,4,5,6,7)* **and**

dp_{ll}_W = this(2)

moreover

assume

inv: *all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S))* **and**

atm-incl: *atm-of ' lits-of-l (trail S) \subseteq atms-of-mm (clauses S)* **and**

cons: *consistent-interp (lits-of-l (trail S))* **and**

no-dup (trail S)

ultimately have *decomp*: *all-decomposition-implies-m (clauses S')*

(get-all-ann-decomposition (trail S')) **and**

atm-incl': *atm-of ' lits-of-l (trail S') \subseteq atms-of-mm (clauses S')* **and**

snd: *clauses S = clauses S'* **and**

cons': *consistent-interp (lits-of-l (trail S'))* **and**

no-dup': *no-dup (trail S')* **by** *blast+*

show *clauses S = clauses S''* **using** *dp_{ll}_W-same-clauses[OF dp_{ll}_W] snd* **by** *metis*

show *all-decomposition-implies-m (clauses S'') (get-all-ann-decomposition (trail S''))*

using *dp_{ll}_W-propagate-is-conclusion[OF dp_{ll}_W] decomp atm-incl'* **by** *auto*

show *atm-of ' lits-of-l (trail S'') \subseteq atms-of-mm (clauses S'')*

using *dp_{ll}_W-vars-in-snd-inv[OF dp_{ll}_W] atm-incl atm-incl'* **by** *auto*

show *no-dup (trail S'')* **using** *dp_{ll}_W-distinct-inv[OF dp_{ll}_W] no-dup' dp_{ll}_W* **by** *auto*

show *consistent-interp (lits-of-l (trail S''))*

using *cons' no-dup' dp_{ll}_W-consistent-interp-inv[OF dp_{ll}_W]* **by** *auto*

qed

definition *dp_{ll}_W-all-inv S* \equiv

(all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S)))

$\wedge \text{atm-of } \text{' lits-of-l (trail S) } \subseteq \text{atms-of-mm (clauses S)}$
 $\wedge \text{consistent-interp (lits-of-l (trail S))}$
 $\wedge \text{no-dup (trail S)}$

lemma *dpll_W-all-inv-dest*[*dest*]:
assumes *dpll_W-all-inv S*
shows *all-decomposition-implies-m (clauses S) (get-all-ann-decomposition (trail S))*
and *atm-of ' lits-of-l (trail S) \subseteq atms-of-mm (clauses S)*
and *consistent-interp (lits-of-l (trail S)) \wedge no-dup (trail S)*
using *assms unfolding dpll_W-all-inv-def lits-of-def* **by** *auto*

lemma *rtrancpl-dpll_W-all-inv*:
assumes *rtrancpl dpll_W S S'*
and *dpll_W-all-inv S*
shows *dpll_W-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) \subseteq atms-of-mm N*
shows *rtrancpl dpll_W ([], N) (map Decided M, N)*
using *assms*
proof (*induct M*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons L M*)
then have *undefined-lit (map Decided M) L*
unfolding *defined-lit-def consistent-interp-def* **by** *auto*
moreover have *atm-of L \in atms-of-mm N* **using** *Cons.prem(3)* **by** *auto*
ultimately have *dpll_W (map Decided M, N) (map Decided (L # M), N)*
using *dpll_W.decided* **by** *auto*
moreover have *consistent-interp (set M)* **and** *distinct M* **and** *atm-of ' set M \subseteq atms-of-mm 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 ($\forall L \in \text{set (trail S)}. \neg \text{is-decided } L$))

$\wedge (\exists C \in \# \text{ clauses } S. \text{ trail } S \models_{as} C \text{Not } C)))$

theorem 2.8.6 page 74 of Weidenbach's book

lemma *dpll_W-strong-completeness*:

assumes *set* $M \models_{sm} N$

and *consistent-interp* (*set* M)

and *distinct* M

and *atm-of* ' (*set* M) \subseteq *atms-of-mm* N

shows $dpll_W^{**} ([], N) (\text{map } Decided\ M, N)$

and *conclusive-dpll_W-state* (*map* *Decided* M, N)

proof –

show *rtrancpl* $dpll_W ([], N) (\text{map } Decided\ M, N)$ **using** *dpll_W-can-do-step* *assms* **by** *auto*

have *map* *Decided* $M \models_{asm} N$ **using** *assms*(1) *true-annots-decided-true-cl* **by** *auto*

then show *conclusive-dpll_W-state* (*map* *Decided* M, N)

unfolding *conclusive-dpll_W-state-def* **by** *auto*

qed

theorem 2.8.5 page 73 of Weidenbach's book

lemma *dpll_W-sound*:

assumes

rtrancpl $dpll_W ([], N) (M, N)$ **and**

$\forall S. \neg dpll_W (M, N) S$

shows $M \models_{asm} N \longleftrightarrow \text{satisfiable } (\text{set-mset } N) (\text{is } ?A \longleftrightarrow ?B)$

proof

let $?M' = \text{lits-of-l } M$

assume $?A$

then have $?M' \models_{sm} N$ **by** (*simp add: true-annots-true-cl*)

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 $(\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of-mm } N) \vee (\exists D \in \# N. M \models_{as} C \text{Not } D)$

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 $(\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of } D) \vee M \models_{as} C \text{Not } D$

unfolding *true-annots-def* *Ball-def* *CNot-def* *true-annot-def*

using *atm-of-lit-in-atms-of* *true-annot-iff-decided-or-true-lit* *true-cl-def* **by** *blast*

then show *?thesis*

by (*metis* *Bex-def* *D* *atms-of-atms-of-ms-mono* *rev-subsetD*)

qed

moreover {

assume $\exists L. \text{undefined-lit } M\ L \wedge \text{atm-of } L \in \text{atms-of-mm } N$

then have *False* **using** *assms*(2) *decided* **by** *fastforce*

}

moreover {

assume $\exists D \in \# N. M \models_{as} C \text{Not } D$

then obtain D **where** $DN: D \in \# N$ **and** $MD: M \models_{as} C \text{Not } D$ **by** *auto*

{

assume $\forall l \in \text{set } M. \neg \text{is-decided } l$

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 set-mset N] DN MD
  rtranchp-dpllW-all-inv[OF assms(1)] by force
then have False using ⟨?B⟩ by blast
}
moreover {
  assume l: ∃ l ∈ set M. is-decided l
  then have False
    using backtrack[of (M, N) - - D ] DN MD assms(2)
    backtrack-split-some-is-decided-then-snd-has-hd[OF l]
    by (metis backtrack-split-snd-hd-decided fst-conv list.distinct(1) list.sel(1) snd-conv)
  }
ultimately have False by blast
}
ultimately show False by blast
qed
qed

```

5.4.3 Termination

definition $dpll_W\text{-mes } M \ n =$
 $\text{map } (\lambda l. \text{if is-decided } l \text{ then } 2 \text{ else } (1::\text{nat})) (\text{rev } M) \ @ \ \text{replicate } (n - \text{length } M) \ 3$

lemma $\text{length-dpll}_W\text{-mes}$:
assumes $\text{length } M \leq n$
shows $\text{length } (dpll_W\text{-mes } M \ n) = n$
using *assms* **unfolding** $dpll_W\text{-mes-def}$ **by** *auto*

lemma $\text{distinctcard-atm-of-lit-of-eq-length}$:
assumes *no-dup S*
shows $\text{card } (\text{atm-of } \text{'lits-of-l } S) = \text{length } S$
using *assms* **by** (*induct S*) (*auto simp add: image-image lits-of-def*)

lemma $dpll_W\text{-card-decrease}$:
assumes $dpll$: $dpll_W \ S \ S'$ **and** $\text{length } (\text{trail } S') \leq \text{card vars}$
and $\text{length } (\text{trail } S) \leq \text{card vars}$
shows $(dpll_W\text{-mes } (\text{trail } S') \ (\text{card vars}), dpll_W\text{-mes } (\text{trail } S) \ (\text{card vars}))$
 $\in \text{lexn } \{(a, b). a < b\} \ (\text{card vars})$
using *assms*

proof (*induct rule: dpll_W.induct*)
case (*propagate C L S*)
have m : $\text{map } (\lambda l. \text{if is-decided } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S))$
 $\ @ \ \text{replicate } (\text{card vars} - \text{length } (\text{trail } S)) \ 3$
 $= \text{map } (\lambda l. \text{if is-decided } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S)) \ @ \ 3$
 $\ \# \ \text{replicate } (\text{card vars} - \text{Suc } (\text{length } (\text{trail } S))) \ 3$
using *propagate.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*
then show *?case*
using *propagate.prem(1)* **unfolding** $dpll_W\text{-mes-def}$ **by** (*fastforce simp add: lexn-conv assms(2)*)

next

case (*decided S L*)
have m : $\text{map } (\lambda l. \text{if is-decided } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S))$
 $\ @ \ \text{replicate } (\text{card vars} - \text{length } (\text{trail } S)) \ 3$
 $= \text{map } (\lambda l. \text{if is-decided } l \text{ then } 2 \text{ else } 1) (\text{rev } (\text{trail } S)) \ @ \ 3$
 $\ \# \ \text{replicate } (\text{card vars} - \text{Suc } (\text{length } (\text{trail } S))) \ 3$
using *decided.prem[simplified]* **using** *Suc-diff-le* **by** *fastforce*
then show *?case*

using *decided.premis* **unfolding** *dpll_W-mes-def* **by** (*force simp add: lexn-conv assms(2)*)
 next
 case (*backtrack S M' L M D*)
 have *L: is-decided L* **using** *backtrack.hyps(2)* **by** *auto*
 have *S: trail S = M' @ L # M*
 using *backtrack.hyps(1) backtrack-split-list-eq[of trail S]* **by** *auto*
 show *?case*
 using *backtrack.premis L* **unfolding** *dpll_W-mes-def S* **by** (*fastforce simp add: lexn-conv assms(2)*)
 qed

theorem 2.8.7 page 74 of Weidenbach's book

lemma *dpll_W-card-decrease'*:

assumes *dpll: dpll_W S S'*
 and *atm-incl: atm-of ' lits-of-l (trail S) ⊆ atms-of-mm (clauses S)*
 and *no-dup: no-dup (trail S)*
 shows (*dpll_W-mes (trail S') (card (atms-of-mm (clauses S')))*),
 (*dpll_W-mes (trail S) (card (atms-of-mm (clauses S)))*) ∈ *lex {(a, b). a < b}*

proof –

have *finite (atms-of-mm (clauses S))* **unfolding** *atms-of-ms-def* **by** *auto*
 then have *1: length (trail S) ≤ card (atms-of-mm (clauses S))*
 using *distinctcard-atm-of-lit-of-eq-length[OF no-dup] atm-incl card-mono* **by** *metis*

moreover

have *no-dup': no-dup (trail S')* **using** *dpll 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-l (trail S') ⊆ atms-of-mm (clauses S')*
 using *atm-incl dpll dpll_W-vars-in-snd-inv[OF dpll]* **by** *force*
 have *finite (atms-of-mm (clauses S'))*
 unfolding *atms-of-ms-def* **by** *auto*
 then have *2: length (trail S') ≤ card (atms-of-mm (clauses S'))*
 using *distinctcard-atm-of-lit-of-eq-length[OF no-dup'] atm-incl' card-mono SS'* **by** *metis*

ultimately have (*dpll_W-mes (trail S') (card (atms-of-mm (clauses S)))*),
 (*dpll_W-mes (trail S) (card (atms-of-mm (clauses S)))*)
 ∈ *lex {(a, b). a < b} (card (atms-of-mm (clauses S)))*
 using *dpll_W-card-decrease[OF assms(1), of atms-of-mm (clauses S)]* **by** *blast*
 then have (*dpll_W-mes (trail S') (card (atms-of-mm (clauses S)))*),
 (*dpll_W-mes (trail S) (card (atms-of-mm (clauses S)))*) ∈ *lex {(a, b). a < b}*
 unfolding *lex-def* **by** *auto*
 then show (*dpll_W-mes (trail S') (card (atms-of-mm (clauses S')))*),
 (*dpll_W-mes (trail S) (card (atms-of-mm (clauses S)))*) ∈ *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-mm (clauses S))))*

proof –

have *m: {(a, b). a < b} = measure id* **by** *auto*
 show *?thesis* **apply** (*rule wf-lexn*) **unfolding** *m* **by** *auto*

qed

lemma *dpll_W-wf*:

wf {(S', S). dpll_W-all-inv S ∧ dpll_W S S'}
apply (*rule wf-wf-if-measure'[OF wf-lex-less, of - -*
 λS. dpll_W-mes (trail S) (card (atms-of-mm (clauses S)))])
using *dpll_W-card-decrease'* **by** *fast*

lemma *dpll_W-trancp-star-commute*:
 $\{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+ = \{(S', S). \text{dpll}_W\text{-all-inv } S \wedge \text{trancp dpll}_W S S'\}$
 (is ?A = ?B)
proof
 { fix S S'
 assume (S, S') ∈ ?A
 then have (S, S') ∈ ?B
 by (induct rule: tranc1.induct, auto)
 }
 then show ?A ⊆ ?B by blast
 { fix S S'
 assume (S, S') ∈ ?B
 then have $\text{dpll}_W^{++} S' S$ and $\text{dpll}_W\text{-all-inv } S'$ by auto
 then have (S, S') ∈ ?A
 proof (induct rule: trancp.induct)
 case r-into-tranc1
 then show ?case by (simp-all add: r-into-tranc1')
 next
 case (tranc1-into-tranc1 S S' S'')
 then have (S', S) ∈ {a. case a of (S', S) ⇒ $\text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+$ by blast
 moreover have $\text{dpll}_W\text{-all-inv } S'$
 using rtrancp-dpll_W-all-inv[OF trancp-into-rtrancp[OF tranc1-into-tranc1.hyps(1)]]
 tranc1-into-tranc1.prem by auto
 ultimately have (S'', S') ∈ {(pa, p). $\text{dpll}_W\text{-all-inv } p \wedge \text{dpll}_W p pa\}^+$
 using ⟨ $\text{dpll}_W\text{-all-inv } S'$ ⟩ tranc1-into-tranc1.hyps(3) by blast
 then show ?case
 using ⟨(S', S) ∈ {a. case a of (S', S) ⇒ $\text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W S S'\}^+ \rangle$ by auto
 qed
 }
 then show ?B ⊆ ?A by blast
qed

lemma *dpll_W-wf-trancp*: wf {(S', S). $\text{dpll}_W\text{-all-inv } S \wedge \text{dpll}_W^{++} S S'\}$
 unfolding dpll_W-trancp-star-commute[symmetric] by (simp add: dpll_W-wf wf-tranc1)

lemma *dpll_W-wf-plus*:
 wf {(S', ([], N)) | S'. $\text{dpll}_W^{++} ([], N) S'\}$ (is wf ?P)
 apply (rule wf-subset[OF dpll_W-wf-trancp, of ?P])
 unfolding dpll_W-all-inv-def by auto

5.4.4 Final States

Proposition 2.8.1: final states are the normal forms of dpll_W

lemma *dpll_W-no-more-step-is-a-conclusive-state*:

assumes $\forall S'. \neg \text{dpll}_W S S'$
 shows conclusive-dpll_W-state S

proof –

have vars: $\forall s \in \text{atms-of-mm } (\text{clauses } S). s \in \text{atm-of ' lits-of-l } (\text{trail } S)$
proof (rule ccontr)
 assume $\neg (\forall s \in \text{atms-of-mm } (\text{clauses } S). s \in \text{atm-of ' lits-of-l } (\text{trail } S))$
 then obtain L where
 L-in-atms: $L \in \text{atms-of-mm } (\text{clauses } S)$ and
 L-not-in-trail: $L \notin \text{atm-of ' lits-of-l } (\text{trail } S)$ bymetis
 obtain L' where $L': \text{atm-of } L' = L$ by (meson literal.sel(2))


```

then have undefined-lit (trail S) L'
  unfolding Decided-Propagated-in-iff-in-lits-of-l by (metis L-notin-trail atm-of-uminus imageI)
then show False using dpllW.decided assms(1) L-in-atms L' by blast
qed
show ?thesis
proof (rule ccontr)
  assume not-final:  $\neg$  ?thesis
  then have
     $\neg$  trail S  $\models_{asm}$  clauses S and
     $(\exists L \in set (trail S). is-decided L) \vee (\forall C \in \# clauses S. \neg trail S \models_{as} CNot C)$ 
  unfolding conclusive-dpllW-state-def by auto
  moreover {
    assume  $\exists L \in set (trail S). is-decided L$ 
    then obtain L M' M where L: backtrack-split (trail S) = (M', L # M)
      using backtrack-split-some-is-decided-then-snd-has-hd by blast
    obtain D where D  $\in \# clauses S$  and  $\neg trail S \models_a D$ 
      using  $\langle \neg trail S \models_{asm} clauses S \rangle$  unfolding true-annots-def by auto
    then have  $\forall s \in atms-of-ms \{D\}. s \in atm-of \text{ 'lits-of-l (trail S) }$ 
      using vars unfolding atms-of-ms-def by auto
    then have trail S  $\models_{as} CNot D$ 
      using all-variables-defined-not-imply-cnot[of D]  $\langle \neg trail S \models_a D \rangle$  by auto
    moreover have is-decided L
      using L by (metis backtrack-split-snd-hd-decided list.distinct(1) list.sel(1) snd-conv)
    ultimately have False
      using assms(1) dpllW.backtrack L  $\langle D \in \# clauses S \rangle \langle trail S \models_{as} CNot D \rangle$  by blast
  }
  moreover {
    assume tr:  $\forall C \in \# clauses S. \neg trail S \models_{as} CNot C$ 
    obtain C where C-in-cl: C  $\in \# clauses S$  and trC:  $\neg trail S \models_a C$ 
      using  $\langle \neg trail S \models_{asm} clauses S \rangle$  unfolding true-annots-def by auto
    have  $\forall s \in atms-of-ms \{C\}. s \in atm-of \text{ 'lits-of-l (trail S) }$ 
      using vars  $\langle C \in \# clauses S \rangle$  unfolding atms-of-ms-def by auto
    then have trail S  $\models_{as} CNot C$ 
      by (meson C-in-cl tr trC all-variables-defined-not-imply-cnot)
    then have False using tr C-in-cl by auto
  }
  ultimately show False by blast
qed
qed

```

```

lemma dpllW-conclusive-state-correct:
  assumes dpllW** ([], N) (M, N) and conclusive-dpllW-state (M, N)
  shows M  $\models_{asm} N \longleftrightarrow$  satisfiable (set-mset N) (is ?A  $\longleftrightarrow$  ?B)
proof
  let ?M' = lits-of-l M
  assume ?A
  then have ?M'  $\models_{sm} N$  by (simp add: true-annots-true-cl)
  moreover have consistent-interp ?M'
    using rtranclp-dpllW-inv-starting-from-0[OF assms(1)] by auto
  ultimately show ?B by auto
next
  assume ?B
  show ?A
  proof (rule ccontr)
    assume n:  $\neg$  ?A
    have no-mark:  $\forall L \in set M. \neg is-decided L \exists C \in \# N. M \models_{as} CNot C$ 

```

```

    using  $n$  assms(2) unfolding conclusive-dpllW-state-def by auto
    moreover obtain  $D$  where  $DN$ :  $D \in \# N$  and  $MD$ :  $M \models_{as} CNot D$  using no-mark by auto
    ultimately have unsatisfiable (set-mset N)
      using only-propagated-vars-unsat rtrancpl-dpllW-all-inv[OF assms(1)]
      unfolding dpllW-all-inv-def by force
      then show False using  $\langle ?B \rangle$  by blast
  qed
qed

```

5.4.5 Link with NOT's DPLL

interpretation *dpll_{W-NOT}*: *dpll-with-backtrack* .

```

declare dpllW-NOT.state-simpNOT[simp del]
lemma state-eqNOT-iff-eq[iff, simp]: dpllW-NOT.state-eqNOT S T  $\longleftrightarrow$   $S = T$ 
  unfolding dpllW-NOT.state-eqNOT-def by (cases S, cases T) auto
lemma dpllW-dpllW-bj:
  assumes inv: dpllW-all-inv S and dpll: dpllW S T
  shows dpllW-NOT.dpll-bj S T
  using dpll inv
  apply (induction rule: dpllW.induct)
    apply (rule dpllW-NOT.bj-propagateNOT)
    apply (rule dpllW-NOT.propagateNOT.propagateNOT; simp?)
    apply fastforce
    apply (rule dpllW-NOT.bj-decideNOT)
    apply (rule dpllW-NOT.decideNOT.decideNOT; simp?)
    apply fastforce
    apply (frule dpllW-NOT.backtrack.intros[of - - - -], simp-all)
    apply (rule dpllW-NOT.dpll-bj.bj-backjump)
    apply (rule dpllW-NOT.backtrack-is-backjump'',
      simp-all add: dpllW-all-inv-def)
  done

lemma dpllW-bj-dpll:
  assumes inv: dpllW-all-inv S and dpll: dpllW-NOT.dpll-bj S T
  shows dpllW S T
  using dpll
  apply (induction rule: dpllW-NOT.dpll-bj.induct)
    apply (elim dpllW-NOT.decideNOTE, cases S)
    apply (frule decided; simp)

    apply (elim dpllW-NOT.propagateNOTE, cases S)
    apply (auto intro!: propagate[of - - (-, -), simplified])[]
    apply (elim dpllW-NOT.backjumpE, cases S)
  by (simp add: dpllW.simps dpll-with-backtrack.backtrack.simps)

lemma rtrancpl-dpllW-rtrancpl-dpllW-NOT:
  assumes dpllW** S T and dpllW-all-inv S
  shows dpllW-NOT.dpll-bj** S T
  using assms apply (induction)
  apply simp
  by (auto intro: rtrancpl-dpllW-all-inv dpllW-dpllW-bj rtrancpl.rtrancpl-into-rtrancpl)

lemma rtrancpl-dpll-rtrancpl-dpllW:
  assumes dpllW-NOT.dpll-bj** S T and dpllW-all-inv S
  shows dpllW** S T

```

```

using assms apply (induction)
apply simp
by (auto intro: dpllW-bj-dpll rtrancp.rtranc-into-rtranc rtrancp-dpllW-all-inv)

lemma dpll-conclusive-state-correctness:
  assumes dpllW-NOT.dpll-bj** ([], N) (M, N) and conclusive-dpllW-state (M, N)
  shows M ⊨asm N ⟷ satisfiable (set-mset N)
proof -
  have dpllW-all-inv ([], N)
  unfolding dpllW-all-inv-def by auto
  show ?thesis
  apply (rule dpllW-conclusive-state-correct)
  apply (simp add: ⟨dpllW-all-inv ([], N)⟩ assms(1) rtrancp-dpll-rtrancp-dpllW)
  using assms(2) by simp
qed

end
theory CDCL-W-Level
imports Partial-Annotated-Clausal-Logic
begin

```

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.

abbreviation *count-decided* :: ('v, 'm) ann-lits ⇒ nat **where**
count-decided l ≡ length (filter is-decided l)

abbreviation *get-level* :: ('v, 'm) ann-lits ⇒ 'v literal ⇒ nat **where**
get-level S L ≡ length (filter is-decided (dropWhile (λS. atm-of (lit-of S) ≠ atm-of L) S))

lemma *get-level-uminus*: *get-level* M (−L) = *get-level* M L
 by auto

lemma *atm-of-notin-get-rev-level-eq-0*[simp]:
 assumes atm-of L ∉ atm-of ' lits-of-l M
 shows *get-level* M L = 0
 using assms by (induct M rule: ann-lit-list-induct) auto

lemma *get-level-ge-0-atm-of-in*:
 assumes *get-level* M L > n
 shows atm-of L ∈ atm-of ' lits-of-l M
 using assms by (induct M arbitrary: n rule: ann-lit-list-induct) fastforce+

In *get-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 ∉ atm-of ' lits-of-l M
 shows *get-level* (M @ M') L = *get-level* M' L
 using assms by (induct M rule: ann-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 ∈ atm-of ' lits-of-l M

shows $\text{get-level } (M @ M') L = \text{get-level } M L + \text{length } (\text{filter is-decided } M')$
using *assms* **by** (*induct* M' *rule*: *ann-lit-list-induct*) (*auto simp*: *lits-of-def*)

lemma *get-level-skip-beginning*:
assumes $\text{atm-of } L' \neq \text{atm-of } (\text{lit-of } K)$
shows $\text{get-level } (K \# M) L' = \text{get-level } M L'$
using *assms* **by** *auto*

lemma *get-level-skip-beginning-not-decided*[*simp*]:
assumes $\text{atm-of } L \notin \text{atm-of ' lits-of-l } S$
and $\forall s \in \text{set } S. \neg \text{is-decided } s$
shows $\text{get-level } (M @ S) L = \text{get-level } M L$
using *assms* **apply** (*induction* S *rule*: *ann-lit-list-induct*)
apply *auto*[2]
apply (*case-tac* $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$)
apply (*auto simp*: *image-iff lits-of-def filter-empty-conv dest*: *set-dropWhileD*)
done

lemma *get-level-skip-in-all-not-decided*:
fixes $M :: ('a, 'b) \text{ ann-lits}$ **and** $L :: 'a \text{ literal}$
assumes $\forall m \in \text{set } M. \neg \text{is-decided } m$
and $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$
shows $\text{get-level } M L = 0$
using *assms* **by** (*induction* M *rule*: *ann-lit-list-induct*) *auto*

lemma *get-level-skip-all-not-decided*[*simp*]:
fixes M
assumes $\forall m \in \text{set } M. \neg \text{is-decided } m$
shows $\text{get-level } M L = 0$
using *assms* **by** (*auto simp*: *filter-empty-conv dest*: *set-dropWhileD*)

abbreviation $M\text{Max } M \equiv \text{Max } (\text{set-mset } M)$

the $\{\#0 :: 'a\# \}$ is there to ensure that the set is not empty.

definition *get-maximum-level* :: $('a, 'b) \text{ ann-lits} \Rightarrow 'a \text{ literal multiset} \Rightarrow \text{nat}$
where
 $\text{get-maximum-level } M D = M\text{Max } (\{\#0\# \} + \text{image-mset } (\text{get-level } M) D)$

lemma *get-maximum-level-ge-get-level*:
 $L \in \# D \Rightarrow \text{get-maximum-level } M D \geq \text{get-level } M L$
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-empty*[*simp*]:
 $\text{get-maximum-level } M \{\#\} = 0$
unfolding *get-maximum-level-def* **by** *auto*

lemma *get-maximum-level-exists-lit-of-max-level*:
 $D \neq \{\#\} \Rightarrow \exists L \in \# D. \text{get-level } M L = \text{get-maximum-level } M D$
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*]:
 $\text{get-maximum-level } [] D = 0$
unfolding *get-maximum-level-def* **by** (*simp add*: *image-constant-conv*)

lemma *get-maximum-level-single*[simp]:
get-maximum-level $M \{ \#L\# \} = \text{get-level } M L$
unfolding *get-maximum-level-def* **by** *simp*

lemma *get-maximum-level-plus*:
get-maximum-level $M (D + D') = \max (\text{get-maximum-level } M D) (\text{get-maximum-level } M D')$
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 $f: \text{finite } (\text{insert } 0 ((\lambda L. \text{get-level } M L) \text{ `set-mset } D))$ **by** *auto*
then have $n \in ((\lambda L. \text{get-level } M L) \text{ `set-mset } D)$
using $n \text{ max Max-in}[OF f]$ **unfolding** *get-maximum-level-def* **by** *simp*
then show $\exists L \in \#D. \text{get-level } M L = n$ **by** *auto*
qed

lemma *get-maximum-level-skip-first*[simp]:
assumes *atm-of* $L \notin \text{atms-of } D$
shows *get-maximum-level* $(\text{Propagated } L C \# M) D = \text{get-maximum-level } M D$
using *assms* **unfolding** *get-maximum-level-def* *atms-of-def*
atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
by (*smt atm-of-in-atm-of-set-in-uminus get-level-skip-beginning image-iff ann-lit.sel*(2)
multiset.map-cong0)

lemma *get-maximum-level-skip-beginning*:
assumes *DH*: $\forall x \in \text{atms-of } D. x \notin \text{atm-of `lits-of-l } c$
shows *get-maximum-level* $(c @ H) D = \text{get-maximum-level } H D$
proof –
have $(\text{get-level } (c @ H)) \text{ `set-mset } D = (\text{get-level } H) \text{ `set-mset } D$
apply (*rule image-cong*)
apply *simp*
using *DH* **unfolding** *atms-of-def* **by** *auto*
then show *?thesis* **using** *DH* **unfolding** *get-maximum-level-def* **by** *auto*
qed

lemma *get-maximum-level-D-single-propagated*:
get-maximum-level $[\text{Propagated } x21 \ x22] D = 0$
unfolding *get-maximum-level-def* **by** (*simp add: image-constant-conv*)

lemma *get-maximum-level-skip-un-decided-not-present*:
assumes
 $\forall L \in \#D. \text{atm-of } L \notin \text{atm-of `lits-of-l } M$ **and**
 $\forall m \in \text{set } M. \neg \text{is-decided } m$
shows *get-maximum-level* $(M @ aa) D = \text{get-maximum-level } aa D$
using *assms* **unfolding** *get-maximum-level-def* **by** *simp*

lemma *get-maximum-level-union-mset*:
get-maximum-level $M (A \# \cup B) = \text{get-maximum-level } M (A + B)$
unfolding *get-maximum-level-def* **by** (*auto simp: image-Un*)

lemma *count-decided-rev*[simp]:
count-decided $(\text{rev } M) = \text{count-decided } M$

```

by (auto simp: rev-filter[symmetric])

lemma count-decided-ge-get-level[simp]:
  count-decided  $M \geq$  get-level  $M L$ 
by (induct  $M$  rule: ann-lit-list-induct) (auto simp add: le-max-iff-disj)

lemma count-decided-ge-get-maximum-level:
  count-decided  $M \geq$  get-maximum-level  $M D$ 
  using get-maximum-level-exists-lit-of-max-level unfolding Bex-def
  by (metis get-maximum-level-empty count-decided-ge-get-level le0)

fun get-all-mark-of-propagated where
  get-all-mark-of-propagated [] = [] |
  get-all-mark-of-propagated (Decided - #  $L$ ) = 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: ann-lit-list-induct) auto

```

Properties about the levels

```

lemma atm-lit-of-set-lits-of-l:
  ( $\lambda l.$  atm-of (lit-of  $l$ )) ' set  $xs =$  atm-of ' lits-of- $l$   $xs$ 
  unfolding lits-of-def by auto

lemma le-count-decided-decomp:
  assumes no-dup  $M$ 
  shows  $i <$  count-decided  $M \longleftrightarrow (\exists c K c'. M = c @$  Decided  $K \# c' \wedge$  get-level  $M K =$  Suc  $i$ )
    (is  $?A \longleftrightarrow ?B$ )
proof
  assume  $?B$ 
  then obtain  $c K c'$  where
     $M = c @$  Decided  $K \# c'$  and get-level  $M K =$  Suc  $i$ 
  by blast
  then show  $?A$  using count-decided-ge-get-level[of  $K M$ ] by auto
next
  assume  $?A$ 
  then show  $?B$ 
  using  $\langle$ no-dup  $M \rangle$ 
  proof (induction  $M$  rule: ann-lit-list-induct)
    case Nil
    then show  $?case$  by simp
  next
    case (Decided  $L M$ ) note  $IH =$  this(1) and  $i =$  this(2) and  $n-d =$  this(3)
    then have  $n-d-M$ : no-dup  $M$  by simp
    show  $?case$ 
    proof (cases  $i <$  count-decided  $M$ )
      case True
      then obtain  $c K c'$  where
         $M$ :  $M = c @$  Decided  $K \# c'$  and lev- $K$ : get-level  $M K =$  Suc  $i$ 
      using  $IH$   $n-d-M$  by blast
      show  $?thesis$ 
      apply (rule exI[of - Decided  $L \# c$ ])
      apply (rule exI[of -  $K$ ])
      apply (rule exI[of -  $c'$ ])

```

```

    using lev-K n-d unfolding M by auto
next
  case False
  show ?thesis
    apply (rule exI[of - []])
    apply (rule exI[of - L])
    apply (rule exI[of - M])
    using False i by auto
  qed
next
  case (Propagated L mark' M) note i = this(2) and n-d = this(3) and IH = this(1)
  then obtain c K c' where
    M: M = c @ Decided K # c' and lev-K: get-level M K = Suc i
    by auto
  show ?case
    apply (rule exI[of - Propagated L mark' # c])
    apply (rule exI[of - K])
    apply (rule exI[of - c'])
    using lev-K n-d unfolding M by (auto simp: atm-lit-of-set-lits-of-l)
  qed
qed

end
theory CDCL-W
imports List-More CDCL-W-Level Wellfounded-More Partial-Annotated-Clausal-Logic

begin

```


Chapter 6

Weidenbach's CDCL

The organisation of the development is the following:

- `CDCL_W.thy` contains the specification of the rules: the rules and the strategy are defined, and we proof the correctness of CDCL.
- `CDCL_W_Termination.thy` contains the proof of termination.
- `CDCL_W_Merge.thy` contains a variant of the calculus: some rules of the raw calculus are always applied together (like the rules analysing the conflict and then backtracking). We define an equivalent version of the calculus where these rules are applied together. This is useful for implementations.
- `CDCL_WNOT.thy` proves the inclusion of Weidenbach's version of CDCL in NOT's version. We use here the version defined in `CDCL_W_Merge.thy`. We need this, because NOT's backjump corresponds to multiple applications of three rules in Weidenbach's calculus. We show also the termination of the calculus without strategy.

We have some variants build on the top of Weidenbach's CDCL calculus:

- `CDCL_W_Incremental.thy` adds incrementality on the top of `CDCL_W.thy`. The way we are doing it is not compatible with `CDCL_W_Merge.thy`, because we add conflicts and the `CDCL_W_Merge.thy` cannot analyse conflicts added externally, because the conflict and analyse are merged.
- `CDCL_W_Restart.thy` adds restart. It is built on the top of `CDCL_W_Merge.thy`.

6.1 Weidenbach's CDCL with Multisets

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

6.1.1 The State

We will abstract the representation of clause and clauses via two locales. We here use multisets, contrary to `CDCL_W_Abstract_State.thy` where we assume only the existence of a conversion to the state.

`locale stateW-ops =`

fixes

trail :: 'st \Rightarrow ('v, 'v clause) ann-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, 'v clause) ann-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-trail :: 'st \Rightarrow 'st **and**
add-learned-clss :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**
remove-clss :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**
update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st **and**
update-conflicting :: 'v clause option \Rightarrow 'st \Rightarrow 'st **and**

init-state :: 'v clauses \Rightarrow 'st

begin

abbreviation *hd-trail* :: 'st \Rightarrow ('v, 'v clause) ann-lit **where**
hd-trail *S* \equiv *hd* (*trail* *S*)

definition *clauses* :: 'st \Rightarrow 'v clauses **where**
clauses *S* = *init-clss* *S* + *learned-clss* *S*

abbreviation *resolve-clss* **where**

resolve-clss *L* *D'* *E* \equiv *remove1-mset* ($-L$) *D'* $\# \cup$ *remove1-mset* *L* *E*

abbreviation *state* :: 'st \Rightarrow ('v, 'v clause) ann-lits \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*)
end

We are using an abstract state to abstract away the detail of the implementation: we do not need to know how the clauses are represented internally, we just need to know that they can be converted to multisets.

Weidenbach state is a five-tuple composed of:

1. the trail is a list of decided literals;
2. the initial set of clauses (that is not changed during the whole calculus);
3. the learned clauses (clauses can be added or remove);
4. the maximum level of the trail;
5. the conflicting clause (if any has been found so far).

There are two different clause representation: one for the conflicting clause ('v *Partial-Clausal-Logic.clause*, standing for conflicting clause) and one for the initial and learned clauses ('v *Partial-Clausal-Logic.clause*, standing for clause). The representation of the clauses annotating literals in the trail is slightly different: being able to convert it to 'v *Partial-Clausal-Logic.clause* is enough (needed for function *hd-trail* below).

There are several axioms to state the independance of the different fields of the state: for example, adding a clause to the learned clauses does not change the trail.

locale *state_W* =

stateW-ops

— functions about the state:

— getter:

trail init-clss learned-clss backtrack-lvl conflicting

— setter:

*cons-trail tl-trail add-learned-clss remove-clss update-backtrack-lvl
update-conflicting*

— Some specific states:

init-state

for

trail :: 'st \Rightarrow ('v, 'v clause) ann-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, 'v clause) ann-lit \Rightarrow 'st \Rightarrow 'st **and**

tl-trail :: 'st \Rightarrow 'st **and**

add-learned-clss :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**

remove-clss :: 'v clause \Rightarrow 'st \Rightarrow 'st **and**

update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st **and**

update-conflicting :: 'v clause option \Rightarrow 'st \Rightarrow 'st **and**

init-state :: 'v clauses \Rightarrow 'st +

assumes

cons-trail:

$\bigwedge S'. \text{state } st = (M, S') \implies$
state (cons-trail L st) = (L # M, S') **and**

tl-trail:

$\bigwedge S'. \text{state } st = (M, S') \implies \text{state } (tl\text{-trail } st) = (tl\ M, S') \text{ **and**}$

remove-clss:

$\bigwedge S'. \text{state } st = (M, N, U, S') \implies$
state (remove-clss C st) =
(M, removeAll-mset C N, removeAll-mset C U, S') **and**

add-learned-clss:

$\bigwedge S'. \text{state } st = (M, N, U, S') \implies$
state (add-learned-clss C st) = (M, N, {#C#} + U, S') **and**

update-backtrack-lvl:

$\bigwedge S'. \text{state } st = (M, N, U, k, S') \implies$
state (update-backtrack-lvl k' st) = (M, N, U, k', S') **and**

update-conflicting:

state st = (M, N, U, k, D) \implies
state (update-conflicting E st) = (M, N, U, k, E) **and**

init-state:

state (init-state N) = ([], N, {#}, 0, None)

begin

lemma

trail-cons-trail[simp]:

$trail (cons-trail L st) = L \# trail st$ **and**
 $trail-tl-trail[simp]: trail (tl-trail st) = tl (trail st)$ **and**
 $trail-add-learned-cls[simp]:$
 $trail (add-learned-cls C st) = trail st$ **and**
 $trail-remove-cls[simp]:$
 $trail (remove-cls C st) = trail st$ **and**
 $trail-update-backtrack-lvl[simp]: trail (update-backtrack-lvl k st) = trail st$ **and**
 $trail-update-conflicting[simp]: trail (update-conflicting E st) = trail st$ **and**

$init-clss-cons-trail[simp]:$
 $init-clss (cons-trail M st) = init-clss st$
and
 $init-clss-tl-trail[simp]:$
 $init-clss (tl-trail st) = init-clss st$ **and**
 $init-clss-add-learned-cls[simp]:$
 $init-clss (add-learned-cls C st) = init-clss st$ **and**
 $init-clss-remove-cls[simp]:$
 $init-clss (remove-cls C st) = removeAll-mset C (init-clss st)$ **and**
 $init-clss-update-backtrack-lvl[simp]:$
 $init-clss (update-backtrack-lvl k st) = init-clss st$ **and**
 $init-clss-update-conflicting[simp]:$
 $init-clss (update-conflicting E st) = init-clss st$ **and**

$learned-clss-cons-trail[simp]:$
 $learned-clss (cons-trail M st) = learned-clss st$ **and**
 $learned-clss-tl-trail[simp]:$
 $learned-clss (tl-trail st) = learned-clss st$ **and**
 $learned-clss-add-learned-cls[simp]:$
 $learned-clss (add-learned-cls C st) = \{\#C\# \} + learned-clss st$ **and**
 $learned-clss-remove-cls[simp]:$
 $learned-clss (remove-cls C st) = removeAll-mset C (learned-clss st)$ **and**
 $learned-clss-update-backtrack-lvl[simp]:$
 $learned-clss (update-backtrack-lvl k st) = learned-clss st$ **and**
 $learned-clss-update-conflicting[simp]:$
 $learned-clss (update-conflicting E st) = learned-clss st$ **and**

$backtrack-lvl-cons-trail[simp]:$
 $backtrack-lvl (cons-trail M st) = backtrack-lvl st$ **and**
 $backtrack-lvl-tl-trail[simp]:$
 $backtrack-lvl (tl-trail st) = backtrack-lvl st$ **and**
 $backtrack-lvl-add-learned-cls[simp]:$
 $backtrack-lvl (add-learned-cls C st) = backtrack-lvl st$ **and**
 $backtrack-lvl-remove-cls[simp]:$
 $backtrack-lvl (remove-cls C st) = backtrack-lvl st$ **and**
 $backtrack-lvl-update-backtrack-lvl[simp]:$
 $backtrack-lvl (update-backtrack-lvl k st) = k$ **and**
 $backtrack-lvl-update-conflicting[simp]:$
 $backtrack-lvl (update-conflicting E st) = backtrack-lvl st$ **and**

$conflicting-cons-trail[simp]:$
 $conflicting (cons-trail M st) = conflicting st$ **and**
 $conflicting-tl-trail[simp]:$
 $conflicting (tl-trail st) = conflicting st$ **and**
 $conflicting-add-learned-cls[simp]:$
 $conflicting (add-learned-cls C st) = conflicting st$
and

conflicting-remove-cls[simp]:
conflicting (remove-cls C st) = *conflicting* st **and**
conflicting-update-backtrack-lvl[simp]:
conflicting (update-backtrack-lvl k st) = *conflicting* st **and**
conflicting-update-conflicting[simp]:
conflicting (update-conflicting E st) = E **and**

init-state-trail[simp]: trail (init-state N) = [] **and**
init-state-clss[simp]: init-clss (init-state N) = N **and**
init-state-learned-clss[simp]: learned-clss (init-state N) = {#}
init-state-backtrack-lvl[simp]: backtrack-lvl (init-state N) = 0 **and**
init-state-conflicting[simp]: conflicting (init-state N) = None

using cons-trail[of st] tl-trail[of st] add-learned-cls[of st - - - C]
update-backtrack-lvl[of st - - - - k] update-conflicting[of st - - - - E]
remove-cls[of st - - - - C]
init-state[of N]
by (cases state st ; auto simp:)+

lemma

shows

clauses-cons-trail[simp]:
clauses (cons-trail M S) = clauses S **and**

clss-tl-trail[simp]: clauses (tl-trail S) = clauses S **and**
clauses-add-learned-cls-unfolded:
clauses (add-learned-cls U S) = {# U #} + learned-clss S + init-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) = removeAll-mset C (clauses S) **and**
clauses-add-learned-cls[simp]:
clauses (add-learned-cls C S) = {# C #} + clauses S **and**
clauses-init-state[simp]: clauses (init-state N) = N
by (auto simp: ac-simps replicate-mset-plus clauses-def intro: multiset-eqI)

abbreviation incr-lvl :: ' $st \Rightarrow 'st$ where

incr-lvl $S \equiv$ update-backtrack-lvl (backtrack-lvl S + 1) S

definition state-eq :: ' $st \Rightarrow 'st \Rightarrow bool$ (infix \sim 50) where

$S \sim T \longleftrightarrow$ state S = state T

lemma state-eq-ref[simp, intro]:

$S \sim S$

unfolding state-eq-def **by** auto

lemma state-eq-sym:

$S \sim T \longleftrightarrow T \sim S$

unfolding state-eq-def **by** auto

lemma state-eq-trans:

$S \sim T \Longrightarrow T \sim U \Longrightarrow S \sim U$

unfolding state-eq-def **by** auto

lemma

shows

state-eq-trail: $S \sim T \implies \text{trail } S = \text{trail } T$ **and**
state-eq-init-clss: $S \sim T \implies \text{init-clss } S = \text{init-clss } T$ **and**
state-eq-learned-clss: $S \sim T \implies \text{learned-clss } S = \text{learned-clss } T$ **and**
state-eq-backtrack-lvl: $S \sim T \implies \text{backtrack-lvl } S = \text{backtrack-lvl } T$ **and**
state-eq-conflicting: $S \sim T \implies \text{conflicting } S = \text{conflicting } T$ **and**
state-eq-clauses: $S \sim T \implies \text{clauses } S = \text{clauses } T$ **and**
state-eq-undefined-lit: $S \sim T \implies \text{undefined-lit } (\text{trail } S) L = \text{undefined-lit } (\text{trail } T) L$
unfolding *state-eq-def clauses-def* **by** *auto*

lemma *state-eq-conflicting-None*:

$S \sim T \implies \text{conflicting } T = \text{None} \implies \text{conflicting } S = \text{None}$
unfolding *state-eq-def clauses-def* **by** *auto*

We combine all simplification rules about $op \sim$ in a single list of theorems. While they are handy as simplification rule as long as we are working on the state, they also cause a *huge* slow-down in all other cases.

lemmas *state-simp*[*simp*] = *state-eq-trail state-eq-init-clss state-eq-learned-clss*
state-eq-backtrack-lvl state-eq-conflicting state-eq-clauses state-eq-undefined-lit
state-eq-conflicting-None

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* [$::('v, 'v \text{ clause}) \text{ ann-lits } S$ *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 *clauses* (*reduce-trail-to* ([::'a list] (*tl-trail Sa*))) = *clauses* (*tl-trail Sa*)
 \vee *trail Sa* = []
by *fastforce*
then show *clauses* (*reduce-trail-to* ([::'a list] *Sa*)) = *clauses Sa*
by (*metis* (*no-types*) *length-0-conv* *reduce-trail-to-eq-length* *clss-tl-trail*
reduce-trail-to-length-ne)
qed

lemma *reduce-trail-to-skip-beginning*:
assumes *trail S = F' @ F*
shows *trail* (*reduce-trail-to F S*) = *F*
using *assms* **by** (*induction F' arbitrary: S*) (*auto simp: reduce-trail-to-length-ne*)

lemma *clauses-reduce-trail-to[simp]*:
clauses (*reduce-trail-to F S*) = *clauses S*
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis clss-tl-trail reduce-trail-to.simps*)

lemma *conflicting-update-trail[simp]*:
conflicting (*reduce-trail-to F S*) = *conflicting S*
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis conflicting-tl-trail reduce-trail-to.simps*)

lemma *backtrack-lvl-update-trail[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-trail[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-trail[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 *conflicting-reduce-trail-to[simp]*:
conflicting (*reduce-trail-to F S*) = *None* \longleftrightarrow *conflicting S = None*
apply (*induction F S rule: reduce-trail-to.induct*)
by (*metis conflicting-update-trail map-option-is-None*)

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 ~ T*
shows *reduce-trail-to F S ~ reduce-trail-to F T*
proof –
have *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
using *trail-eq-reduce-trail-to-eq[of S T F] ST* **by** *auto*

then show *?thesis* **using** *ST* **by** (*auto simp del: state-simp simp: state-eq-def*)
qed

lemma *reduce-trail-to-trail-tl-trail-decomp*[*simp*]:
 $trail\ S = F' @ Decided\ K \# F \implies (trail\ (reduce-trail-to\ F\ S)) = F$
apply (*rule reduce-trail-to-skip-beginning*[*of - F' @ Decided K # []*])
by (*cases F'*) (*auto simp add:tl-append reduce-trail-to-skip-beginning*)

lemma *reduce-trail-to-add-learned-cls*[*simp*]:
 $trail\ (reduce-trail-to\ F\ (add-learned-cls\ C\ S)) = trail\ (reduce-trail-to\ F\ S)$
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-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*]:
 $trail\ (reduce-trail-to\ F\ (update-backtrack-lvl\ k\ S)) = trail\ (reduce-trail-to\ F\ S)$
by (*rule trail-eq-reduce-trail-to-eq*) *auto*

lemma *reduce-trail-to-length*:
 $length\ M = length\ M' \implies reduce-trail-to\ M\ S = reduce-trail-to\ M'\ S$
apply (*induction M S rule: reduce-trail-to.induct*)
by (*simp add: reduce-trail-to.simps*)

lemma *trail-reduce-trail-to-drop*:
 $trail\ (reduce-trail-to\ F\ S) =$
 $(if\ length\ (trail\ S) \geq length\ F$
 $then\ drop\ (length\ (trail\ S) - length\ F)\ (trail\ S)$
 $else\ [])$
apply (*induction F S rule: reduce-trail-to.induct*)
apply (*rename-tac F S, case-tac trail S*)
apply *auto*
apply (*rename-tac list, case-tac Suc (length list) > length F*)
prefer 2 apply (*metis diff-is-0-eq drop-Cons' length-Cons nat-le-linear nat-less-le*
 $reduce-trail-to-eq-length\ trail-reduce-trail-to-length-le$)
apply (*subgoal-tac Suc (length list) - length F = Suc (length list - length F)*)
by (*auto simp add: reduce-trail-to-length-ne*)

lemma *in-get-all-ann-decomposition-trail-update-trail*[*simp*]:
assumes *H*: $(L \# M1, M2) \in set\ (get-all-ann-decomposition\ (trail\ S))$
shows $trail\ (reduce-trail-to\ M1\ S) = M1$
proof –
obtain *K* **where**
 $L: L = Decided\ K$
using *H* **by** (*cases L*) (*auto dest!: in-get-all-ann-decomposition-decided-or-empty*)
obtain *c* **where**
 $tr-S: trail\ S = c @ M2 @ L \# M1$
using *H* **by** *auto*
show *?thesis*
by (*rule reduce-trail-to-trail-tl-trail-decomp*[*of - c @ M2 K*])
 $(auto\ simp: tr-S\ L)$

qed

lemma *conflicting-cons-trail-conflicting*[simp]:
assumes *undefined-lit* (*trail* *S*) (*lit-of* *L*)
shows
 $\text{conflicting } (\text{cons-trail } L \ S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
using *assms* *conflicting-cons-trail*[of *L* *S*] *map-option-is-None* **by** *fastforce*+

lemma *conflicting-add-learned-cls-conflicting*[simp]:
 $\text{conflicting } (\text{add-learned-cls } C \ S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
by *fastforce*+

lemma *conflicting-update-backtrack-lvl*[simp]:
 $\text{conflicting } (\text{update-backtrack-lvl } k \ S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
using *map-option-is-None* *conflicting-update-backtrack-lvl*[of *k* *S*] **by** *fastforce*+

end — end of *state_W* locale

6.1.2 CDCL Rules

Because of the strategy we will later use, we distinguish propagate, conflict from the other rules

locale *conflict-driven-clause-learning_W* =
state_W
 — functions for the state:
 — access functions:
trail *init-clss* *learned-clss* *backtrack-lvl* *conflicting*
 — changing state:
cons-trail *tl-trail* *add-learned-cls* *remove-cls* *update-backtrack-lvl*
update-conflicting
 — get state:
init-state
for
trail :: '*st* ⇒ ('*v*, '*v* clause) ann-lits **and**
init-clss :: '*st* ⇒ '*v* clauses **and**
learned-clss :: '*st* ⇒ '*v* clauses **and**
backtrack-lvl :: '*st* ⇒ nat **and**
conflicting :: '*st* ⇒ '*v* clause option **and**

cons-trail :: ('*v*, '*v* clause) ann-lit ⇒ '*st* ⇒ '*st* **and**
tl-trail :: '*st* ⇒ '*st* **and**
add-learned-cls :: '*v* clause ⇒ '*st* ⇒ '*st* **and**
remove-cls :: '*v* clause ⇒ '*st* ⇒ '*st* **and**
update-backtrack-lvl :: nat ⇒ '*st* ⇒ '*st* **and**
update-conflicting :: '*v* clause option ⇒ '*st* ⇒ '*st* **and**

init-state :: '*v* clauses ⇒ '*st*
begin

inductive *propagate* :: '*st* ⇒ '*st* ⇒ bool **for** *S* :: '*st* **where**
propagate-rule: *conflicting* *S* = None ⇒
 $E \in \# \text{ clauses } S \Rightarrow$
 $L \in \# E \Rightarrow$
 $\text{trail } S \models_{\text{as}} \text{CNot } (E - \{\#L\}) \Rightarrow$
 $\text{undefined-lit } (\text{trail } S) \ L \Rightarrow$

$T \sim \text{cons-trail } (\text{Propagated } L \ E) \ S \implies$
 $\text{propagate } S \ T$

inductive-cases propagateE : $\text{propagate } S \ T$

inductive $\text{conflict} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
 conflict-rule :

$\text{conflicting } S = \text{None} \implies$
 $D \in \# \text{ clauses } S \implies$
 $\text{trail } S \models_{\text{as}} \text{CNot } D \implies$
 $T \sim \text{update-conflicting } (\text{Some } D) \ S \implies$
 $\text{conflict } S \ T$

inductive-cases conflictE : $\text{conflict } S \ T$

inductive $\text{backtrack} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
 backtrack-rule :

$\text{conflicting } S = \text{Some } D \implies$
 $L \in \# D \implies$
 $(\text{Decided } K \ \# \ M1, M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S)) \implies$
 $\text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S \implies$
 $\text{get-level } (\text{trail } S) \ L = \text{get-maximum-level } (\text{trail } S) \ D \implies$
 $\text{get-maximum-level } (\text{trail } S) \ (D - \{\#L\# \}) \equiv i \implies$
 $\text{get-level } (\text{trail } S) \ K = i + 1 \implies$
 $T \sim \text{cons-trail } (\text{Propagated } L \ D)$
 $\quad (\text{reduce-trail-to } M1$
 $\quad \quad (\text{add-learned-cls } D$
 $\quad \quad \quad (\text{update-backtrack-lvl } i$
 $\quad \quad \quad \quad (\text{update-conflicting } \text{None } S)))) \implies$
 $\text{backtrack } S \ T$

inductive-cases backtrackE : $\text{backtrack } S \ T$

thm backtrackE

inductive $\text{decide} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
 decide-rule :

$\text{conflicting } S = \text{None} \implies$
 $\text{undefined-lit } (\text{trail } S) \ L \implies$
 $\text{atm-of } L \in \text{atms-of-mm } (\text{init-clss } S) \implies$
 $T \sim \text{cons-trail } (\text{Decided } L) \ (\text{incr-lvl } S) \implies$
 $\text{decide } S \ T$

inductive-cases decideE : $\text{decide } S \ T$

inductive $\text{skip} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
 skip-rule :

$\text{trail } S = \text{Propagated } L \ C' \ \# \ M \implies$
 $\text{conflicting } S = \text{Some } E \implies$
 $-L \notin \# E \implies$
 $E \neq \{\#\} \implies$
 $T \sim \text{tl-trail } S \implies$
 $\text{skip } S \ T$

inductive-cases skipE : $\text{skip } S \ T$

$\text{get-maximum-level } (\text{Propagated } L \ (C + \{\#L\# \}) \ \# \ M) \ D = k \vee k = 0$ (that was in a previous

version of the book) is equivalent to *get-maximum-level* (*Propagated* $L (C + \{\#L\# \}) \# M$) $D = k$, when the structural invariants holds.

inductive *resolve* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S ::$ 'st **where**

resolve-rule: $\text{trail } S \neq [] \implies$

$\text{hd-trail } S = \text{Propagated } L \ E \implies$

$L \in \# \ E \implies$

$\text{conflicting } S = \text{Some } D' \implies$

$-L \in \# \ D' \implies$

$\text{get-maximum-level } (\text{trail } S) ((\text{remove1-mset } (-L) \ D')) = \text{backtrack-lvl } S \implies$

$T \sim \text{update-conflicting } (\text{Some } (\text{resolve-cls } L \ D' \ E))$

$(\text{tl-trail } S) \implies$

$\text{resolve } S \ T$

inductive-cases *resolveE*: $\text{resolve } S \ T$

inductive *restart* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S ::$ 'st **where**

restart: $\text{state } S = (M, N, U, k, \text{None}) \implies$

$\neg M \models_{\text{asm}} \text{clauses } S \implies$

$U' \subseteq \# \ U \implies$

$\text{state } T = ([], N, U', 0, \text{None}) \implies$

$\text{restart } S \ T$

inductive-cases *restartE*: $\text{restart } S \ T$

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:

$\text{conflicting } S = \text{None} \implies$

$C \in \# \ \text{learned-clss } S \implies$

$\neg(\text{trail } S) \models_{\text{asm}} \text{clauses } S \implies$

$C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \implies$

$C \notin \# \ \text{init-clss } S \implies$

$T \sim \text{remove-cls } C \ S \implies$

$\text{forget } S \ T$

inductive-cases *forgetE*: $\text{forget } S \ T$

inductive *cdcl_W-rf* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S ::$ 'st **where**

restart: $\text{restart } S \ T \implies \text{cdcl}_W\text{-rf } S \ T \mid$

forget: $\text{forget } S \ T \implies \text{cdcl}_W\text{-rf } S \ T$

inductive *cdcl_W-bj* :: 'st \Rightarrow 'st \Rightarrow bool **where**

skip: $\text{skip } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S' \mid$

resolve: $\text{resolve } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S' \mid$

backtrack: $\text{backtrack } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S'$

inductive-cases *cdcl_W-bjE*: $\text{cdcl}_W\text{-bj } S \ T$

inductive *cdcl_W-o* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S ::$ 'st **where**

decide: $\text{decide } S \ S' \implies \text{cdcl}_W\text{-o } S \ S' \mid$

bj: $\text{cdcl}_W\text{-bj } S \ S' \implies \text{cdcl}_W\text{-o } S \ S'$

inductive *cdcl_W-restart* :: 'st \Rightarrow 'st \Rightarrow bool **for** $S ::$ 'st **where**

propagate: $\text{propagate } S \ S' \implies \text{cdcl}_W\text{-restart } S \ S' \mid$

conflict: $\text{conflict } S \ S' \implies \text{cdcl}_W\text{-restart } S \ S' \mid$

other: $cdcl_W\text{-}o\ S\ S' \implies cdcl_W\text{-}restart\ S\ S' \mid$
rf: $cdcl_W\text{-}rf\ S\ S' \implies cdcl_W\text{-}restart\ S\ S'$

lemma *rtrancplp-propagate-is-rtrancplp-cdcl_W-restart*:
*propagate*** $S\ S' \implies cdcl_W\text{-}restart^{**}\ S\ S'$
apply (*induction rule*: *rtrancplp-induct*)
apply *simp*
apply (*frule propagate*)
using *rtrancplp-trans*[*of cdcl_W-restart*] **by** *blast*

inductive *cdcl_W* :: '*st* \Rightarrow '*st* \Rightarrow *bool* **for** *S* :: '*st* **where**
W-propagate: *propagate* *S* *S'* $\implies cdcl_W\ S\ S' \mid$
W-conflict: *conflict* *S* *S'* $\implies cdcl_W\ S\ S' \mid$
W-other: *cdcl_W-o* *S* *S'* $\implies cdcl_W\ S\ S'$

lemma *cdcl_W-cdcl_W-restart*:
 $cdcl_W\ S\ T \implies cdcl_W\text{-}restart\ S\ T$
by (*induction rule*: *cdcl_W.induct*) (*auto intro*: *cdcl_W-restart.intros*)

lemma *cdcl_W-restart-all-rules-induct*[*consumes 1, case-names propagate conflict forget restart decide skip resolve backtrack*]:

fixes *S* :: '*st*
assumes
cdcl_W-restart: *cdcl_W-restart* *S* *S'* **and**
propagate: $\bigwedge T. \text{propagate } S\ T \implies P\ S\ T$ **and**
conflict: $\bigwedge T. \text{conflict } S\ T \implies P\ S\ T$ **and**
forget: $\bigwedge T. \text{forget } S\ T \implies P\ S\ T$ **and**
restart: $\bigwedge T. \text{restart } S\ T \implies P\ S\ T$ **and**
decide: $\bigwedge T. \text{decide } S\ T \implies P\ S\ T$ **and**
skip: $\bigwedge T. \text{skip } S\ T \implies P\ S\ T$ **and**
resolve: $\bigwedge T. \text{resolve } S\ T \implies P\ S\ T$ **and**
backtrack: $\bigwedge T. \text{backtrack } S\ T \implies P\ S\ T$
shows $P\ S\ S'$
using *assms*(1)
proof (*induct* *S'* *rule*: *cdcl_W-restart.induct*)
case (*propagate* *S'*) **note** *propagate = this*(1)
then show ?*case* **using** *assms*(2) **by** *auto*
next
case (*conflict* *S'*)
then show ?*case* **using** *assms*(3) **by** *auto*
next
case (*other* *S'*)
then show ?*case*
proof (*induct rule*: *cdcl_W-o.induct*)
case (*decide* *U*)
then show ?*case* **using** *assms*(6) **by** *auto*
next
case (*bj* *S'*)
then show ?*case* **using** *assms*(7–9) **by** (*induction rule*: *cdcl_W-bj.induct*) *auto*
qed
next
case (*rf* *S'*)
then show ?*case*
by (*induct rule*: *cdcl_W-rf.induct*) (*fast dest*: *forget restart*) +
qed

lemma *cdcl_W-restart-all-induct*[consumes 1, case-names propagate conflict forget restart decide skip
 resolve backtrack]:
fixes $S :: 'st$
assumes
cdcl_W-restart: *cdcl_W-restart* $S S'$ **and**
propagateH: $\bigwedge C L T. \text{conflicting } S = \text{None} \implies$
 $C \in \# \text{ clauses } S \implies$
 $L \in \# C \implies$
 $\text{trail } S \models_{\text{as}} \text{CNot } (\text{remove1-mset } L C) \implies$
 $\text{undefined-lit } (\text{trail } S) L \implies$
 $T \sim \text{cons-trail } (\text{Propagated } L C) S \implies$
 $P S T$ **and**
conflictH: $\bigwedge D T. \text{conflicting } S = \text{None} \implies$
 $D \in \# \text{ clauses } S \implies$
 $\text{trail } S \models_{\text{as}} \text{CNot } D \implies$
 $T \sim \text{update-conflicting } (\text{Some } D) S \implies$
 $P S T$ **and**
forgetH: $\bigwedge C T. \text{conflicting } S = \text{None} \implies$
 $C \in \# \text{ learned-clss } S \implies$
 $\neg(\text{trail } S) \models_{\text{asm}} \text{clauses } S \implies$
 $C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \implies$
 $C \notin \# \text{ init-clss } S \implies$
 $T \sim \text{remove-clss } C S \implies$
 $P S T$ **and**
restartH: $\bigwedge T U. \neg \text{trail } S \models_{\text{asm}} \text{clauses } S \implies$
 $\text{conflicting } S = \text{None} \implies$
 $\text{state } T = ([], \text{init-clss } S, U, 0, \text{None}) \implies$
 $U \subseteq \# \text{ learned-clss } S \implies$
 $P S T$ **and**
decideH: $\bigwedge L T. \text{conflicting } S = \text{None} \implies$
 $\text{undefined-lit } (\text{trail } S) L \implies$
 $\text{atm-of } L \in \text{atms-of-mm } (\text{init-clss } S) \implies$
 $T \sim \text{cons-trail } (\text{Decided } L) (\text{incr-lvl } S) \implies$
 $P S T$ **and**
skipH: $\bigwedge L C' M E T.$
 $\text{trail } S = \text{Propagated } L C' \# M \implies$
 $\text{conflicting } S = \text{Some } E \implies$
 $\neg L \notin \# E \implies E \neq \{\#\} \implies$
 $T \sim \text{tl-trail } S \implies$
 $P S T$ **and**
resolveH: $\bigwedge L E M D T.$
 $\text{trail } S = \text{Propagated } L E \# M \implies$
 $L \in \# E \implies$
 $\text{hd-trail } S = \text{Propagated } L E \implies$
 $\text{conflicting } S = \text{Some } D \implies$
 $\neg L \in \# D \implies$
 $\text{get-maximum-level } (\text{trail } S) ((\text{remove1-mset } (\neg L) D)) = \text{backtrack-lvl } S \implies$
 $T \sim \text{update-conflicting}$
 $(\text{Some } (\text{resolve-clss } L D E)) (\text{tl-trail } S) \implies$
 $P S T$ **and**
backtrackH: $\bigwedge L D K i M1 M2 T.$
 $\text{conflicting } S = \text{Some } D \implies$
 $L \in \# D \implies$
 $(\text{Decided } K \# M1, M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S)) \implies$
 $\text{get-level } (\text{trail } S) L = \text{backtrack-lvl } S \implies$
 $\text{get-level } (\text{trail } S) L = \text{get-maximum-level } (\text{trail } S) D \implies$

```

    get-maximum-level (trail S) (remove1-mset L D)  $\equiv i \implies$ 
    get-level (trail S) K = i+1  $\implies$ 
    T  $\sim$  cons-trail (Propagated L D)
      (reduce-trail-to M1
        (add-learned-cls D
          (update-backtrack-lvl i
            (update-conflicting None S))))  $\implies$ 
    P S T
  shows P S S'
  using cdclW-restart
proof (induct S S' rule: cdclW-restart-all-rules-induct)
  case (propagate S')
  then show ?case
    by (auto elim!: propagateE intro!: propagateH)
next
  case (conflict S')
  then show ?case
    by (auto elim!: conflictE intro!: conflictH)
next
  case (restart S')
  then show ?case
    by (auto elim!: restartE intro!: restartH)
next
  case (decide T)
  then show ?case
    by (auto elim!: decideE intro!: decideH)
next
  case (backtrack S')
  then show ?case by (auto elim!: backtrackE intro!: backtrackH
    simp del: state-simp simp add: state-eq-def)
next
  case (forget S')
  then show ?case by (auto elim!: forgetE intro!: forgetH)
next
  case (skip S')
  then show ?case by (auto elim!: skipE intro!: skipH)
next
  case (resolve S')
  then show ?case
    by (cases trail S) (auto elim!: resolveE intro!: resolveH)
qed

```

lemma *cdcl_W-o-induct*[consumes 1, case-names decide skip resolve backtrack]:
fixes S :: 'st
assumes cdcl_W-restart: cdcl_W-o S T **and**
 decideH: $\bigwedge L$ T. conflicting S = None \implies undefined-lit (trail S) L
 \implies atm-of L \in atms-of-mm (init-clss S)
 \implies T \sim cons-trail (Decided L) (incr-lvl S)
 \implies P S T **and**
 skipH: $\bigwedge L$ C' M E T.
 trail S = Propagated L C' # M \implies
 conflicting S = Some E \implies
 $-L \notin \# E \implies E \neq \{\#\} \implies$
 T \sim tl-trail S \implies
 P S T **and**
 resolveH: $\bigwedge L$ E M D T.

```

trail S = Propagated L E # M ==>
L ∈# E ==>
hd-trail S = Propagated L E ==>
conflicting S = Some D ==>
-L ∈# D ==>
get-maximum-level (trail S) ((remove1-mset (-L) D)) = backtrack-lvl S ==>
T ~ update-conflicting
  (Some (resolve-cls L D E)) (tl-trail S) ==>
P S T and
backtrackH: ∧ L D K i M1 M2 T.
  conflicting S = Some D ==>
  L ∈# D ==>
  (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (trail S)) ==>
  get-level (trail S) L = backtrack-lvl S ==>
  get-level (trail S) L = get-maximum-level (trail S) D ==>
  get-maximum-level (trail S) (remove1-mset L D) ≡ i ==>
  get-level (trail S) K = i + 1 ==>
  T ~ cons-trail (Propagated L D)
    (reduce-trail-to M1
      (add-learned-cls D
        (update-backtrack-lvl i
          (update-conflicting None S)))) ==>
P S T
shows P S T
using cdclW-restart apply (induct T rule: cdclW-o.induct)
  using assms(2) apply (auto elim: decideE)[1]
apply (elim cdclW-bjE skipE resolveE backtrackE)
  apply (frule skipH; simp)
  apply (cases trail S; auto elim!: resolveE intro!: resolveH)
apply (frule backtrackH; simp)
done

lemma cdclW-o-all-rules-induct[consumes 1, case-names decide backtrack skip resolve]:
  fixes S T :: 'st
  assumes
    cdclW-o S T and
    ∧ T. decide S T ==> P S T and
    ∧ T. backtrack S T ==> P S T and
    ∧ T. skip S T ==> P S T and
    ∧ T. resolve S T ==> P S T
  shows P S T
  using assms by (induct T rule: cdclW-o.induct) (auto simp: cdclW-bj.simps)

lemma cdclW-o-rule-cases[consumes 1, case-names decide backtrack skip resolve]:
  fixes S T :: 'st
  assumes
    cdclW-o S T and
    decide S T ==> P and
    backtrack S T ==> P and
    skip S T ==> P and
    resolve S T ==> P
  shows P
  using assms by (auto simp: cdclW-o.simps cdclW-bj.simps)

```

6.1.3 Structural Invariants

Properties of the trail

We here establish that:

- the consistency of the trail;
- the fact that there is no duplicate in the trail.

lemma *backtrack-lit-skipped*:

assumes

L: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**

M1: (*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**

no-dup: *no-dup* (*trail S*) **and**

bt-l: *backtrack-lvl S* = *length* (*filter is-decided* (*trail S*)) **and**

lev-K: *get-level* (*trail S*) *K* = *i* + 1

shows *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*

proof (*rule ccontr*)

let ?*M* = *trail S*

assume *L-in-M1*: ¬*atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*

obtain *c* **where**

Mc: *trail S* = *c* @ *M2* @ *Decided K* # *M1*

using *M1* **by** *blast*

have *atm-of L* ∉ *atm-of* ‘ *lits-of-l c* **and** *atm-of L* ∉ *atm-of* ‘ *lits-of-l M2* **and**

atm-of L ≠ *atm-of K* **and** *Kc*: *atm-of K* ∉ *atm-of* ‘ *lits-of-l c* **and**

KM2: *atm-of K* ∉ *atm-of* ‘ *lits-of-l M2*

using *L-in-M1 no-dup unfolding Mc lits-of-def* **by** *force+*

then have *g-M-eq-g-M1*: *get-level* ?*M L* = *get-level M1 L*

using *L-in-M1 unfolding Mc* **by** *auto*

then have *get-level M1 L* < *Suc i*

using *count-decided-ge-get-level[of L M1] KM2 lev-K Kc unfolding Mc*

by (*auto simp del: count-decided-ge-get-level*)

moreover have *Suc i* ≤ *backtrack-lvl S* **using** *bt-l KM2 lev-K Kc unfolding Mc* **by** (*simp add: Mc*)

ultimately show *False* **using** *L g-M-eq-g-M1* **by** *auto*

qed

lemma *cdcl_W-restart-distinctinv-1*:

assumes

cdcl_W-restart S S' **and**

no-dup (*trail S*) **and**

bt-lev: *backtrack-lvl S* = *count-decided* (*trail S*)

shows *no-dup* (*trail S'*)

using *assms*

proof (*induct rule: cdcl_W-restart-all-induct*)

case (*backtrack L D K i M1 M2 T*) **note** *decomp* = *this(3)* **and** *L* = *this(4)* **and** *lev-K* = *this(7)*

and

T = *this(8)* **and** *n-d* = *this(9)*

obtain *c* **where** *Mc*: *trail S* = *c* @ *M2* @ *Decided K* # *M1*

using *decomp* **by** *auto*

have *no-dup* (*M2* @ *Decided K* # *M1*)

using *Mc n-d* **by** *fastforce*

moreover have *atm-of L* ∉ *atm-of* ‘ *lits-of-l M1*

using *backtrack-lit-skipped[of L S K M1 M2 i] L decomp lev-K n-d bt-lev* **by** *fast*

moreover then have *undefined-lit M1 L*

by (*simp add: defined-lit-map lits-of-def image-image*)

ultimately show ?case **using** *decomp* *T n-d* **by** (*simp* *add: lits-of-def image-image*)
qed (*auto simp: defined-lit-map*)

Item 1 page 81 of Weidenbach's book

lemma *cdcl_W-restart-consistent-inv-2*:

assumes

cdcl_W-restart *S S'* **and**

no-dup (*trail S*) **and**

backtrack-lvl S = *count-decided* (*trail S*)

shows *consistent-interp* (*lits-of-l* (*trail S'*))

using *cdcl_W-restart-distinctinv-1* [*OF assms*] *distinct-consistent-interp* **by** *fast*

lemma *cdcl_W-o-bt*:

assumes

cdcl_W-o *S S'* **and**

backtrack-lvl S = *count-decided* (*trail S*) **and**

n-d[*simp*]: *no-dup* (*trail S*)

shows *backtrack-lvl S'* = *count-decided* (*trail S'*)

using *assms*

proof (*induct rule: cdcl_W-o-induct*)

case (*backtrack L D K i M1 M2 T*) **note** *decomp* = *this*(3) **and** *levK* = *this*(7) **and** *T* = *this*(8)

and

level = *this*(9)

have [*simp*]: *trail* (*reduce-trail-to M1 S*) = *M1*

using *decomp* **by** *auto*

obtain *c* **where** *M*: *trail S* = *c @ M2 @ Decided K # M1* **using** *decomp* **by** *auto*

moreover **have** *atm-of L* \notin *atm-of* ' *lits-of-l M1*

using *backtrack-lit-skipped*[*of L S K M1 M2 i*] *backtrack*(4,8,9) *levK decomp*

by (*fastforce simp add: lits-of-def*)

moreover **then** **have** *undefined-lit M1 L*

by (*simp add: defined-lit-map lits-of-def image-image*)

moreover

have *atm-of K* \notin *atm-of* ' *lits-of-l M1* **and** *atm-of K* \notin *atm-of* ' *lits-of-l c*

and *atm-of K* \notin *atm-of* ' *lits-of-l M2*

using *T n-d levK unfolding M* **by** (*auto simp: lits-of-def*)

ultimately show ?case

using *T levK unfolding M* **by** (*auto dest!: append-cons-eq-upt-length*)

qed *auto*

lemma *cdcl_W-rf-bt*:

assumes

cdcl_W-rf *S S'* **and**

backtrack-lvl S = *count-decided* (*trail S*)

shows *backtrack-lvl S'* = *count-decided* (*trail S'*)

using *assms* **by** (*induct rule: cdcl_W-rf.induct*) (*auto elim: restartE forgetE*)

Item 7 page 81 of Weidenbach's book

lemma *cdcl_W-restart-bt*:

assumes

cdcl_W-restart *S S'* **and**

backtrack-lvl S = *count-decided* (*trail S*) **and**

no-dup (*trail S*)

shows *backtrack-lvl S'* = *count-decided* (*trail S'*)

using *assms* **by** (*induct rule: cdcl_W-restart.induct*) (*auto simp: cdcl_W-o-bt cdcl_W-rf-bt*
elim: conflictE propagateE)

We write $1 + \text{count-decided } (\text{trail } S)$ instead of $\text{backtrack-lvl } S$ to avoid non termination of rewriting.

definition $\text{cdcl}_W\text{-restart-M-level-inv} :: 'st \Rightarrow \text{bool}$ **where**

$\text{cdcl}_W\text{-restart-M-level-inv } S \longleftrightarrow$
 $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$
 $\wedge \text{no-dup } (\text{trail } S)$
 $\wedge \text{backtrack-lvl } S = \text{count-decided } (\text{trail } S)$

lemma $\text{cdcl}_W\text{-restart-M-level-inv-decomp}$:

assumes $\text{cdcl}_W\text{-restart-M-level-inv } S$

shows

$\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$ **and**

$\text{no-dup } (\text{trail } S)$

using *assms* **unfolding** $\text{cdcl}_W\text{-restart-M-level-inv-def}$ **by** *fastforce+*

lemma $\text{cdcl}_W\text{-restart-consistent-inv}$:

fixes $S S' :: 'st$

assumes

$\text{cdcl}_W\text{-restart } S S'$ **and**

$\text{cdcl}_W\text{-restart-M-level-inv } S$

shows $\text{cdcl}_W\text{-restart-M-level-inv } S'$

using *assms* $\text{cdcl}_W\text{-restart-consistent-inv-2}$ $\text{cdcl}_W\text{-restart-distinctinv-1}$ $\text{cdcl}_W\text{-restart-bt}$

unfolding $\text{cdcl}_W\text{-restart-M-level-inv-def}$ **by** *meson+*

lemma $\text{rtrancpl-cdcl}_W\text{-restart-consistent-inv}$:

assumes

$\text{cdcl}_W\text{-restart}^{**} S S'$ **and**

$\text{cdcl}_W\text{-restart-M-level-inv } S$

shows $\text{cdcl}_W\text{-restart-M-level-inv } S'$

using *assms* **by** (*induct rule: rtrancpl-induct*) (*auto intro: cdcl_W-restart-consistent-inv*)

lemma $\text{trancpl-cdcl}_W\text{-restart-consistent-inv}$:

assumes

$\text{cdcl}_W\text{-restart}^{++} S S'$ **and**

$\text{cdcl}_W\text{-restart-M-level-inv } S$

shows $\text{cdcl}_W\text{-restart-M-level-inv } S'$

using *assms* **by** (*induct rule: trancpl-induct*) (*auto intro: cdcl_W-restart-consistent-inv*)

lemma $\text{cdcl}_W\text{-restart-M-level-inv-S0-cdcl}_W\text{-restart[simp]}$:

$\text{cdcl}_W\text{-restart-M-level-inv } (\text{init-state } N)$

unfolding $\text{cdcl}_W\text{-restart-M-level-inv-def}$ **by** *auto*

lemma $\text{cdcl}_W\text{-restart-M-level-inv-get-level-le-backtrack-lvl}$:

assumes *inv*: $\text{cdcl}_W\text{-restart-M-level-inv } S$

shows $\text{get-level } (\text{trail } S) L \leq \text{backtrack-lvl } S$

using *inv* **unfolding** $\text{cdcl}_W\text{-restart-M-level-inv-def}$

by *simp*

lemma $\text{backtrack-ex-decomp}$:

assumes

M-l: $\text{cdcl}_W\text{-restart-M-level-inv } S$ **and**

i-S: $i < \text{backtrack-lvl } S$

shows $\exists K M1 M2. (\text{Decided } K \# M1, M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S)) \wedge$

$\text{get-level } (\text{trail } S) K = \text{Suc } i$

proof –

```

let ?M = trail S
have i < count-decided (trail S)
  using i-S M-l by (auto simp: cdclW-restart-M-level-inv-def)
then obtain c K c' where tr-S: trail S = c @ Decided K # c' and
  lev-K: get-level (trail S) K = Suc i
  using le-count-decided-decomp[of trail S i] M-l by (auto simp: cdclW-restart-M-level-inv-def)
obtain M1 M2 where (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (trail S))
  using Decided-cons-in-get-all-ann-decomposition-append-Decided-cons unfolding tr-S by fast
then show ?thesis using lev-K by blast
qed

```

```

lemma backtrack-lvl-backtrack-decrease:
  assumes inv: cdclW-restart-M-level-inv S and bt: backtrack S T
  shows backtrack-lvl T < backtrack-lvl S
  using inv bt le-count-decided-decomp[of trail S backtrack-lvl T]
  unfolding cdclW-restart-M-level-inv-def
  by (fastforce elim!: backtrackE dest!: get-all-ann-decomposition-exists-prepend
    simp: append-assoc[of - - -# -, symmetric] simp del: append-assoc)

```

Compatibility with $op \sim$

```

lemma propagate-state-eq-compatible:
  assumes
    propa: propagate S T and
    SS': S ~ S' and
    TT': T ~ T'
  shows propagate S' T'
proof -
  obtain C L where
    conf: conflicting S = None and
    C: C ∈# clauses S and
    L: L ∈# C and
    tr: trail S ⊨as CNot (remove1-mset L C) and
    undef: undefined-lit (trail S) L and
    T: T ~ cons-trail (Propagated L C) S
  using propa by (elim propagateE) auto

  have C': C ∈# clauses S'
  using SS' C
  by (auto simp: state-eq-def clauses-def simp del: state-simp)

  show ?thesis
  apply (rule propagate-rule[of - C])
  using state-eq-sym[of S S'] SS' conf C' L tr undef TT' T
  by (auto simp: state-eq-def simp del: state-simp)
qed

```

```

lemma conflict-state-eq-compatible:
  assumes
    confl: conflict S T and
    TT': T ~ T' and
    SS': S ~ S'
  shows conflict S' T'
proof -
  obtain D where
    conf: conflicting S = None and

```

```

  D: D ∈# clauses S and
  tr: trail S ⊨as CNot D and
  T: T ∼ update-conflicting (Some D) S
using confl by (elim conflictE) auto

have D': D ∈# clauses S'
  using D SS' by fastforce

show ?thesis
  apply (rule conflict-rule[of - D])
  using state-eq-sym[of S S'] SS' conf D' tr TT' T
  by (auto simp: state-eq-def simp del: state-simp)
qed

lemma backtrack-state-eq-compatible:
  assumes
    bt: backtrack S T and
    SS': S ∼ S' and
    TT': T ∼ T' and
    inv: cdclW-restart-M-level-inv S
  shows backtrack S' T'
proof -
  obtain D L K i M1 M2 where
    conf: conflicting S = Some D and
    L: L ∈# D and
    decomp: (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (trail S)) and
    lev: get-level (trail S) L = backtrack-lvl S and
    max: get-level (trail S) L = get-maximum-level (trail S) D and
    max-D: get-maximum-level (trail S) (remove1-mset L D) ≡ i and
    lev-K: get-level (trail S) K = Suc i and
    T: T ∼ cons-trail (Propagated L D)
    (reduce-trail-to M1
     (add-learned-cls D
      (update-backtrack-lvl i
       (update-conflicting None S))))
  using bt inv by (elim backtrackE) metis
  have D': conflicting S' = Some D
    using SS' conf by (cases conflicting S') auto

  have T': T' ∼ cons-trail (Propagated L D)
    (reduce-trail-to M1 (add-learned-cls D
      (update-backtrack-lvl i (update-conflicting None S'))))
  using TT' unfolding state-eq-def
  using decomp D' inv SS' T by (auto simp add: cdclW-restart-M-level-inv-def)

  show ?thesis
    apply (rule backtrack-rule[of - D])
    apply (rule D')
    using state-eq-sym[of S S'] TT' SS' D' conf L decomp lev max max-D T
    apply (auto simp: state-eq-def simp del: state-simp)
    using decomp SS' lev SS' max-D max T' lev-K by (auto simp: state-eq-def simp del: state-simp)
  qed

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)
by (rule decide-rule) (auto simp: state-eq-def clauses-def simp del: state-simp)

lemma skip-state-eq-compatible:
  assumes
    skip: skip  $S T$  and
     $SS': S \sim S'$  and
     $TT': T \sim T'$ 
  shows skip  $S' T'$ 
proof -
  obtain  $L C' M E$  where
    tr: trail  $S = \text{Propagated } L C' \# M$  and
    raw: conflicting  $S = \text{Some } E$  and
    L:  $-L \notin \# E$  and
    E:  $E \neq \{\#\}$  and
    T:  $T \sim \text{tl-trail } S$ 
  using skip by (elim skipE) simp
  obtain  $E'$  where  $E': \text{conflicting } S' = \text{Some } E'$ 
    using  $SS'$  raw by (cases conflicting  $S'$ ) (auto simp: state-eq-def simp del: state-simp)
  show ?thesis
    apply (rule skip-rule)
      using tr raw L E T  $SS'$  apply (auto simp: simp del: )[]
      using  $E'$  apply simp
      using  $E' SS' L$  raw E apply (auto simp: state-eq-def simp del: state-simp)[2]
      using T  $TT' SS'$  by (auto simp: state-eq-def simp del: state-simp)
qed

lemma resolve-state-eq-compatible:
  assumes
    res: resolve  $S T$  and
     $TT': T \sim T'$  and
     $SS': S \sim S'$ 
  shows resolve  $S' T'$ 
proof -
  obtain  $E D L$  where
    tr: trail  $S \neq []$  and
    hd: hd-trail  $S = \text{Propagated } L E$  and
    L:  $L \in \# E$  and
    raw: conflicting  $S = \text{Some } D$  and
    LD:  $-L \in \# D$  and
    i: get-maximum-level (trail  $S$ ) ((remove1-mset  $(-L) D$ )) = backtrack-lvl  $S$  and
    T:  $T \sim \text{update-conflicting } (\text{Some } (\text{resolve-cls } L D E)) (\text{tl-trail } S)$ 
  using assms by (elim resolveE) simp

  obtain  $D'$  where
     $D': \text{conflicting } S' = \text{Some } D'$ 
    using  $SS'$  raw by fastforce
  have [simp]:  $D = D'$ 
    using  $D' SS'$  raw state-simp(5) by fastforce
  have  $T'T: T' \sim T$ 
    using  $TT'$  state-eq-sym by auto
  show ?thesis
    apply (rule resolve-rule)

```

```

    using tr SS' apply simp
    using hd SS' apply simp
    using L apply simp
    using D' apply simp
    using D' SS' raw LD apply (auto simp add: state-eq-def simp del: state-simp)[]
    using D' SS' raw LD apply (auto simp add: state-eq-def simp del: state-simp)[]
    using raw SS' i apply (auto simp add: state-eq-def simp del: state-simp)[]
    using T T'T SS' by (auto simp: state-eq-def simp del: state-simp)
qed

```

lemma *forget-state-eq-compatible*:

```

assumes
  forget: forget S T and
  SS': S ~ S' and
  TT': T ~ T'
shows forget S' T'
proof -
  obtain C where
    conf: conflicting S = None and
    C: C ∈# learned-clss S and
    tr: ¬(trail S) ⊨asm clauses S and
    C1: C ∉ set (get-all-mark-of-propagated (trail S)) and
    C2: C ∉# init-clss S and
    T: T ~ remove-cls C S
    using forget by (elim forgetE) simp

```

```

show ?thesis
  apply (rule forget-rule)
    using SS' conf apply simp
    using C SS' apply simp
    using SS' tr apply simp
    using SS' C1 apply simp
    using SS' C2 apply simp
    using T TT' SS' by (auto simp: state-eq-def simp del: state-simp)
qed

```

lemma *cdcl_W-restart-state-eq-compatible*:

```

assumes
  cdclW-restart S T and ¬restart S T and
  S ~ S'
  T ~ T' and
  cdclW-restart-M-level-inv S
shows cdclW-restart S' T'
using assms by (meson backtrack backtrack-state-eq-compatible bj cdclW-restart.simps cdclW-o-rule-cases
  cdclW-rf.cases conflict-state-eq-compatible decide decide-state-eq-compatible forget
  forget-state-eq-compatible propagate-state-eq-compatible resolve resolve-state-eq-compatible
  skip skip-state-eq-compatible state-eq-ref)

```

lemma *cdcl_W-bj-state-eq-compatible*:

```

assumes
  cdclW-bj S T and cdclW-restart-M-level-inv S
  T ~ T'
shows cdclW-bj S T'
using assms by (meson backtrack backtrack-state-eq-compatible cdclW-bjE resolve
  resolve-state-eq-compatible skip skip-state-eq-compatible state-eq-ref)

```

lemma *tranclp-cdcl_W-bj-state-eq-compatible*:
assumes
 $cdcl_W\text{-bj}^{++} S T$ **and** *inv*: $cdcl_W\text{-restart-M-level-inv } S$ **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows $cdcl_W\text{-bj}^{++} S' T'$
using *assms*
proof (*induction arbitrary: S' T'*)
case *base*
then show *?case*
unfolding *tranclp-unfold-end* **by** (*meson backtrack-state-eq-compatible cdcl_W-bj.simps*
resolve-state-eq-compatible rtranclp-unfold skip-state-eq-compatible)
next
case (*step T U*) **note** $IH = \text{this}(3)[OF \text{this}(4-5)]$
have $cdcl_W\text{-restart}^{++} S T$
using *tranclp-mono*[*of cdcl_W-bj cdcl_W-restart*] *step.hyps*(1) $cdcl_W\text{-restart.other } cdcl_W\text{-o.bj}$ **by** *blast*
then have $cdcl_W\text{-restart-M-level-inv } T$
using *inv tranclp-cdcl_W-restart-consistent-inv* **by** *blast*
then have $cdcl_W\text{-bj}^{++} T T'$
using $\langle U \sim T' \rangle$ *cdcl_W-bj-state-eq-compatible*[*of T U*] $\langle cdcl_W\text{-bj } T U \rangle$ **by** *auto*
then show *?case*
using $IH[of T]$ **by** *auto*
qed

Conservation of some Properties

lemma *cdcl_W-o-no-more-init-clss*:
assumes
 $cdcl_W\text{-o } S S'$ **and**
inv: $cdcl_W\text{-restart-M-level-inv } S$
shows $\text{init-clss } S = \text{init-clss } S'$
using *assms* **by** (*induct rule: cdcl_W-o-induct*) (*auto simp: inv cdcl_W-restart-M-level-inv-decomp*)

lemma *tranclp-cdcl_W-o-no-more-init-clss*:
assumes
 $cdcl_W\text{-o}^{++} S S'$ **and**
inv: $cdcl_W\text{-restart-M-level-inv } S$
shows $\text{init-clss } S = \text{init-clss } S'$
using *assms* **apply** (*induct rule: tranclp.induct*)
by (*auto dest: cdcl_W-o-no-more-init-clss*
dest!: tranclp-cdcl_W-restart-consistent-inv dest: tranclp-mono-explicit[*of cdcl_W-o - - cdcl_W-restart*]
simp: other)

lemma *rtranclp-cdcl_W-o-no-more-init-clss*:
assumes
 $cdcl_W\text{-o}^{**} S S'$ **and**
inv: $cdcl_W\text{-restart-M-level-inv } S$
shows $\text{init-clss } S = \text{init-clss } S'$
using *assms* **unfolding** *rtranclp-unfold* **by** (*auto intro: tranclp-cdcl_W-o-no-more-init-clss*)

lemma *cdcl_W-restart-init-clss*:
assumes
 $cdcl_W\text{-restart } S T$ **and**
inv: $cdcl_W\text{-restart-M-level-inv } S$
shows $\text{init-clss } S = \text{init-clss } T$
using *assms* **by** (*induction rule: cdcl_W-restart-all-induct*)

(*auto simp: inv cdcl_W-restart-M-level-inv-decomp not-in-iff*)

lemma *rtrancpl-cdcl_W-restart-init-clss:*

*cdcl_W-restart** S T \implies cdcl_W-restart-M-level-inv S \implies init-clss S = init-clss T*

by (*induct rule: rtrancpl-induct*) (*auto dest: cdcl_W-restart-init-clss rtrancpl-cdcl_W-restart-consistent-inv*)

lemma *trancpl-cdcl_W-restart-init-clss:*

cdcl_W-restart++ S T \implies cdcl_W-restart-M-level-inv S \implies init-clss S = init-clss T

using *rtrancpl-cdcl_W-restart-init-clss*[*of S T*] **unfolding** *rtrancpl-unfold* **by** *auto*

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.

definition *cdcl_W-restart-learned-clause* (*S :: 'st*) \longleftrightarrow

(*init-clss S \models_{psm} learned-clss S*

$\wedge (\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{init-clss } S \models_{pm} T)$

$\wedge \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \subseteq \text{set-mset } (\text{clauses } S))$

of Weidenbach's book for the initial state and some additional structural properties about the trail.

lemma *cdcl_W-restart-learned-clause-S0-cdcl_W-restart*[*simp*]:

cdcl_W-restart-learned-clause (*init-state N*)

unfolding *cdcl_W-restart-learned-clause-def* **by** *auto*

Item 4 page 81 of Weidenbach's book

lemma *cdcl_W-restart-learned-clss:*

assumes

cdcl_W-restart S S' and

learned: cdcl_W-restart-learned-clause S and

lev-inv: cdcl_W-restart-M-level-inv S

shows *cdcl_W-restart-learned-clause S'*

using *assms(1) lev-inv learned*

proof (*induct rule: cdcl_W-restart-all-induct*)

case (*backtrack K i M1 M2 L D T*) **note** *decomp = this(3) and confl = this(1) and lev-K = this(7) and*

undef = this(8) and T = this(9)

show *?case*

using *decomp confl learned undef T lev-K unfolding cdcl_W-restart-learned-clause-def*

by (*auto dest!: get-all-ann-decomposition-exists-prepend*

simp: clauses-def lev-inv cdcl_W-restart-M-level-inv-decomp dest: true-clss-clss-left-right)

next

case (*resolve L C M D*) **note** *trail = this(1) and CL = this(2) and confl = this(4) and DL = this(5) and lvl = this(6) and T = this(7)*

moreover

have *init-clss S \models_{psm} learned-clss S*

using *learned trail unfolding cdcl_W-restart-learned-clause-def clauses-def* **by** *auto*

then have *init-clss S \models_{pm} C + {#L#}*

using *trail learned unfolding cdcl_W-restart-learned-clause-def clauses-def*


```

    by (auto dest: true-clss-clss-in-imp-true-clss-clss)
  moreover have remove1-mset (- L) D + {#- L#} = D
    using DL by (auto simp: multiset-eq-iff)
  moreover have remove1-mset L C + {#L#} = C
    using CL by (auto simp: multiset-eq-iff)
  ultimately show ?case
    using learned T
  by (auto dest: mk-disjoint-insert
    simp add: cdclW-restart-learned-clause-def clauses-def
    intro!: true-clss-clss-union-mset-true-clss-clss-or-not-true-clss-clss-or[of - - L])
next
case (restart T)
then show ?case
  using learned
  by (auto
    simp: clauses-def state-eq-def cdclW-restart-learned-clause-def
    simp del: state-simp
    dest: true-clss-clssm-subsetE)
next
case propagate
then show ?case using learned by (auto simp: cdclW-restart-learned-clause-def)
next
case conflict
then show ?case using learned
  by (fastforce simp: cdclW-restart-learned-clause-def clauses-def
    true-clss-clss-in-imp-true-clss-clss)
next
case (forget U)
then show ?case using learned
  by (auto simp: cdclW-restart-learned-clause-def clauses-def split: if-split-asm)
qed (auto simp: cdclW-restart-learned-clause-def clauses-def)

lemma rtranclp-cdclW-restart-learned-clss:
  assumes
    cdclW-restart** S S' and
    cdclW-restart-M-level-inv S
    cdclW-restart-learned-clause S
  shows cdclW-restart-learned-clause S'
  using assms by induction (auto dest: cdclW-restart-learned-clss intro: rtranclp-cdclW-restart-consistent-inv)

```

No alien atom in the state

This invariant means that all the literals are in the set of clauses. These properties are implicit in Weidenbach's book.

definition *no-strange-atm* $S' \longleftrightarrow$ (
 $(\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S'))$
 $\wedge (\forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S') \longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S'))$
 $\wedge \text{atms-of-mm } (\text{learned-clss } S') \subseteq \text{atms-of-mm } (\text{init-clss } S')$
 $\wedge \text{atm-of } ' (\text{lits-of-l } (\text{trail } S')) \subseteq \text{atms-of-mm } (\text{init-clss } S')$)

lemma *no-strange-atm-decomp*:
 assumes *no-strange-atm* S
 shows *conflicting* S = *Some* T \implies *atms-of* T \subseteq *atms-of-mm* (*init-clss* S)
 and $(\forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S))$

$\rightarrow \text{atms-of mark} \subseteq \text{atms-of-mm (init-clss S)}$
and $\text{atms-of-mm (learned-clss S)} \subseteq \text{atms-of-mm (init-clss S)}$
and $\text{atm-of ' (lits-of-l (trail S))} \subseteq \text{atms-of-mm (init-clss S)}$
using *assms* **unfolding** *no-strange-atm-def* **by** *blast+*

lemma *no-strange-atm-S0 [simp]: no-strange-atm (init-state N)*
unfolding *no-strange-atm-def* **by** *auto*

lemma *in-atms-of-implies-atm-of-on-atms-of-ms:*
 $C + \{\#L\# \} \in \# A \implies x \in \text{atms-of } C \implies x \in \text{atms-of-mm } A$
using *multi-member-split* **by** *fastforce*

lemma *propagate-no-strange-atm-inv:*

assumes

propagate S T **and**

alien: no-strange-atm S

shows *no-strange-atm T*

using *assms(1)*

proof (*induction*)

case (*propagate-rule C L T*) **note** *confl = this(1)* **and** *C = this(2)* **and** *C-L = this(3)* **and**
 $tr = this(4)$ **and** *undef = this(5)* **and** *T = this(6)*

have *atm-CL: atms-of C* \subseteq *atms-of-mm (init-clss S)*

using *C alien* **unfolding** *no-strange-atm-def*

by (*auto simp: clauses-def atms-of-ms-def*)

show *?case*

unfolding *no-strange-atm-def*

proof (*intro conjI allI impI, goal-cases*)

case 1

then show *?case*

using *confl T undef* **by** *auto*

next

case (*2 L' mark'*)

then show *?case*

using *C-L T alien undef atm-CL* **unfolding** *no-strange-atm-def clauses-def* **by** (*auto 5 5*)

next

case (*3*)

show *?case* **using** *T alien undef* **unfolding** *no-strange-atm-def* **by** *auto*

next

case (*4*)

show *?case*

using *T alien undef C-L atm-CL* **unfolding** *no-strange-atm-def* **by** (*auto simp: atms-of-def*)

qed

qed

lemma *in-atms-of-remove1-mset-in-atms-of:*

$x \in \text{atms-of (remove1-mset L C)} \implies x \in \text{atms-of } C$

using *in-diffD* **unfolding** *atms-of-def* **by** *fastforce*

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI:*

$\text{atms-of-mm (learned-clss S)} \subseteq \text{atms-of-mm (init-clss S)} \implies$

$x \in \text{atms-of-mm (learned-clss T)} \implies$

$\text{learned-clss T} \subseteq \# \text{ learned-clss S} \implies$

$x \in \text{atms-of-mm (init-clss S)}$

by (*meson atms-of-ms-mono contra-subsetD set-mset-mono*)

lemma *cdcl_W-restart-no-strange-atm-explicit:*

assumes
cdcl_W-restart $S \ S'$ **and**
lev: cdcl_W-restart-M-level-inv S **and**
conf: $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S)$ **and**
decided: $\forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S)$
 $\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S)$ **and**
learned: $\text{atms-of-mm } (\text{learned-clss } S) \subseteq \text{atms-of-mm } (\text{init-clss } S)$ **and**
trail: $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{init-clss } S)$
shows
 $(\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S')) \wedge$
 $(\forall L \text{ mark. Propagated } L \text{ mark} \in \text{set } (\text{trail } S'))$
 $\longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S')) \wedge$
 $\text{atms-of-mm } (\text{learned-clss } S') \subseteq \text{atms-of-mm } (\text{init-clss } S') \wedge$
 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S')) \subseteq \text{atms-of-mm } (\text{init-clss } S')$
 $(\text{is } ?C \ S' \wedge ?M \ S' \wedge ?U \ S' \wedge ?V \ S')$
using *assms(1,2)*
proof (*induct rule: cdcl_W-restart-all-induct*)
case (*propagate* $C \ L \ T$) **note** *confl = this(1)* **and** *C-L = this(2)* **and** *tr = this(3)* **and** *undef = this(4)*
and $T = \text{this}(5)$
show *?case*
using *propagate-rule[OF propagate.hyps(1-3) - propagate.hyps(5,6), simplified]*
propagate.hyps(4) propagate-no-strange-atm-inv[of S T]
conf decided learned trail unfolding no-strange-atm-def by presburger
next
case (*decide* L)
then show *?case using learned decided conf trail unfolding clauses-def by auto*
next
case (*skip* $L \ C \ M \ D$)
then show *?case using learned decided conf trail by auto*
next
case (*conflict* $D \ T$) **note** *D-S = this(2)* **and** $T = \text{this}(4)$
have $D: \text{atm-of } ' \text{ set-mset } D \subseteq \bigcup (\text{atms-of } ' (\text{set-mset } (\text{clauses } S)))$
using *D-S by (auto simp add: atms-of-def atms-of-ms-def)*
moreover {
fix $xa :: 'v \text{ literal}$
assume $a1: \text{atm-of } ' \text{ set-mset } D \subseteq (\bigcup_{x \in \text{set-mset } (\text{init-clss } S)}. \text{atms-of } x)$
 $\cup (\bigcup_{x \in \text{set-mset } (\text{learned-clss } S)}. \text{atms-of } x)$
assume $a2:$
 $(\bigcup_{x \in \text{set-mset } (\text{learned-clss } S)}. \text{atms-of } x) \subseteq (\bigcup_{x \in \text{set-mset } (\text{init-clss } S)}. \text{atms-of } x)$
assume $xa \in \# \ D$
then have $\text{atm-of } xa \in \text{UNION } (\text{set-mset } (\text{init-clss } S)) \text{ atms-of}$
using $a2 \ a1$ **by** (*metis (no-types) Un-iff atm-of-lit-in-atms-of atms-of-def subset-Un-eq*)
then have $\exists m \in \text{set-mset } (\text{init-clss } S). \text{atm-of } xa \in \text{atms-of } m$
by *blast*
} **note** $H = \text{this}$
ultimately show *?case using conflict.premis T learned decided 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 decided conf trail*
by (*auto intro: atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*)
next
case (*forget* $C \ T$) **note** *confl = this(1)* **and** $C = \text{this}(4)$ **and** $C\text{-le} = \text{this}(5)$ **and**
 $T = \text{this}(6)$

```

have H:  $\bigwedge L$  mark. Propagated L mark  $\in$  set (trail S)  $\implies$  atms-of mark  $\subseteq$  atms-of-mm (init-cls S)
  using decided by simp
show ?case unfolding clauses-def apply (intro conjI)
  using conf confl T trail C unfolding clauses-def apply (auto dest!: H)[]
  using T trail C C-le apply (auto dest!: H)[]
  using T learned C-le atms-of-ms-remove-subset[of set-mset (learned-cls S)] apply auto[]
  using T trail C-le apply (auto simp: clauses-def lits-of-def)[]
done
next
case (backtrack L D K i M1 M2 T) note confl = this(1) and LD = this(2) and decomp = this(3)
and
  lev-K = this(7) and T = this(8)
have ?C T
  using conf T decomp lev lev-K by (auto simp: cdclW-restart-M-level-inv-decomp)
moreover have set M1  $\subseteq$  set (trail S)
  using decomp by auto
then have M: ?M T
  using decided conf confl T decomp lev lev-K
  by (auto simp: image-subset-iff clauses-def cdclW-restart-M-level-inv-decomp)
moreover have ?U T
  using learned decomp conf confl T lev lev-K unfolding clauses-def
  by (auto simp: cdclW-restart-M-level-inv-decomp)
moreover have ?V T
  using M conf confl trail T decomp lev LD lev-K
  by (auto simp: cdclW-restart-M-level-inv-decomp atms-of-def
    dest!: get-all-ann-decomposition-exists-prepend)
ultimately show ?case by blast
next
case (resolve L C M D T) note trail-S = this(1) and confl = this(4) and T = this(7)
let ?T = update-conflicting (Some (resolve-cls L D C)) (tl-trail S)
have ?C ?T
  using confl trail-S conf decided by (auto dest!: in-atms-of-remove1-mset-in-atms-of)
moreover have ?M ?T
  using confl trail-S conf decided by auto
moreover have ?U ?T
  using trail learned by auto
moreover have ?V ?T
  using confl trail-S trail by auto
ultimately show ?case using T by simp
qed

```

lemma *cdcl_W-restart-no-strange-atm-inv*:

```

assumes cdclW-restart S S' and no-strange-atm S and cdclW-restart-M-level-inv S
shows no-strange-atm S'
using cdclW-restart-no-strange-atm-explicit[OF assms(1)] assms(2,3) unfolding no-strange-atm-def
by fast

```

lemma *rtranclp-cdcl_W-restart-no-strange-atm-inv*:

```

assumes cdclW-restart** S S' and no-strange-atm S and cdclW-restart-M-level-inv S
shows no-strange-atm S'
using assms by induction (auto intro: cdclW-restart-no-strange-atm-inv rtranclp-cdclW-restart-consistent-inv)

```

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 also. Remark that we will show later that there cannot be duplicate *clause*.

definition *distinct-cdcl_W-restart-state* ($S :: 'st$)

$\longleftrightarrow ((\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{distinct-mset } T)$
 $\wedge \text{distinct-mset-mset } (\text{learned-clss } S)$
 $\wedge \text{distinct-mset-mset } (\text{init-clss } S)$
 $\wedge (\forall L \text{ mark. } (\text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset } \text{mark})))$

lemma *distinct-cdcl_W-restart-state-decomp*:

assumes *distinct-cdcl_W-restart-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**
 $\forall L \text{ mark. } (\text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset } \text{mark})$

using *assms unfolding distinct-cdcl_W-restart-state-def* **by** *blast+*

lemma *distinct-cdcl_W-restart-state-decomp-2*:

assumes *distinct-cdcl_W-restart-state* ($S :: 'st$) **and** *conflicting* $S = \text{Some } T$

shows *distinct-mset* T

using *assms unfolding distinct-cdcl_W-restart-state-def* **by** *auto*

lemma *distinct-cdcl_W-restart-state-S0-cdcl_W-restart[simp]*:

distinct-mset-mset $N \implies \text{distinct-cdcl}_W\text{-restart-state } (\text{init-state } N)$

unfolding *distinct-cdcl_W-restart-state-def* **by** *auto*

lemma *distinct-cdcl_W-restart-state-inv*:

assumes

cdcl_W-restart $S \ S'$ **and**
lev-inv: *cdcl_W-restart-M-level-inv* S **and**
distinct-cdcl_W-restart-state S

shows *distinct-cdcl_W-restart-state* S'

using *assms(1,2,2,3)*

proof (*induct rule*: *cdcl_W-restart-all-induct*)

case (*backtrack* $L \ D \ K \ i \ M1 \ M2$)

then show *?case*

using *lev-inv unfolding distinct-cdcl_W-restart-state-def*

by (*auto dest*: *get-all-ann-decomposition-incl simp*: *cdcl_W-restart-M-level-inv-decomp*)

next

case *restart*

then show *?case*

unfolding *distinct-cdcl_W-restart-state-def distinct-mset-set-def clauses-def* **by** *auto*

next

case *resolve*

then show *?case*

by (*auto simp add*: *distinct-cdcl_W-restart-state-def distinct-mset-set-def clauses-def*
distinct-mset-single-add

intro!: *distinct-mset-union-mset*)

qed (*auto simp*: *distinct-cdcl_W-restart-state-def distinct-mset-set-def clauses-def*

dest!: *in-diffD*)

lemma *rtanclp-distinct-cdcl_W-restart-state-inv*:

assumes
*cdcl_W-restart** S S' and*
cdcl_W-restart-M-level-inv S and
distinct-cdcl_W-restart-state S
shows *distinct-cdcl_W-restart-state S'*
using *assms apply (induct rule: rtrancpl-induct)*
using *distinct-cdcl_W-restart-state-inv rtrancpl-cdcl_W-restart-consistent-inv by blast+*

Conflicts and Annotations

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-restart-conflicting S \longleftrightarrow*
 $(\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot \ T)$
 $\wedge \text{every-mark-is-a-conflict } S$

lemma *backtrack-atms-of-D-in-M1:*

fixes *M1 :: ('v, 'v clause) ann-lits*

assumes

inv: cdcl_W-restart-M-level-inv S and
i: get-maximum-level (trail S) ((remove1-mset L D)) \equiv i and
decomp: (Decided K # M1, M2)
 $\in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S)) \text{ and}$
S-lvl: backtrack-lvl S = get-maximum-level (trail S) D and
S-conf: conflicting S = Some D and
lev-K: get-level (trail S) K = Suc i and
T: T \sim cons-trail (Propagated L D)
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } D$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting None } S)))) \text{ and}$

confl: $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot \ T$

shows *atms-of ((remove1-mset L D)) \subseteq atm-of ' lits-of-l (tl (trail T))*

proof (rule ccontr)

let *?k = get-maximum-level (trail S) D*

let *?D' = remove1-mset L D*

have *trail S $\models_{as} CNot \ D$ using confl S-conf by auto*

then have *vars-of-D: atms-of D \subseteq atm-of ' lits-of-l (trail S) unfolding atms-of-def*
by (meson image-subsetI true-annots-CNot-all-atms-defined)

obtain *M0 where M: trail S = M0 @ M2 @ Decided K # M1*

using *decomp by auto*

have *max: ?k = count-decided (M0 @ M2 @ Decided K # M1)*

using *inv unfolding cdcl_W-restart-M-level-inv-def S-lvl M by simp*

assume *a: \neg ?thesis*

then obtain *L' where*

L': L' \in atms-of ?D' and

L'-notin-M1: L' \notin atm-of ' lits-of-l M1

using *T decomp inv by (auto simp: cdcl_W-restart-M-level-inv-decomp)*

then have $L'\text{-in}: L' \in \text{atm-of } \text{'lits-of-l } (M0 @ M2 @ \text{Decided } K \# [])$
 using $\text{vars-of-}D$ **unfolding** M by $(\text{auto dest: in-atms-of-remove1-mset-in-atms-of})$
 then obtain L'' where
 $L'' \in \# ?D'$ and
 $L'': L' = \text{atm-of } L''$
 using $L' L'\text{-notin-}M1$ **unfolding** atms-of-def by auto
 have $\text{atm-of } K \notin \text{atm-of } \text{'lits-of-l } (M0 @ M2)$
 using inv by $(\text{auto simp: cdcl}_W\text{-restart-}M\text{-level-inv-def } M \text{ lits-of-def})$
 then have $\text{count-decided } M1 = i$
 using $\text{lev-}K$ **unfolding** M by $(\text{auto simp: image-Un})$
 then have $\text{lev-}L''$:
 $\text{get-level } (\text{trail } S) L'' = \text{get-level } (M0 @ M2 @ \text{Decided } K \# []) L'' + i$
 using $L'\text{-notin-}M1$ L'' $\text{get-rev-level-skip-end}[OF L'\text{-in}[\text{unfolded } L''], \text{ of } M1]$ M by auto
 moreover
 consider
 $(M0) L' \in \text{atm-of } \text{'lits-of-l } M0 \mid$
 $(M2) L' \in \text{atm-of } \text{'lits-of-l } M2 \mid$
 $(K) L' = \text{atm-of } K$
 using $\text{inv } L'\text{-in}$ **unfolding** L'' by $(\text{auto simp: cdcl}_W\text{-restart-}M\text{-level-inv-def})$
 then have $\text{get-level } (M0 @ M2 @ \text{Decided } K \# []) L'' \geq \text{Suc } 0$
proof cases
 case $M0$
 then have $L' \neq \text{atm-of } K$
 using $\text{inv } \langle \text{atm-of } K \notin \text{atm-of } \text{'lits-of-l } (M0 @ M2) \rangle$ **unfolding** L'' by auto
 then show $?thesis$ using $M0$ **unfolding** L'' by auto
 next
 case $M2$
 then have $L' \notin \text{atm-of } \text{'lits-of-l } (M0 @ \text{Decided } K \# [])$
 using $\text{inv } \langle \text{atm-of } K \notin \text{atm-of } \text{'lits-of-l } (M0 @ M2) \rangle$ **unfolding** L''
 by $(\text{auto simp: } M \text{ cdcl}_W\text{-restart-}M\text{-level-inv-def atm-lit-of-set-lits-of-l})$
 then show $?thesis$ using $M2$ **unfolding** L'' by $(\text{auto simp: image-Un})$
 next
 case K
 then have $L' \notin \text{atm-of } \text{'lits-of-l } (M0 @ M2)$
 using inv **unfolding** L'' by $(\text{auto simp: cdcl}_W\text{-restart-}M\text{-level-inv-def atm-lit-of-set-lits-of-l } M)$
 then show $?thesis$ using K **unfolding** L'' by $(\text{auto simp: image-Un})$
 qed
 ultimately have $\text{get-level } (\text{trail } S) L'' \geq i + 1$
 using $\text{lev-}L''$ **unfolding** M by simp
 then have $\text{get-maximum-level } (\text{trail } S) ?D' \geq i + 1$
 using $\text{get-maximum-level-ge-get-level}[OF \langle L'' \in \# ?D' \rangle, \text{ of trail } S]$ by auto
 then show False using i by auto
 qed

lemma *distinct-atms-of-incl-not-in-other*:

assumes

$a1$: $\text{no-dup } (M @ M')$ **and**

$a2$: $\text{atms-of } D \subseteq \text{atm-of } \text{'lits-of-l } M'$ **and**

$a3$: $x \in \text{atms-of } D$

shows $x \notin \text{atm-of } \text{'lits-of-l } M$

proof –

have $\text{ff1}: \bigwedge l \text{ ms. } \text{undefined-lit } ms \ l \vee \text{atm-of } l$

$\in \text{set } (\text{map } (\lambda m. \text{atm-of } (\text{lit-of } (m :: ('a, 'b) \text{ ann-lit}))) \text{ ms})$

by $(\text{simp add: defined-lit-map})$

have $\text{ff2}: \bigwedge a. a \notin \text{atms-of } D \vee a \in \text{atm-of } \text{'lits-of-l } M'$

using $a2$ by (meson subsetCE)

```

have ff3:  $\bigwedge a. a \notin \text{set } (\text{map } (\lambda m. \text{atm-of } (\text{lit-of } m)) M') \wedge a \notin \text{set } (\text{map } (\lambda m. \text{atm-of } (\text{lit-of } m)) M)$ 
  using a1 by (metis (lifting) IntI distinct-append empty-iff map-append)
have  $\forall L a f. \exists l. ((a::'a) \notin f \wedge L \vee (l::'a \text{ literal}) \in L) \wedge (a \notin f \wedge L \vee f l = a)$ 
  by blast
then show  $x \notin \text{atm-of } ' \text{ lits-of-l } M$ 
  using ff3 ff2 ff1 a3 by (metis (no-types) Decided-Propagated-in-iff-in-lits-of-l)
qed

```

Item 5 page 81 of Weidenbach's book

lemma *cdcl_W-restart-propagate-is-conclusion:*

```

assumes
  cdclW-restart  $S S'$  and
  inv: cdclW-restart-M-level-inv  $S$  and
  decomp: all-decomposition-implies-m (init-clss  $S$ ) (get-all-ann-decomposition (trail  $S$ )) and
  learned: cdclW-restart-learned-clause  $S$  and
  conf:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{\text{as}} \text{CNot } T$  and
  alien: no-strange-atm  $S$ 
shows all-decomposition-implies-m (init-clss  $S'$ ) (get-all-ann-decomposition (trail  $S'$ ))
using assms(1,2)
proof (induct rule: cdclW-restart-all-induct)
  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  $T = \text{this}(7)$ 
  let ?decomp = get-all-ann-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-ann-decomposition  $M$ ))
      (auto simp:  $M$ )
next
  case (skip  $L C' M D$ )
  note tr = this(1) and  $T = \text{this}(5)$ 
  have  $M$ : set (get-all-ann-decomposition  $M$ )
    = insert (hd (get-all-ann-decomposition  $M$ )) (set (tl (get-all-ann-decomposition  $M$ )))
    by (cases get-all-ann-decomposition  $M$ ) auto
  show ?case
    using decomp tr  $T$  unfolding all-decomposition-implies-def
    by (cases hd (get-all-ann-decomposition  $M$ ))
      (auto simp add:  $M$ )
next
  case decide
  note  $S = \text{this}(1)$  and undef = this(2) and  $T = \text{this}(4)$ 
  show ?case using decomp  $T$  undef unfolding  $S$  all-decomposition-implies-def by auto
next
  case (propagate  $C L T$ )
  note propa = this(2) and  $L = \text{this}(3)$  and undef = this(5) and  $T = \text{this}(6)$ 
  obtain  $a y$  where ay: hd (get-all-ann-decomposition (trail  $S$ )) = ( $a, y$ )
    by (cases hd (get-all-ann-decomposition (trail  $S$ )))
  then have  $M$ : trail  $S = y @ a$  using get-all-ann-decomposition-decomp by blast
  have  $M'$ : set (get-all-ann-decomposition (trail  $S$ ))

```



```

= insert (a, y) (set (tl (get-all-ann-decomposition (trail S))))
using ay by (cases get-all-ann-decomposition (trail S)) auto
have unmark-l a  $\cup$  set-mset (init-clss S)  $\models_{ps}$  unmark-l y
using decomp ay unfolding all-decomposition-implies-def
by (cases get-all-ann-decomposition (trail S)) fastforce+
then have a-Un-N-M: unmark-l a  $\cup$  set-mset (init-clss S)
 $\models_{ps}$  unmark-l (trail S)
unfolding M by (auto simp add: all-in-true-clss-clss image-Un)

have unmark-l a  $\cup$  set-mset (init-clss S)  $\models_p \{\#L\# \}$  (is ?I  $\models_p -$ )
proof (rule true-clss-clss-plus-CNot)
show ?I  $\models_p$  remove1-mset L C +  $\{\#L\# \}$ 
apply (rule true-clss-clss-in-imp-true-clss-clss[of -
  set-mset (init-clss S)  $\cup$  set-mset (learned-clss S)])
using learned propa L by (auto simp: clauses-def cdclW-restart-learned-clause-def
  true-annot-CNot-diff)
next
have unmark-l (trail S)  $\models_{ps}$  CNot (remove1-mset L C)
using  $\langle (trail S) \models_{as} CNot (remove1-mset L C) \rangle$  true-annots-true-clss-clss
by blast
then show ?I  $\models_{ps}$  CNot (remove1-mset L C)
using a-Un-N-M true-clss-clss-left-right true-clss-clss-union-l-r by blast
qed
moreover have  $\bigwedge aa b.$ 
 $\forall (Ls, seen) \in set (get-all-ann-decomposition (y @ a)).$ 
  unmark-l Ls  $\cup$  set-mset (init-clss S)  $\models_{ps}$  unmark-l seen  $\implies$ 
   $(aa, b) \in set (tl (get-all-ann-decomposition (y @ a))) \implies$ 
  unmark-l aa  $\cup$  set-mset (init-clss S)  $\models_{ps}$  unmark-l b
by (metis (no-types, lifting) case-prod-conv get-all-ann-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  $\langle unmark-l a \cup set-mset (init-clss S) \models_{ps} unmark-l y \rangle$ 
ay by auto
next
case (backtrack L D K i M1 M2 T) note conf = this(1) and LD = this(2) and decomp' = this(3)
and
  lev-L = this(4) and lev-K = this(7) and undef = this(8) and T = this(9)
let ?D' = remove1-mset L D
have  $\forall l \in set M2. \neg is-decided l$ 
using get-all-ann-decomposition-snd-not-decided decomp' by blast
obtain M0 where M: trail S = M0 @ M2 @ Decided K # M1
using decomp' by auto
show ?case unfolding all-decomposition-implies-def
proof
fix x
assume x  $\in set (get-all-ann-decomposition (trail T))$ 
then have x: x  $\in set (get-all-ann-decomposition (Propagated L D \# M1))$ 
using T decomp' undef inv by (simp add: cdclW-restart-M-level-inv-decomp)
let ?m = get-all-ann-decomposition (Propagated L D # M1)
let ?hd = hd ?m
let ?tl = tl ?m
consider
  (hd) x = ?hd
  | (tl) x  $\in set ?tl$ 

```

```

using x by (cases ?m) auto
then show case x of (Ls, seen) ⇒ unmark-l Ls ∪ set-mset (init-clss T) ⊨ps unmark-l seen
proof cases
  case tl
  then have x ∈ set (get-all-ann-decomposition (trail S))
    using tl-get-all-ann-decomposition-skip-some[of x] by (simp add: list.set-sel(2) M)
  then show ?thesis
    using decomp learned decomp confl alien inv T undef M
    unfolding all-decomposition-implies-def cdclW-restart-M-level-inv-def
    by auto
next
case hd
obtain M1' M1'' where M1: hd (get-all-ann-decomposition M1) = (M1', M1'')
  by (cases hd (get-all-ann-decomposition M1))
then have x': x = (M1', Propagated L D # M1'')
  using ⟨x = ?hd⟩ by auto
have (M1', M1'') ∈ set (get-all-ann-decomposition (trail S))
  using M1[symmetric] hd-get-all-ann-decomposition-skip-some[OF M1[symmetric],
    of M0 @ M2] unfolding M by fastforce
then have 1: unmark-l M1' ∪ set-mset (init-clss S) ⊨ps unmark-l M1''
  using decomp unfolding all-decomposition-implies-def by auto

moreover
have vars-of-D: atms-of ?D' ⊆ atm-of ' lits-of-l M1
  using backtrack-atms-of-D-in-M1[of S D L i K M1 M2 T] backtrack.hyps inv conf confl
  by (auto simp: cdclW-restart-M-level-inv-decomp)
have no-dup (trail S) using inv by (auto simp: cdclW-restart-M-level-inv-decomp)
then have vars-in-M1:
  ∀ x ∈ atms-of ?D'. x ∉ atm-of ' lits-of-l (M0 @ M2 @ Decided K # [])
  using vars-of-D distinct-atms-of-incl-not-in-other[of
    M0 @ M2 @ Decided K # [] M1] unfolding M by auto
have trail S ⊨as CNot (remove1-mset L D)
  using conf confl LD unfolding M true-annots-true-cls-def-iff-negation-in-model
  by (auto dest!: Multiset.in-diffD)
then have M1 ⊨as CNot ?D'
  using vars-in-M1 true-annots-remove-if-notin-vars[of M0 @ M2 @ Decided K # []
    M1 CNot ?D'] conf confl unfolding M lits-of-def by simp
have M1 = M1'' @ M1' by (simp add: M1 get-all-ann-decomposition-decomp)
have TT: unmark-l M1' ∪ set-mset (init-clss S) ⊨ps CNot ?D'
  using true-annots-true-clss-cls[OF ⟨M1 ⊨as CNot ?D'⟩] true-clss-clss-left-right[OF 1]
  unfolding ⟨M1 = M1'' @ M1'⟩ by (auto simp add: inf-sup-aci(5,7))
have init-clss S ⊨pm ?D' + {#L#}
  using conf learned confl LD unfolding cdclW-restart-learned-clause-def by auto
then have T': unmark-l M1' ∪ set-mset (init-clss S) ⊨p ?D' + {#L#} by auto
have atms-of (?D' + {#L#}) ⊆ atms-of-mm (clauses S)
  using alien conf LD unfolding no-strange-atm-def clauses-def by auto
then have unmark-l M1' ∪ set-mset (init-clss S) ⊨p {#L#}
  using true-clss-cls-plus-CNot[OF T' TT] by auto

ultimately show ?thesis
  using T' T decomp' undef inv unfolding x' by (simp add: cdclW-restart-M-level-inv-decomp)
qed
qed
qed

```

lemma cdcl_W-restart-propagate-is-false:

```

assumes
  cdclW-restart  $S\ S'$  and
  lev: cdclW-restart-M-level-inv  $S$  and
  learned: cdclW-restart-learned-clause  $S$  and
  decomp: all-decomposition-implies-m (init-clss  $S$ ) (get-all-ann-decomposition (trail  $S$ )) and
  confl:  $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot\ T$  and
  alien: no-strange-atm  $S$  and
  mark-confl: every-mark-is-a-conflict  $S$ 
shows every-mark-is-a-conflict  $S'$ 
using assms(1,2)
proof (induct rule: cdclW-restart-all-induct)
  case (propagate  $C\ L\ T$ ) note  $LC = \text{this}(3)$  and  $\text{confl} = \text{this}(4)$  and  $\text{undef} = \text{this}(5)$  and  $T = \text{this}(6)$ 
  show ?case
    proof (intro allI impI)
      fix  $L'$  mark  $a\ b$ 
      assume  $a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } T$ 
      then consider
        (hd)  $a = []$  and  $L = L'$  and  $\text{mark} = C$  and  $b = \text{trail } S$ 
        | (tl)  $tl\ a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } S$ 
        using  $T\ \text{undef}$  by (cases  $a$ ) fastforce+
      then show  $b \models_{as} CNot\ (\text{mark} - \{\#L'\# \}) \wedge L' \in \# \text{mark}$ 
      using mark-confl confl LC by cases auto
    qed
  next
    case (decide  $L$ ) note  $\text{undef}[\text{simp}] = \text{this}(2)$  and  $T = \text{this}(4)$ 
    have  $\bigwedge a\ La\ \text{mark } b. a @ \text{Propagated } La\ \text{mark } \# b = \text{Decided } L \# \text{trail } S$ 
     $\implies tl\ a @ \text{Propagated } La\ \text{mark } \# b = \text{trail } S$  by (case-tac  $a$ ) auto
    then show ?case using mark-confl T unfolding decide.hyps(1) by fastforce
  next
    case (skip  $L\ C'\ M\ D\ T$ ) note  $tr = \text{this}(1)$  and  $T = \text{this}(5)$ 
    show ?case
      proof (intro allI impI)
        fix  $L'$  mark  $a\ b$ 
        assume  $a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } T$ 
        then have  $a @ \text{Propagated } L' \text{ mark } \# b = M$  using  $tr\ T$  by simp
        then have (Propagated  $L\ C' \# a$ ) @ Propagated  $L' \text{ mark } \# b = \text{Propagated } L\ C' \# M$  by auto
        moreover have  $\forall La\ \text{mark } a\ b. a @ \text{Propagated } La\ \text{mark } \# b = \text{Propagated } L\ C' \# M$ 
         $\longrightarrow b \models_{as} CNot\ (\text{mark} - \{\#La\# \}) \wedge La \in \# \text{mark}$ 
        using mark-confl unfolding skip.hyps(1) by simp
        ultimately show  $b \models_{as} CNot\ (\text{mark} - \{\#L'\# \}) \wedge L' \in \# \text{mark}$  by blast
      qed
    next
      case (conflict  $D$ )
      then show ?case using mark-confl by simp
    next
      case (resolve  $L\ C\ M\ D\ T$ ) note  $tr-S = \text{this}(1)$  and  $T = \text{this}(7)$ 
      show ?case unfolding resolve.hyps(1)
      proof (intro allI impI)
        fix  $L'$  mark  $a\ b$ 
        assume  $a @ \text{Propagated } L' \text{ mark } \# b = \text{trail } T$ 
        then have (Propagated  $L\ (C + \{\#L\# \}) \# a$ ) @ Propagated  $L' \text{ mark } \# b$ 
         $= \text{Propagated } L\ (C + \{\#L\# \}) \# M$ 
        using  $T\ tr-S$  by auto
        then show  $b \models_{as} CNot\ (\text{mark} - \{\#L'\# \}) \wedge L' \in \# \text{mark}$ 
        using mark-confl unfolding tr-S by (metis Cons-eq-appendI list.sel(3))

```

```

qed
next
case restart
then show ?case by auto
next
case forget
then show ?case using mark-confl by auto
next
case (backtrack L D K i M1 M2 T) note conf = this(1) and LD = this(2) and decomp = this(3)
and
  lev-K = this(7) and T = this(8)
have  $\forall l \in \text{set } M2. \neg \text{is-decided } l$ 
  using get-all-ann-decomposition-snd-not-decided decomp by blast
obtain M0 where M: trail S = M0 @ M2 @ Decided K # M1
  using decomp by auto
have [simp]: trail (reduce-trail-to M1 (add-learned-cls D
  (update-backtrack-lvl i (update-conflicting None S)))) = M1
  using decomp lev by (auto simp: cdclW-restart-M-level-inv-decomp)
let ?D' = remove1-mset L D
show ?case
proof (intro allI impI)
  fix La :: 'v literal and mark :: 'v clause and
    a b :: ('v, 'v clause) ann-lits
  assume a @ Propagated La mark # b = trail T
  then consider
    (hd-tr) a = [] and
      (Propagated La mark :: ('v, 'v clause) ann-lit) = Propagated L D and
      b = M1
  | (tl-tr) tl a @ Propagated La mark # b = M1
  using M T decomp lev by (cases a) (auto simp: cdclW-restart-M-level-inv-def)
then show b  $\models_{\text{as}}$  CNot (mark - {#La#})  $\wedge$  La  $\in \#$  mark
proof cases
  case hd-tr note A = this(1) and P = this(2) and b = this(3)
  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-l (trail S)
    unfolding atms-of-def
    by (meson image-subsetI true-annots-CNot-all-atms-defined)
  have vars-of-D: atms-of ?D'  $\subseteq$  atm-of ' lits-of-l M1
    using backtrack-atms-of-D-in-M1[of S D L i K M1 M2 T] T backtrack lev confl
    by (auto simp: cdclW-restart-M-level-inv-decomp)
  have no-dup (trail S) using lev by (auto simp: cdclW-restart-M-level-inv-decomp)
  then have  $\forall x \in \text{atms-of } ?D'. x \notin \text{atm-of ' lits-of-l } (M0 @ M2 @ \text{Decided } K \# [])$ 
    using vars-of-D distinct-atms-of-incl-not-in-other[of
      M0 @ M2 @ Decided K # [] M1] unfolding M by auto
  then have M1  $\models_{\text{as}}$  CNot ?D'
    using true-annots-remove-if-notin-vars[of M0 @ M2 @ Decided K # []
      M1 CNot ?D'] (trail S  $\models_{\text{as}}$  CNot D) unfolding M lits-of-def
    by (simp add: true-annot-CNot-diff)
  then show b  $\models_{\text{as}}$  CNot (mark - {#La#})  $\wedge$  La  $\in \#$  mark
    using P LD b by auto
next
case tl-tr
then obtain c' where c' @ Propagated La mark # b = trail S
  unfolding M by auto
then show b  $\models_{\text{as}}$  CNot (mark - {#La#})  $\wedge$  La  $\in \#$  mark
  using mark-confl by auto

```

qed
qed
qed

lemma *cdcl_W-restart-conflicting-is-false:*

assumes

cdcl_W-restart S S' **and**

M-lev: *cdcl_W-restart-M-level-inv* S **and**

confl-inv: $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot \ T$ **and**

decided-confl: $\forall L \text{ mark } a \ b. a \ @ \ \text{Propagated } L \ \text{mark } \# \ b = (\text{trail } S)$

$\longrightarrow (b \models_{as} CNot \ (\text{mark} - \{\#L\# \}) \wedge L \in \# \ \text{mark})$ **and**

dist: *distinct-cdcl_W-restart-state* S

shows $\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{trail } S' \models_{as} CNot \ T$

using *assms*(1,2)

proof (*induct rule*: *cdcl_W-restart-all-induct*)

case (*skip* L C' M D T) **note** $tr-S = \text{this}(1)$ **and** $confl = \text{this}(2)$ **and** $L-D = \text{this}(3)$ **and** $T = \text{this}(5)$

have D : *Propagated* L $C' \# M \models_{as} CNot \ D$ **using** *assms skip* **by** *auto*

moreover

have $L \notin \# \ D$

proof (*rule ccontr*)

assume $\neg \ ?thesis$

then have $- L \in \text{lits-of-l } M$

using *in-CNot-implies-uminus*(2)[*of* L D *Propagated* L $C' \# M$]

$\langle \text{Propagated } L \ C' \# M \models_{as} CNot \ D \rangle$ **by** *simp*

then show *False*

by (*metis* (*no-types*, *hide-lams*) *M-lev cdcl_W-restart-M-level-inv-decomp*(1) *consistent-interp-def* *image-insert insert-iff list.set*(2) *lits-of-def ann-lit.sel*(2) $tr-S$)

qed

ultimately show *?case*

using $tr-S$ *confl* $L-D$ T **unfolding** *cdcl_W-restart-M-level-inv-def*

by (*auto intro*: *true-annots-CNot-lit-of-notin-skip*)

next

case (*resolve* L C M D T) **note** $tr = \text{this}(1)$ **and** $LC = \text{this}(2)$ **and** $confl = \text{this}(4)$ **and** $LD = \text{this}(5)$

and $T = \text{this}(7)$

let $?C = \text{remove1-mset } L \ C$

let $?D = \text{remove1-mset } (-L) \ D$

show *?case*

proof (*intro allI impI*)

fix T'

have $tl \ (\text{trail } S) \models_{as} CNot \ ?C$ **using** tr *decided-confl* **by** *fastforce*

moreover

have *distinct-mset* $(?D + \{\#- L\# \})$ **using** *confl dist LD*

unfolding *distinct-cdcl_W-restart-state-def* **by** *auto*

then have $-L \notin \# \ ?D$ **unfolding** *distinct-mset-def*

by (*meson* $\langle \text{distinct-mset } (?D + \{\#- L\# \}) \rangle$ *distinct-mset-single-add*)

have $M \models_{as} CNot \ ?D$

proof $-$

have *Propagated* L $(?C + \{\#L\# \}) \# M \models_{as} CNot \ ?D \cup CNot \ \{\#- L\# \}$

using *confl tr confl-inv LC* **by** (*metis* *CNot-plus LD insert-DiffM2*)

then show *?thesis*

using *M-lev* $\langle - L \notin \# \ ?D \rangle$ tr *true-annots-lit-of-notin-skip*

unfolding *cdcl_W-restart-M-level-inv-def* **by** *force*

qed

moreover assume *conflicting* $T = \text{Some } T'$

ultimately
 show $\text{trail } T \models_{as} \text{CNot } T'$
 using $\text{tr } T$ by *auto*
 qed
 qed (*auto simp: M-lev cdcl_W-restart-M-level-inv-decomp*)

lemma *cdcl_W-restart-conflicting-decomp*:
 assumes *cdcl_W-restart-conflicting* S
 shows $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$
 and $\forall L \text{ mark } a \ b. a \ @ \ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S)$
 $\longrightarrow (b \models_{as} \text{CNot } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$
 using *assms unfolding cdcl_W-restart-conflicting-def* by *blast+*

lemma *cdcl_W-restart-conflicting-decomp2*:
 assumes *cdcl_W-restart-conflicting* S and *conflicting* $S = \text{Some } T$
 shows $\text{trail } S \models_{as} \text{CNot } T$
 using *assms unfolding cdcl_W-restart-conflicting-def* by *blast+*

lemma *cdcl_W-restart-conflicting-S0-cdcl_W-restart[simp]*:
cdcl_W-restart-conflicting (*init-state* N)
unfolding cdcl_W-restart-conflicting-def by *auto*

Putting all the invariants together

lemma *cdcl_W-restart-all-inv*:
 assumes
 cdcl_W-restart: cdcl_W-restart $S \ S'$ and
 1: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) and
 2: *cdcl_W-restart-learned-clause* S and
 4: *cdcl_W-restart-M-level-inv* S and
 5: *no-strange-atm* S and
 7: *distinct-cdcl_W-restart-state* S and
 8: *cdcl_W-restart-conflicting* S
 shows
 all-decomposition-implies-m (*init-clss* S') (*get-all-ann-decomposition* (*trail* S')) and
 cdcl_W-restart-learned-clause S' and
 cdcl_W-restart-M-level-inv S' and
 no-strange-atm S' and
 distinct-cdcl_W-restart-state S' and
 cdcl_W-restart-conflicting S'
 proof –
 show $S1$: *all-decomposition-implies-m* (*init-clss* S') (*get-all-ann-decomposition* (*trail* S'))
 using *cdcl_W-restart-propagate-is-conclusion*[*OF cdcl_W-restart 4 1 2 - 5*] 8 *unfolding cdcl_W-restart-conflicting-def*
 by *blast*
 show $S2$: *cdcl_W-restart-learned-clause* S' using *cdcl_W-restart-learned-clss*[*OF cdcl_W-restart 2 4*] .
 show $S4$: *cdcl_W-restart-M-level-inv* S' using *cdcl_W-restart-consistent-inv*[*OF cdcl_W-restart 4*] .
 show $S5$: *no-strange-atm* S' using *cdcl_W-restart-no-strange-atm-inv*[*OF cdcl_W-restart 5 4*] .
 show $S7$: *distinct-cdcl_W-restart-state* S' using *distinct-cdcl_W-restart-state-inv*[*OF cdcl_W-restart 4 7*]
 .
 show $S8$: *cdcl_W-restart-conflicting* S'
 using *cdcl_W-restart-conflicting-is-false*[*OF cdcl_W-restart 4 - - 7*] 8 *cdcl_W-restart-propagate-is-false*[*OF cdcl_W-restart 4 2 1 -*
 5]
 unfolding cdcl_W-restart-conflicting-def by *fast*
 qed

lemma *rtrancpl-cdcl_W-restart-all-inv*:

assumes

cdcl_W-restart: *rtrancpl cdcl_W-restart S S'* **and**

1: *all-decomposition-implies-m (init-clss S) (get-all-ann-decomposition (trail S))* **and**

2: *cdcl_W-restart-learned-clause S* **and**

4: *cdcl_W-restart-M-level-inv S* **and**

5: *no-strange-atm S* **and**

7: *distinct-cdcl_W-restart-state S* **and**

8: *cdcl_W-restart-conflicting S*

shows

all-decomposition-implies-m (init-clss S') (get-all-ann-decomposition (trail S')) **and**

cdcl_W-restart-learned-clause S' **and**

cdcl_W-restart-M-level-inv S' **and**

no-strange-atm S' **and**

distinct-cdcl_W-restart-state S' **and**

cdcl_W-restart-conflicting S'

using *assms*

proof (*induct rule: rtrancpl-induct*)

case *base*

case 1 **then show** ?*case* **by** *blast*

case 2 **then show** ?*case* **by** *blast*

case 3 **then show** ?*case* **by** *blast*

case 4 **then show** ?*case* **by** *blast*

case 5 **then show** ?*case* **by** *blast*

case 6 **then show** ?*case* **by** *blast*

next

case (*step S' S''*) **note** *H = this*

case 1 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-restart-all-inv[OF H(2)]*

H **by** *presburger*

case 2 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-restart-all-inv[OF H(2)]*

H **by** *presburger*

case 3 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-restart-all-inv[OF H(2)]*

H **by** *presburger*

case 4 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-restart-all-inv[OF H(2)]*

H **by** *presburger*

case 5 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-restart-all-inv[OF H(2)]*

H **by** *presburger*

case 6 **with** *H(3-7)[OF this(1-6)]* **show** ?*case* **using** *cdcl_W-restart-all-inv[OF H(2)]*

H **by** *presburger*

qed

lemma *all-invariant-S0-cdcl_W-restart*:

assumes *distinct-mset-mset N*

shows

all-decomposition-implies-m (init-clss (init-state N))

(*get-all-ann-decomposition (trail (init-state N))*) **and**

cdcl_W-restart-learned-clause (init-state N) **and**

$\forall T. \text{conflicting } (init-state N) = \text{Some } T \longrightarrow (\text{trail } (init-state N)) \models_{as} CNot T$ **and**

no-strange-atm (init-state N) **and**

consistent-interp (lits-of-l (trail (init-state N))) **and**

$\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = \text{trail } (init-state N) \longrightarrow$

$(b \models_{as} CNot (\text{mark} - \{\#L\}) \wedge L \in \# \text{mark})$ **and**

distinct-cdcl_W-restart-state (init-state N)

using *assms* **by** *auto*

lemma *cdcl_W-only-propagated-vars-unsat*:

assumes

decided: $\forall x \in \text{set } M. \neg \text{is-decided } x$ **and**

DN: $D \in \# \text{ clauses } S$ **and**

D: $M \models_{as} CNot\ D$ **and**

inv: *all-decomposition-implies-m* *N* (*get-all-ann-decomposition* *M*) **and**

state: *state* *S* = (*M*, *N*, *U*, *k*, *C*) **and**

learned-cl: *cdcl_W-restart-learned-clause* *S* **and**

atm-incl: *no-strange-atm* *S*

shows *unsatisfiable* (*set-mset* *N*)

proof (*rule ccontr*)

assume $\neg \text{unsatisfiable} (\text{set-mset } N)$

then obtain *I* **where**

I: $I \models_s \text{set-mset } N$ **and**

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

tot: *total-over-m* *I* (*set-mset* *N*)

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

have *atms-of-mm* $N \cup \text{atms-of-mm } U = \text{atms-of-mm } N$

using *atm-incl* *state* **unfolding** *total-over-m-def* *no-strange-atm-def*

by (*auto simp add: clauses-def*)

then have *total-over-m* *I* (*set-mset* *N*) **using** *tot* **unfolding** *total-over-m-def* **by** *auto*

moreover then have *total-over-m* *I* (*set-mset* (*learned-clss* *S*))

using *atm-incl* *state* **unfolding** *no-strange-atm-def* *total-over-m-def* *total-over-set-def*

by *auto*

moreover have $N \models_{psm} U$ **using** *learned-cl* *state* **unfolding** *cdcl_W-restart-learned-clause-def* **by** *auto*

ultimately have *I-D*: $I \models D$

using *I DN cons state* **unfolding** *true-clss-clss-def* *true-clss-def* *Ball-def*

by (*auto simp add: clauses-def*)

have *l0*: $\{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M\} = \{\}$ **using** *decided* **by** *auto*

have *atms-of-ms* (*set-mset* $N \cup \text{unmark-l } M$) = *atms-of-mm* *N*

using *atm-incl* *state* **unfolding** *no-strange-atm-def* **by** *auto*

then have *total-over-m* *I* (*set-mset* $N \cup \text{unmark-l } M$)

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

then have $I \models_s \text{unmark-l } M$

using *all-decomposition-implies-propagated-lits-are-implied*[*OF inv*] *cons I*

unfolding *true-clss-clss-def* *l0* **by** *auto*

then have *IM*: $I \models_s \text{unmark-l } M$ **by** *auto*

{

fix *K*

assume $K \in \# D$

then have $-K \in \text{lits-of-l } M$

using *D* **unfolding** *true-annots-def* *Ball-def* *CNot-def* *true-annot-def* *true-clss-def* *true-lit-def*

Bex-def **by** *force*

then have $-K \in I$ **using** *IM* *true-clss-singleton-lit-of-implies-incl* *lits-of-def* **by** *fastforce* }

then have $\neg I \models D$ **using** *cons* **unfolding** *true-clss-def* *true-lit-def* *consistent-interp-def* **by** *auto*

then show *False* **using** *I-D* **by** *blast*

qed

Item 5 page 81 of Weidenbach's book

We have actually a much stronger theorem, namely *all-decomposition-implies-propagated-lits-are-implied*, that show that the only choices we made are decided in the formula

lemma

assumes *all-decomposition-implies-m* *N* (*get-all-ann-decomposition* *M*)

and $\forall m \in \text{set } M. \neg \text{is-decided } m$
shows $\text{set-mset } N \models_{ps} \text{unmark-l } M$
proof –
have $T: \{\text{unmark } L \mid L. \text{is-decided } L \wedge L \in \text{set } M\} = \{\}$ **using** $\text{assms}(2)$ **by** auto
then show $?thesis$
using $\text{all-decomposition-implies-propagated-lits-are-implied}[OF \text{ assms}(1)]$ **unfolding** T **by** simp
qed

Item 7 page 81 of Weidenbach’s book (part 1)

lemma *conflict-with-false-implies-unsat:*

assumes
 $\text{cdcl}_W\text{-restart}: \text{cdcl}_W\text{-restart } S \ S'$ **and**
 $\text{lev}: \text{cdcl}_W\text{-restart-}M\text{-level-inv } S$ **and**
 $[\text{simp}]: \text{conflicting } S' = \text{Some } \{\#\}$ **and**
 $\text{learned}: \text{cdcl}_W\text{-restart-learned-clause } S$
shows $\text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$
using assms

proof –
have $\text{cdcl}_W\text{-restart-learned-clause } S'$ **using** $\text{cdcl}_W\text{-restart-learned-clss } \text{cdcl}_W\text{-restart learned lev}$ **by** auto
then have $\text{init-clss } S' \models_{pm} \{\#\}$ **using** $\text{assms}(3)$ **unfolding** $\text{cdcl}_W\text{-restart-learned-clause-def}$ **by** auto
then have $\text{init-clss } S \models_{pm} \{\#\}$
using $\text{cdcl}_W\text{-restart-init-clss}[OF \text{ assms}(1) \text{ lev}]$ **by** auto
then show $?thesis$ **unfolding** $\text{satisfiable-def true-clss-clss-def}$ **by** auto
qed

Item 7 page 81 of Weidenbach’s book (part 2)

lemma *conflict-with-false-implies-terminated:*

assumes $\text{cdcl}_W\text{-restart } S \ S'$
and $\text{conflicting } S = \text{Some } \{\#\}$
shows False
using assms **by** $(\text{induct rule: } \text{cdcl}_W\text{-restart-all-induct}) \text{ auto}$

No tautology is learned

This is a simple consequence of all we have shown previously. It is not strictly necessary, but helps finding a better bound on the number of learned clauses.

lemma *learned-clss-are-not-tautologies:*

assumes
 $\text{cdcl}_W\text{-restart } S \ S'$ **and**
 $\text{lev}: \text{cdcl}_W\text{-restart-}M\text{-level-inv } S$ **and**
 $\text{conflicting}: \text{cdcl}_W\text{-restart-conflicting } S$ **and**
 $\text{no-tauto}: \forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$
shows $\forall s \in \# \text{ learned-clss } S'. \neg \text{tautology } s$
using assms

proof $(\text{induct rule: } \text{cdcl}_W\text{-restart-all-induct})$
case $(\text{backtrack } L \ D \ K \ i \ M1 \ M2 \ T)$ **note** $\text{confl} = \text{this}(1)$
have $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$ **using** lev **by** $(\text{auto simp: } \text{cdcl}_W\text{-restart-}M\text{-level-inv-decomp})$
moreover
have $\text{trail } S \models_{as} \text{CNot } D$
using conflicting confl **unfolding** $\text{cdcl}_W\text{-restart-conflicting-def}$ **by** auto
then have $\text{lits-of-l } (\text{trail } S) \models_s \text{CNot } D$
using $\text{true-annots-true-clss}$ **by** blast
ultimately have $\neg \text{tautology } D$ **using** $\text{consistent-CNot-not-tautology}$ **by** blast
then show $?case$ **using** $\text{backtrack no-tauto lev}$

by (auto simp: cdcl_W-restart-M-level-inv-decomp split: if-split-asm)
 next
 case restart
 then show ?case using state-eq-learned-clss no-tauto
 by (auto intro: atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI)
 qed (auto dest!: in-diffD)

definition final-cdcl_W-restart-state ($S :: 'st$)
 \longleftrightarrow (trail $S \models_{asm} init-clss\ S$
 $\vee ((\forall L \in set\ (trail\ S). \neg is-decided\ L) \wedge$
 $(\exists C \in \# init-clss\ S. trail\ S \models_{as}\ CNot\ C)))$

definition termination-cdcl_W-restart-state ($S :: 'st$)
 \longleftrightarrow (trail $S \models_{asm} init-clss\ S$
 $\vee ((\forall L \in atms-of-mm\ (init-clss\ S). L \in atm-of\ ' lits-of-l\ (trail\ S))$
 $\wedge (\exists C \in \# init-clss\ S. trail\ S \models_{as}\ CNot\ C)))$

6.1.4 CDCL Strong Completeness

lemma cdcl_W-restart-can-do-step:

assumes
 consistent-interp (set M) **and**
 distinct M **and**
 atm-of ' (set M) \subseteq atms-of-mm N
shows $\exists S. rtrancplp\ cdcl_W-restart\ (init-state\ N)\ S$
 $\wedge state\ S = (map\ (\lambda L. Decided\ L)\ M, N, \{\#\}, length\ M, None)$
using assms
proof (induct M)
 case Nil
 then show ?case apply – by (rule exI[of - init-state N]) auto
 next
 case (Cons $L\ M$) **note** $IH = this(1)$
have consistent-interp (set M) **and** distinct M **and** atm-of ' set $M \subseteq$ atms-of-mm N
using Cons.prem(1–3) **unfolding** consistent-interp-def **by** auto
then obtain S **where**
 $st: cdcl_W-restart^{**}\ (init-state\ N)\ S$ **and**
 $S: state\ S = (map\ (\lambda L. Decided\ L)\ M, N, \{\#\}, length\ M, None)$
using IH **by** blast
 let $?S_0 = incr-lvl\ (cons-trail\ (Decided\ L)\ S)$
have undefined-lit (map (λ L . Decided L) M) L
using Cons.prem(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-mm N **using** Cons.prem(3) **by** auto
moreover have undef: undefined-lit (trail S) L
using S <distinct $(L\ \# M)$ > calculation(1) **by** (auto simp: defined-lit-map)
ultimately have cdcl_W-restart $S\ ?S_0$
using cdcl_W-restart.other[OF cdcl_W-o.decide[OF decide-rule[of $S\ L\ ?S_0$]]] S
by (auto simp: state-eq-def simp del: state-simp)
then have cdcl_W-restart^{**} (init-state N) $?S_0$
using st **by** auto
then show ?case
using S undef **by** (auto intro!: exI[of - ? S_0] del: simp del:)
 qed

theorem 2.9.11 page 84 of Weidenbach's book

lemma *cdcl_W-restart-strong-completeness*:
assumes
MN: *set M* \models_{sm} *N* **and**
cons: *consistent-interp* (*set M*) **and**
dist: *distinct M* **and**
atm: *atm-of* ' (*set M*) \subseteq *atms-of-mm N*
obtains *S* **where**
state S = (*map* ($\lambda L.$ *Decided L*) *M*, *N*, $\{\#\}$, *length M*, *None*) **and**
rtranclp cdcl_W-restart (*init-state N*) *S* **and**
final-cdcl_W-restart-state S
proof –
obtain *S* **where**
st: *rtranclp cdcl_W-restart* (*init-state N*) *S* **and**
S: *state S* = (*map* ($\lambda L.$ *Decided L*) *M*, *N*, $\{\#\}$, *length M*, *None*)
using *cdcl_W-restart-can-do-step*[*OF cons dist atm*] **by** *auto*
have *lits-of-l* (*map* ($\lambda L.$ *Decided L*) *M*) = *set M*
by (*induct M*, *auto*)
then have *map* ($\lambda L.$ *Decided L*) *M* \models_{asm} *N* **using** *MN true-annots-true-cls* **by** *metis*
then have *final-cdcl_W-restart-state S*
using *S unfolding final-cdcl_W-restart-state-def* **by** *auto*
then show *?thesis* **using** *that st S* **by** *blast*
qed

6.1.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have to add a strategy.

Definition

lemma *tranclp-conflict*:
tranclp conflict S S' \implies conflict S S'
apply (*induct rule: tranclp.induct*)
apply *simp*
by (*metis conflictE conflicting-update-conflicting option.distinct(1) state-eq-conflicting*)

lemma *tranclp-conflict-iff*[*iff*]:
full1 conflict S S' \longleftrightarrow conflict S S'

proof –
have *tranclp conflict S S' \implies conflict S S'* **by** (*meson tranclp-conflict rtranclpD*)
then show *?thesis* **unfolding** *full1-def*
by (*metis conflict.simps conflicting-update-conflicting option.distinct(1) 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-restart*:
*cdcl_W-cp** S T \implies cdcl_W-restart** S T*
by (*induction rule: rtranclp-induct*) (*auto simp: cdcl_W-cp.simps dest: cdcl_W-restart.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-cdcl_W-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 cdcl_W-cp-state-eq-compatible by blast
next
  case (step  $U\ V$ )
  obtain  $ss :: 'st$  where
     $cdcl_W\text{-}cp\ S\ ss$  and  $cdcl_W\text{-}cp^{**}\ ss\ U$ 
  by (metis (no-types) step(1) tranclpD)
  then show ?case
    by (meson cdcl_W-cp-state-eq-compatible rtranclp.rtrancl-into-rtrancl rtranclp-into-tranclp2
      state-eq-ref step(2) step(4) step(5))
qed

lemma option-full-cdcl_W-cp:
   $conflicting\ S \neq None \implies full\ cdcl_W\text{-}cp\ S\ S$ 
  unfolding full-def rtranclp-unfold tranclp-unfold
  by (auto simp add: cdcl_W-cp.simps elim: conflictE propagateE)

lemma skip-unique:
   $skip\ S\ T \implies skip\ S\ T' \implies T \sim T'$ 
  by (fastforce simp: state-eq-def simp del: state-simp elim: skipE)

lemma resolve-unique:
   $resolve\ S\ T \implies resolve\ S\ T' \implies T \sim T'$ 
  by (fastforce simp: state-eq-def simp del: state-simp elim: resolveE)

lemma cdcl_W-cp-no-more-clauses:
  assumes  $cdcl_W\text{-}cp\ S\ S'$ 
  shows  $clauses\ S = clauses\ S'$ 
  using assms by (induct rule: cdcl_W-cp.induct) (auto elim!: conflictE propagateE)

lemma tranclp-cdcl_W-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: cdcl_W-cp-no-more-clauses)

lemma rtranclp-cdcl_W-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: cdcl_W-cp-no-more-clauses) +

```

lemma *no-conflict-after-conflict*:
 $\text{conflict } S \ T \implies \neg \text{conflict } T \ U$
by (metis *conflictE conflicting-update-conflicting option.distinct(1) state-simp(5)*)

lemma *no-propagate-after-conflict*:
 $\text{conflict } S \ T \implies \neg \text{propagate } T \ U$
by (metis *conflictE conflicting-update-conflicting option.distinct(1) propagate.cases state-eq-conflicting*)

lemma *tranclp-cdcl_W-cp-propagate-with-conflict-or-not*:
assumes $\text{cdcl}_W\text{-cp}^{++} \ S \ U$
shows $(\text{propagate}^{++} \ S \ U \wedge \text{conflicting } U = \text{None})$
 $\vee (\exists T \ D. \text{propagate}^{**} \ S \ T \wedge \text{conflict } T \ U \wedge \text{conflicting } U = \text{Some } D)$
proof –
have $\text{propagate}^{++} \ S \ U \vee (\exists T. \text{propagate}^{**} \ S \ T \wedge \text{conflict } T \ U)$
using *assms* **by** *induction*
(force simp: cdcl_W-cp.simps tranclp-into-rtranclp dest: no-conflict-after-conflict no-propagate-after-conflict)+
moreover
have $\text{propagate}^{++} \ S \ U \implies \text{conflicting } U = \text{None}$
unfolding *tranclp-unfold-end* **by** (auto *elim!*: *propagateE*)
moreover
have $\bigwedge T. \text{conflict } T \ U \implies \exists D. \text{conflicting } U = \text{Some } D$
by (auto *elim!*: *conflictE simp: state-eq-def simp del: state-simp*)
ultimately show *?thesis* **by** *meson*
qed

lemma *cdcl_W-cp-conflicting-not-empty[simp]*: $\text{conflicting } S = \text{Some } D \implies \neg \text{cdcl}_W\text{-cp } S \ S'$
proof
assume $\text{cdcl}_W\text{-cp } S \ S'$ **and** $\text{conflicting } S = \text{Some } D$
then show *False* **by** (*induct rule: cdcl_W-cp.induct*)
(auto elim: conflictE propagateE simp: state-eq-def simp del: state-simp)
qed

lemma *no-step-cdcl_W-cp-no-conflict-no-propagate*:
assumes *no-step cdcl_W-cp S*
shows *no-step conflict S and no-step propagate S*
using *assms conflict'* **apply** *blast*
by (*meson assms conflict' propagate'*)

CDCL with the reasonable strategy: we fully propagate the conflict and propagate, then we apply any other possible rule $\text{cdcl}_W\text{-o } S \ S'$ and re-apply conflict and propagate $\text{cdcl}_W\text{-cp}^\downarrow \ S' \ S''$

inductive *cdcl_W-stgy* :: *'st* \Rightarrow *'st* \Rightarrow *bool* **for** *S* :: *'st* **where**
conflict': $\text{full1 } \text{cdcl}_W\text{-cp } S \ S' \implies \text{cdcl}_W\text{-stgy } S \ S' \mid$
other': $\text{cdcl}_W\text{-o } S \ S' \implies \text{no-step } \text{cdcl}_W\text{-cp } S \implies \text{full } \text{cdcl}_W\text{-cp } S' \ S'' \implies \text{cdcl}_W\text{-stgy } S \ S''$

lemma *cdcl_W-cp-cdcl_W*: $\text{cdcl}_W\text{-cp } S \ T \implies \text{cdcl}_W \ S \ T$
by (*induction rule: cdcl_W-cp.induct*) (auto *intro: cdcl_W.intros*)

lemma *cdcl_W-stgy* $S \ T \implies \text{cdcl}_W^{++} \ S \ T$
proof (*induction rule: cdcl_W-stgy.induct*)
case *conflict'*
then show *?case*

```

  unfolding full1-def
  using rtrancpl-mono[of cdclW-cp cdclW] rtrancpl-mono[of cdclW-cp cdclW]
  apply (auto intro: cdclW.intros simp: le-fun-def
    dest!: cdclW-cp-cdclW W-other)
  done
next
case (other' T U) note o = this(1) and n-s = this(2) and full = this(3)
have cdclW S T
  using o by (auto dest!: W-other)
moreover have cdclW** T U
  using full rtrancpl-mono[of cdclW-cp cdclW] cdclW-cp-cdclW unfolding full-def
  by (auto intro: cdclW.intros simp: le-fun-def
    dest!: cdclW-cp-cdclW W-other)
ultimately show ?case by auto
qed

```

Invariants

These are the same invariants as before, but lifted

```

lemma cdclW-cp-learned-clause-inv:
  assumes cdclW-cp S S'
  shows learned-clss S = learned-clss S'
  using assms by (induct rule: cdclW-cp.induct) (fastforce elim: conflictE propagateE)+

lemma rtrancpl-cdclW-cp-learned-clause-inv:
  assumes cdclW-cp** S S'
  shows learned-clss S = learned-clss S'
  using assms by (induct rule: rtrancpl-induct) (fastforce dest: cdclW-cp-learned-clause-inv)+

lemma trancpl-cdclW-cp-learned-clause-inv:
  assumes cdclW-cp++ S S'
  shows learned-clss S = learned-clss S'
  using assms by (simp add: rtrancpl-cdclW-cp-learned-clause-inv trancpl-into-rtrancpl)

lemma cdclW-cp-backtrack-lvl:
  assumes cdclW-cp S S'
  shows backtrack-lvl S = backtrack-lvl S'
  using assms by (induct rule: cdclW-cp.induct) (fastforce elim: conflictE propagateE)+

lemma rtrancpl-cdclW-cp-backtrack-lvl:
  assumes cdclW-cp** S S'
  shows backtrack-lvl S = backtrack-lvl S'
  using assms by (induct rule: rtrancpl-induct) (fastforce dest: cdclW-cp-backtrack-lvl)+

lemma cdclW-cp-consistent-inv:
  assumes cdclW-cp S S' and cdclW-restart-M-level-inv S
  shows cdclW-restart-M-level-inv S'
  using assms
proof (induct rule: cdclW-cp.induct)
  case (conflict')
  then show ?case using cdclW-restart-consistent-inv cdclW-restart.conflict by blast
next
  case (propagate' S S')
  have cdclW-restart S S'
  using propagate'.hyps(1) propagate by blast

```

then show *cdcl_W-restart-M-level-inv S'*
using *propagate'.prems(1) cdcl_W-restart-consistent-inv propagate* **by** *blast*
qed

lemma *full1-cdcl_W-cp-consistent-inv*:
assumes *full1 cdcl_W-cp S S' and cdcl_W-restart-M-level-inv S*
shows *cdcl_W-restart-M-level-inv S'*
using *assms unfolding full1-def*
by (*metis rtrancpl-cdcl_W-cp-rtrancpl-cdcl_W-restart rtrancpl-unfold trancpl-cdcl_W-restart-consistent-inv*)

lemma *rtrancpl-cdcl_W-cp-consistent-inv*:
assumes *rtrancpl cdcl_W-cp S S' and cdcl_W-restart-M-level-inv S*
shows *cdcl_W-restart-M-level-inv S'*
using *assms unfolding full1-def*
by (*induction rule: rtrancpl-induct*) (*blast intro: cdcl_W-cp-consistent-inv*)**+**

lemma *cdcl_W-stgy-consistent-inv*:
assumes *cdcl_W-stgy S S' and cdcl_W-restart-M-level-inv S*
shows *cdcl_W-restart-M-level-inv S'*
using *assms apply (induct rule: cdcl_W-stgy.induct)*
unfolding *full-unfold* **by** (*blast intro: cdcl_W-restart-consistent-inv full1-cdcl_W-cp-consistent-inv cdcl_W-restart.other*)**+**

lemma *rtrancpl-cdcl_W-stgy-consistent-inv*:
assumes *cdcl_W-stgy** S S' and cdcl_W-restart-M-level-inv S*
shows *cdcl_W-restart-M-level-inv S'*
using *assms by induction (auto dest!: cdcl_W-stgy-consistent-inv)*

lemma *cdcl_W-cp-no-more-init-clss*:
assumes *cdcl_W-cp S S'*
shows *init-clss S = init-clss S'*
using *assms by (induct rule: cdcl_W-cp.induct) (auto elim: conflictE propagateE)*

lemma *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-restart-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-restart-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** $\text{trail } S' = M @ \text{trail } S$ **and** $(\forall l \in \text{set } M. \neg \text{is-decided } l)$
using *assms* **by induction** (*fastforce elim: conflictE propagateE*) $+$

lemma *rtrancp-cdcl_W-cp-dropWhile-trail'*:
assumes *cdcl_W-cp*** $S S'$
obtains $M :: ('v, 'v \text{ clause}) \text{ ann-lits}$ **where**
 $\text{trail } S' = M @ \text{trail } S$ **and** $\forall l \in \text{set } M. \neg \text{is-decided } l$
using *assms* **by induction** (*fastforce dest!: cdcl_W-cp-dropWhile-trail'*) $+$

lemma *cdcl_W-cp-dropWhile-trail*:
assumes *cdcl_W-cp* $S S'$
shows $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-decided } l)$
using *assms* **by induction** (*fastforce elim: conflictE propagateE*) $+$

lemma *rtrancp-cdcl_W-cp-dropWhile-trail*:
assumes *cdcl_W-cp*** $S S'$
shows $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-decided } l)$
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
 $\text{no-strange-atm } S$ **and**
 $\text{no-d: no-dup } (\text{trail } S)$ **and**
 $\text{finite } (\text{atms-of-mm } (\text{init-clss } S))$
shows $\text{length } (\text{trail } S) \leq \text{card } (\text{atms-of-mm } (\text{init-clss } S))$

proof –
obtain $M N U k D$ **where** $S: \text{state } S = (M, N, U, k, D)$ **by** (*cases state S, auto*)
have $\text{finite } (\text{atm-of } ' \text{ lits-of-l } (\text{trail } S))$
using *assms*(1,3) **unfolding** S **by** (*auto simp add: finite-subset*)
have $\text{length } (\text{trail } S) = \text{card } (\text{atm-of } ' \text{ lits-of-l } (\text{trail } S))$
using *no-dup-length-eq-card-atm-of-lits-of-l no-d* **by** *blast*
then show *?thesis* **using** *assms*(1) **unfolding** *no-strange-atm-def*
by (*auto simp add: assms*(3) *card-mono*)
qed

lemma *cdcl_W-cp-decreasing-measure*:
assumes
 $\text{cdcl}_W\text{-restart: cdcl}_W\text{-cp } S T$ **and**
 $M\text{-lev: cdcl}_W\text{-restart-M-level-inv } S$ **and**
 $\text{alien: no-strange-atm } S$
shows $(\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$
 $+ (\text{if conflicting } S = \text{None then } 1 \text{ else } 0)) S$
 $> (\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$
 $+ (\text{if conflicting } S = \text{None then } 1 \text{ else } 0)) T$
using *assms*
proof –
have $\text{length } (\text{trail } T) \leq \text{card } (\text{atms-of-mm } (\text{init-clss } T))$
apply (*rule length-model-le-vars*)
using *cdcl_W-restart-no-strange-atm-inv alien M-lev* **apply** (*meson cdcl_W-restart cdcl_W-restart.simps*
cdcl_W-cp.cases)
using *M-lev cdcl_W-restart cdcl_W-cp-consistent-inv cdcl_W-restart-M-level-inv-def* **apply** *blast*
using *cdcl_W-restart* **by** (*auto simp: cdcl_W-cp.simps*)
with *assms*
show *?thesis* **by induction** (*auto elim!: conflictE propagateE*
simp del: state-simp simp: state-eq-def) $+$

qed

lemma *cdcl_W-cp-wf*: *wf* $\{(b, a). (cdcl_W\text{-restart-}M\text{-level-inv } a \wedge no\text{-strange-atm } a) \wedge cdcl_W\text{-cp } a \ b)\}$
apply (*rule* *wf-wf-if-measure*'[*of less-than - -*
 $(\lambda S. card (atms\text{-of-mm } (init\text{-cls } S)) - length (trail S)$
 $+ (if\ conflicting\ S = None\ then\ 1\ else\ 0))]$)
apply *simp*
using *cdcl_W-cp-decreasing-measure* **unfolding** *less-than-iff* **by** *blast*

lemma *rtrancp-cdcl_W-restart-all-struct-inv-cdcl_W-cp-iff-rtrancp-cdcl_W-cp*:

assumes

lev: *cdcl_W-restart- M -level-inv* *S* **and**

alien: *no-strange-atm* *S*

shows $(\lambda a\ b. (cdcl_W\text{-restart-}M\text{-level-inv } a \wedge no\text{-strange-atm } a) \wedge cdcl_W\text{-cp } a \ b)^{**} S\ T$

$\longleftrightarrow cdcl_W\text{-cp}^{**} S\ T$

(**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 *cdcl_W-restart*^{**} *S* *T*

by (*metis* *rtrancp-unfold* *cdcl_W-cp-conflicting-not-empty* *cp* *st*

rtrancp-propagate-is-rtrancp-cdcl_W-restart *trancp-cdcl_W-cp-propagate-with-conflict-or-not*)

then have

cdcl_W-restart- M -level-inv *T* **and**

no-strange-atm *T*

using $\langle cdcl_W\text{-restart}^{**} S\ T \rangle$ **apply** (*simp* *add*: *assms*(1) *rtrancp-cdcl_W-restart-consistent-inv*)

using $\langle cdcl_W\text{-restart}^{**} S\ T \rangle$ *alien* *rtrancp-cdcl_W-restart-no-strange-atm-inv* *lev* **by** *blast*

then have $(\lambda a\ b. (cdcl_W\text{-restart-}M\text{-level-inv } a \wedge no\text{-strange-atm } a) \wedge cdcl_W\text{-cp } a \ b)^{**} T\ U$

using *cp* **by** *auto*

then show *?case* **using** *IH* **by** *auto*

qed

qed

lemma *cdcl_W-cp-normalized-element*:

assumes

lev: *cdcl_W-restart- M -level-inv* *S* **and**

no-strange-atm *S*

obtains *T* **where** *full* *cdcl_W-cp* *S* *T*

proof –

let *?inv* = $\lambda a. (cdcl_W\text{-restart-}M\text{-level-inv } a \wedge no\text{-strange-atm } a)$

obtain *T* **where** *T*: *full* $(\lambda a\ b. ?inv\ a \wedge cdcl_W\text{-cp } a \ b)$ *S* *T*

using *cdcl_W-cp-wf* *wf-exists-normal-form*[*of* $\lambda a\ b. ?inv\ a \wedge cdcl_W\text{-cp } a \ b]$

unfolding *full-def* **by** *blast*

then have *cdcl_W-cp*^{**} *S* *T*

using *rtrancp-cdcl_W-restart-all-struct-inv-cdcl_W-cp-iff-rtrancp-cdcl_W-cp* *assms* **unfolding** *full-def*
by *blast*

moreover
then have $cdcl_W\text{-restart}^{**} S T$
using $rtranclp\text{-}cdcl_W\text{-}cp\text{-}rtranclp\text{-}cdcl_W\text{-}restart$ **by** $blast$
then have
 $cdcl_W\text{-restart}\text{-}M\text{-level}\text{-}inv T$ **and**
 $no\text{-}strange\text{-}atm T$
using $\langle cdcl_W\text{-restart}^{**} S T \rangle$ **apply** $(simp\ add: assms(1)\ rtranclp\text{-}cdcl_W\text{-}restart\text{-}consistent\text{-}inv)$
using $\langle cdcl_W\text{-restart}^{**} S T \rangle$ $assms(2)\ rtranclp\text{-}cdcl_W\text{-}restart\text{-}no\text{-}strange\text{-}atm\text{-}inv\ lev$ **by** $blast$
then have $no\text{-}step\ cdcl_W\text{-}cp T$
using T **unfolding** $full\text{-}def$ **by** $auto$
ultimately show $thesis$ **using** $that$ **unfolding** $full\text{-}def$ **by** $blast$
qed

lemma $always\text{-}exists\text{-}full\text{-}cdcl_W\text{-}cp\text{-}step$:

assumes $no\text{-}strange\text{-}atm S$

shows $\exists S''. full\ cdcl_W\text{-}cp S S''$

using $assms$

proof $(induct\ card\ (atms\text{-}of\text{-}mm\ (init\text{-}clss\ S) - atm\text{-}of\ 'lits\text{-}of\text{-}l\ (trail\ S))\ arbitrary: S)$

case 0 **note** $card = this(1)$ **and** $alien = this(2)$

then have $atm: atms\text{-}of\text{-}mm\ (init\text{-}clss\ S) = atm\text{-}of\ 'lits\text{-}of\text{-}l\ (trail\ S)$

unfolding $no\text{-}strange\text{-}atm\text{-}def$ **by** $auto$

{ assume $a: \exists S'. conflict\ S S'$

then obtain S' **where** $S': conflict\ S S'$ **by** $metis$

then have $\forall S''. \neg cdcl_W\text{-}cp\ S' S''$

by $(auto\ simp: cdcl_W\text{-}cp.simps\ elim!: conflictE\ propagateE$

$simp\ del: state\text{-}simp\ simp: state\text{-}eq\text{-}def)$

then have $?case$ **using** $a\ S'\ cdcl_W\text{-}cp.conflict'$ **unfolding** $full\text{-}def$ **by** $blast$

}

moreover **{**

assume $a: \exists S'. propagate\ S S'$

then obtain S' **where** $propagate\ S S'$ **by** $blast$

then obtain $E\ L$ **where**

$S: conflicting\ S = None$ **and**

$E: E \in \# clauses\ S$ **and**

$LE: L \in \# E$ **and**

$tr: trail\ S \models_{as} CNot\ (E - \{\#L\#})$ **and**

$undef: undefined\text{-}lit\ (trail\ S)\ L$ **and**

$S': S' \sim cons\text{-}trail\ (Propagated\ L\ E)\ S$

by $(elim\ propagateE)\ simp$

have $atms\text{-}of\text{-}mm\ (learned\text{-}clss\ S) \subseteq atms\text{-}of\text{-}mm\ (init\text{-}clss\ S)$

using $alien\ S$ **unfolding** $no\text{-}strange\text{-}atm\text{-}def$ **by** $auto$

then have $atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (init\text{-}clss\ S)$

using $E\ LE\ S\ undef$ **unfolding** $clauses\text{-}def$ **by** $(force\ simp: in\text{-}implies\text{-}atm\text{-}of\text{-}on\text{-}atms\text{-}of\text{-}ms)$

then have $False$ **using** $undef\ S$ **unfolding** atm **unfolding** $lits\text{-}of\text{-}def$

by $(auto\ simp\ add: defined\text{-}lit\text{-}map)$

}

ultimately show $?case$ **unfolding** $full\text{-}def$ **by** $(metis\ cdcl_W\text{-}cp.cases\ rtranclp.rtrancl\text{-}refl)$

next

case $(Suc\ n)$ **note** $IH = this(1)$ **and** $card = this(2)$ **and** $alien = this(3)$

{ assume $a: \exists S'. conflict\ S S'$

then obtain S' **where** $S': conflict\ S S'$ **by** $metis$

then have $\forall S''. \neg cdcl_W\text{-}cp\ S' S''$

by $(auto\ simp: cdcl_W\text{-}cp.simps\ elim!: conflictE\ propagateE$

$simp\ del: state\text{-}simp\ simp: state\text{-}eq\text{-}def)$

then have $?case$ **unfolding** $full\text{-}def\ Ex\text{-}def$ **using** $S'\ cdcl_W\text{-}cp.conflict'$ **by** $blast$

}

```

moreover {
  assume  $a: \exists S'. \text{propagate } S S'$ 
  then obtain  $S'$  where  $\text{propagate: propagate } S S'$  by  $\text{blast}$ 
  then obtain  $E L$  where
     $S: \text{conflicting } S = \text{None}$  and
     $E: E \in \# \text{ clauses } S$  and
     $LE: L \in \# E$  and
     $\text{tr: trail } S \models_{\text{as}} \text{CNot } (E - \{\#L\#})$  and
     $\text{undef: undefined-lit } (\text{trail } S) L$  and
     $S': S' \sim \text{cons-trail } (\text{Propagated } L E) S$ 
    by  $(\text{elim propagate } E)$   $\text{simp}$ 
  then have  $\text{atm-of } L \notin \text{atm-of ' lits-of-l } (\text{trail } S)$ 
  unfolding  $\text{lits-of-def}$  by  $(\text{auto simp add: defined-lit-map})$ 
  moreover
    have  $\text{no-strange-atm } S'$  using  $\text{alien propagate propagate-no-strange-atm-inv}$  by  $\text{blast}$ 
    then have  $\text{atm-of } L \in \text{atms-of-mm } (\text{init-clss } S)$ 
    using  $S' LE E \text{undef}$  unfolding  $\text{no-strange-atm-def}$ 
    by  $(\text{auto simp: clauses-def in-implies-atm-of-on-atms-of-ms})$ 
    then have  $\bigwedge A. \{\text{atm-of } L\} \subseteq \text{atms-of-mm } (\text{init-clss } S) - A \vee \text{atm-of } L \in A$  by  $\text{force}$ 
  moreover have  $\text{Suc } n - \text{card } \{\text{atm-of } L\} = n$  by  $\text{simp}$ 
  moreover have  $\text{card } (\text{atms-of-mm } (\text{init-clss } S) - \text{atm-of ' lits-of-l } (\text{trail } S)) = \text{Suc } n$ 
  using  $\text{card } S S'$  by  $\text{simp}$ 
  ultimately
    have  $\text{card } (\text{atms-of-mm } (\text{init-clss } S) - \text{atm-of ' insert } L (\text{lits-of-l } (\text{trail } S))) = n$ 
    by  $(\text{metis } (\text{no-types}) \text{Diff-insert card-Diff-subset finite.emptyI finite.insertI image-insert})$ 
    then have  $n = \text{card } (\text{atms-of-mm } (\text{init-clss } S') - \text{atm-of ' lits-of-l } (\text{trail } S'))$ 
    using  $\text{card } S S' \text{undef}$  by  $\text{simp}$ 
  then have  $a1: \text{Ex } (\text{full cdcl}_W\text{-cp } S')$  using  $IH \langle \text{no-strange-atm } S' \rangle$  by  $\text{blast}$ 
  have  $?case$ 
  proof –
    obtain  $S'' :: 'st$  where
       $\text{ff1: cdcl}_W\text{-cp}^{**} S' S'' \wedge \text{no-step cdcl}_W\text{-cp } S''$ 
      using  $a1$  unfolding  $\text{full-def}$  by  $\text{blast}$ 
    have  $\text{cdcl}_W\text{-cp}^{**} S S''$ 
    using  $\text{ff1 cdcl}_W\text{-cp.intros}(2)[OF \text{propagate}]$ 
    by  $(\text{metis } (\text{no-types}) \text{converse-rtranclp-into-rtranclp})$ 
    then have  $\exists S''. \text{cdcl}_W\text{-cp}^{**} S S'' \wedge (\forall S'''. \neg \text{cdcl}_W\text{-cp } S'' S''')$ 
    using  $\text{ff1}$  by  $\text{blast}$ 
    then show  $?thesis$  unfolding  $\text{full-def}$ 
    by  $\text{meson}$ 
  qed
}
ultimately show  $?case$  unfolding  $\text{full-def}$  by  $(\text{metis cdcl}_W\text{-cp.cases rtranclp.rtrancl-refl})$ 
qed

```

Literal of highest level in conflicting clauses

One important property of the $\text{cdcl}_W\text{-restart}$ with strategy is that, whenever a conflict takes place, there is at least a literal of level k involved (except if we have derived the false clause). The reason is that we apply conflicts before a decision is taken.

abbreviation $\text{no-clause-is-false} :: 'st \Rightarrow \text{bool}$ **where**

$\text{no-clause-is-false} \equiv$

$\lambda S. (\text{conflicting } S = \text{None} \longrightarrow (\forall D \in \# \text{ clauses } S. \neg \text{trail } S \models_{\text{as}} \text{CNot } D))$

abbreviation $\text{conflict-is-false-with-level} :: 'st \Rightarrow \text{bool}$ **where**

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

proof (*clarify*)

fix D

assume $D \in \# \text{local.clauses } S$ **and** *conflicting* $S = \text{None}$ **and** *trail* $S \models_{\text{as}} \text{CNot } D$

then show *False*

using *conflict-rule*[*of* S D *update-conflicting* ($\text{Some } D$) S] *assms*

by *auto*

qed

lemma *full-cdcl_W-cp-not-any-negated-init-clss:*

assumes *full cdcl_W-cp* $S S'$

shows *no-clause-is-false* S'

using *assms not-conflict-not-any-negated-init-clss* **unfolding** *full-def* **by** *auto*

lemma *full1-cdcl_W-cp-not-any-negated-init-clss:*

assumes *full1 cdcl_W-cp* $S S'$

shows *no-clause-is-false* S'

using *assms not-conflict-not-any-negated-init-clss* **unfolding** *full1-def* **by** *auto*

lemma *cdcl_W-stgy-not-non-negated-init-clss:*

assumes *cdcl_W-stgy* $S S'$

shows *no-clause-is-false* S'

using *assms apply* (*induct rule: cdcl_W-stgy.induct*)

using *full1-cdcl_W-cp-not-any-negated-init-clss* *full-cdcl_W-cp-not-any-negated-init-clss* **by** *metis+*

lemma *rtranclp-cdcl_W-stgy-not-non-negated-init-clss:*

assumes *cdcl_W-stgy*** $S S'$ **and** *no-clause-is-false* S

shows *no-clause-is-false* S'

using *assms* **by** (*induct rule: rtranclp-induct*) (*auto simp: cdcl_W-stgy-not-non-negated-init-clss*)

lemma *cdcl_W-stgy-conflict-ex-lit-of-max-level:*

assumes

cdcl_W-cp $S S'$ **and**

no-clause-is-false S **and**

cdcl_W-restart-M-level-inv S

shows *conflict-is-false-with-level* S'

using *assms*

proof (*induct rule: cdcl_W-cp.induct*)

case *conflict'*

then show *?case* **by** (*auto elim: conflictE*)

next

case *propagate'*

then show *?case* **by** (*auto elim: propagateE*)

qed

lemma *no-chained-conflict:*

assumes *conflict* $S S'$ **and** *conflict* $S' S''$

shows *False*

using *assms* **unfolding** *conflict.simps*

by (*metis conflicting-update-conflicting option.distinct(1) state-eq-conflicting*)

```

lemma rtrancp-cdclW-cp-propa-or-propa-conf:
  assumes cdclW-cp** S U
  shows propagate** S U  $\vee$  ( $\exists T. \text{propagate**} S T} \wedge \text{conflict T U}$ )
  using assms
proof induction
  case base
  then show ?case by auto
next
  case (step U V) note SU = this(1) and UV = this(2) and IH = this(3)
  consider (conf) T where propagate** S T and conflict T U
    | (propa) propagate** S U using IH by auto
  then show ?case
  proof cases
    case conf
    then have False using UV by (auto elim: conflictE)
    then show ?thesis by fast
  next
    case propa
    also have conflict U V  $\vee$  propagate U V using UV by (auto simp add: cdclW-cp.simps)
    ultimately show ?thesis by force
  qed
qed

lemma rtrancp-cdclW-restart-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-restart-M-level-inv S
  shows conflict-is-false-with-level U
proof (intro allI impI)
  fix D
  assume
    confl: conflicting U = Some D and
    D: D  $\neq$  {#}
  consider (CT) conflicting S = None | (SD) D' where conflicting S = Some D'
  by (cases conflicting S) auto
  then show  $\exists L \in \#D. \text{get-level (trail U) L} = \text{backtrack-lvl U}$ 
  proof cases
    case SD
    then have S = U
    by (metis (no-types) assms(1) cdclW-cp-conflicting-not-empty full-def rtrancpD trancpD)
    then show ?thesis using assms(3) confl D by blast-
  next
    case CT
    have init-clss U = init-clss S and learned-clss U = learned-clss S
    using full unfolding full-def
    apply (metis (no-types) rtrancpD trancp-cdclW-cp-no-more-init-clss)
    by (metis (mono-tags, lifting) full full-def rtrancp-cdclW-cp-learned-clause-inv)
    obtain T where propagate** S T and TU: conflict T U
    proof  $-$ 
      have f5: U  $\neq$  S
      using confl CT by force
      then have cdclW-cp++ S U
      by (metis full full-def rtrancpD)
      have  $\bigwedge p \text{ pa. } \neg \text{propagate p pa} \vee \text{conflicting pa} =$ 
        (None :: 'v clause option)

```

```

    by (auto elim: propagateE)
  then show ?thesis
    using f5 that tranclp-cdclW-cp-propagate-with-conflict-or-not[OF ‹cdclW-cp++ S U›]
    full confI CT unfolding full-def by auto
qed
obtain D' where
  conflicting T = None and
  D': D' ∈ # clauses T and
  tr: trail T ⊨as CNot (D') and
  U: U ∼ update-conflicting (Some (D')) T
  using TU by (auto elim!: conflictE)
have init-clss T = init-clss S and learned-clss T = learned-clss S
  using U ‹init-clss U = init-clss S› ‹learned-clss U = learned-clss S› by auto
then have D ∈ # clauses S
  using confI U D' by (auto simp: clauses-def)
then have ¬ trail S ⊨as CNot D
  using cls-f CT by simp

moreover
  obtain M where tr-U: trail U = M @ trail S and nm: ∀ m ∈ set M. ¬ is-decided m
    by (metis (mono-tags, lifting) assms(1) full-def rtranclp-cdclW-cp-dropWhile-trail)
  have trail U ⊨as CNot D
    using tr confI U by (auto elim!: conflictE)
ultimately obtain L where L ∈ # D and ¬ L ∈ lits-of-l M
  unfolding tr-U CNot-def true-annots-def Ball-def true-annot-def true-clss-def by force

moreover have inv-U: cdclW-restart-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: rtranclp-cdclW-cp-backtrack-lvl)

moreover
  have no-dup (trail U)
    using inv-U unfolding cdclW-restart-M-level-inv-def by auto
  { fix x :: ('v, 'v clause) ann-lit and
    xb :: ('v, 'v clause) ann-lit
    assume a1: atm-of L = atm-of (lit-of xb)
    moreover assume a2: ¬ L = lit-of x
    moreover assume a3: (λl. atm-of (lit-of l)) ‹ set M
      ∩ (λl. atm-of (lit-of l)) ‹ set (trail S) = {}
    moreover assume a4: x ∈ set M
    moreover assume a5: xb ∈ set (trail S)
    moreover have atm-of (¬ L) = atm-of L
      by auto
    ultimately have False
      by auto
  }
  then have LS: atm-of L ∉ atm-of ‹ lits-of-l (trail S)
    using ‹¬ L ∈ lits-of-l M› ‹no-dup (trail U)› unfolding tr-U lits-of-def by auto
ultimately have get-level (trail U) L = backtrack-lvl U
proof (cases count-decided (trail S) ≠ 0, 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-restart-M-level-inv-def by auto

```

```

moreover have get-level  $M\ L = 0$ 
  using nm by auto
ultimately show ?thesis using  $LS\ ne\ US$  unfolding tr-U
  by (simp add: lits-of-def filter-empty-conv)
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 count-decided (trail  $S$ ) = backtrack-lvl  $S$ 
    using  $ne\ lev$  unfolding cdclW-restart-M-level-inv-def by auto
  moreover have atm-of  $L \in atm-of\ 'lits-of-l\ M$ 
    using  $\langle -L \in lits-of-l\ M \rangle$  by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      lits-of-def)
  ultimately show ?thesis
    using  $nm\ ne\ get-level-skip-in-all-not-decided[of\ M\ L]$  unfolding lits-of-def  $US\ tr-U$ 
    by auto
  qed
then show  $\exists L \in \#D. get-level\ (trail\ U)\ L = backtrack-lvl\ U$ 
  using  $\langle L \in \#D \rangle$  by blast
qed
qed

```

Literal of highest level in decided literals

definition *mark-is-false-with-level* :: $'st \Rightarrow bool$ **where**

mark-is-false-with-level $S' \equiv$

$\forall D\ M1\ M2\ L. M1\ @\ Propagated\ L\ D\ \# \ M2 = trail\ S' \longrightarrow D - \{\#L\# \} \neq \{\# \}$
 $\longrightarrow (\exists L. L \in \#D \wedge get-level\ (trail\ S')\ L = count-decided\ M1)$

definition *no-more-propagation-to-do* :: $'st \Rightarrow bool$ **where**

no-more-propagation-to-do $S \equiv$

$\forall D\ M\ M'\ L. D + \{\#L\# \} \in \# clauses\ S \longrightarrow trail\ S = M' @ M \longrightarrow M \models_{as} CNot\ D$
 $\longrightarrow undefined-lit\ M\ L \longrightarrow count-decided\ M < backtrack-lvl\ S$
 $\longrightarrow (\exists L. L \in \#D \wedge get-level\ (trail\ S)\ L = count-decided\ M)$

lemma *propagate-no-more-propagation-to-do*:

assumes *propagate*: *propagate* $S\ S'$
and H : *no-more-propagation-to-do* S
and *lev-inv*: *cdcl_W-restart-M-level-inv* S
shows *no-more-propagation-to-do* S'
using *assms*

proof –

obtain $E\ L$ **where**
 S : *conflicting* $S = None$ **and**
 E : $E \in \# clauses\ S$ **and**
 LE : $L \in \# E$ **and**
 tr : $trail\ S \models_{as} CNot\ (E - \{\#L\# \})$ **and**
 $undefL$: *undefined-lit* (*trail* S) L **and**
 S' : $S' \sim cons-trail\ (Propagated\ L\ E)\ S$
using *propagate* **by** (*elim propagateE*) *simp*
let $?M' = Propagated\ L\ E\ \# trail\ S$
show *?thesis* **unfolding** *no-more-propagation-to-do-def*
proof (*intro allI impI*)
fix $D\ M1\ M2\ L'$
assume
 $D-L$: $D + \{\#L'\# \} \in \# clauses\ S'$ **and**

```

    trail S' = M2 @ M1 and
    get-max: count-decided M1 < backtrack-lvl S' and
    M1  $\models_{as}$  CNot D and
    undef: undefined-lit M1 L'
have tl M2 @ M1 = trail S  $\vee$  (M2 = []  $\wedge$  M1 = Propagated L E # trail S)
  using (trail S' = M2 @ M1) S' S undefL lev-inv
  by (cases M2) (auto simp:cdclW-restart-M-level-inv-decomp)
moreover {
  assume tl M2 @ M1 = trail S
  moreover have D + {#L'#}  $\in$  # clauses S
    using D-L S S' undefL unfolding clauses-def by auto
  moreover have count-decided M1 < backtrack-lvl S
    using get-max S S' undefL by auto
  ultimately obtain L' where L'  $\in$  # D and
    get-level (trail S) L' = count-decided M1
    using H (M1  $\models_{as}$  CNot D) undef unfolding no-more-propagation-to-do-def by metis
  moreover
    { have cdclW-restart-M-level-inv S'
      using cdclW-restart-consistent-inv lev-inv cdclW-restart.propagate[OF propagate] by blast
      then have no-dup ?M' using S' undefL unfolding cdclW-restart-M-level-inv-def by auto
      moreover
        have atm-of L'  $\in$  atm-of ' (lits-of-l M1)
          using (L'  $\in$  # D) (M1  $\models_{as}$  CNot D) by (metis atm-of-uminus image-eqI
            in-CNot-implies-uminus(2))
        then have atm-of L'  $\in$  atm-of ' (lits-of-l (trail S))
          using (tl M2 @ M1 = trail S)[symmetric] S undefL by auto
        ultimately have atm-of L  $\neq$  atm-of L' unfolding lits-of-def by auto
      }
    ultimately have  $\exists L' \in \# D. \text{get-level (trail S')} L' = \text{count-decided M1}$ 
      using S S' undefL by auto
  }
  moreover {
    assume M2 = [] and M1: M1 = Propagated L E # trail S
    have cdclW-restart-M-level-inv S'
      using cdclW-restart-consistent-inv[OF lev-inv] cdclW-restart.propagate[OF propagate] by blast
    then have count-decided M1 = backtrack-lvl S'
      using S' M1 undefL unfolding cdclW-restart-M-level-inv-def by (auto intro: Max-eqI)
    then have False using get-max by auto
  }
  ultimately show  $\exists L. L \in \# D \wedge \text{get-level (trail S')} L = \text{count-decided M1}$ 
    by fast
qed
qed

```

lemma conflict-no-more-propagation-to-do:

```

assumes
  conflict: conflict S S' and
  H: no-more-propagation-to-do S and
  M: cdclW-restart-M-level-inv S
shows no-more-propagation-to-do S'
using assms unfolding no-more-propagation-to-do-def by (force elim!: conflictE)

```

lemma cdcl_W-cp-no-more-propagation-to-do:

```

assumes
  conflict: cdclW-cp S S' and
  H: no-more-propagation-to-do S and

```


M : $cdcl_W$ -restart- M -level-inv S
shows *no-more-propagation-to-do* S'
using *assms*
proof (*induct rule*: $cdcl_W$ -cp.induct)
case (*conflict'* $S S'$)
then show ?case **using** *conflict-no-more-propagation-to-do*[*of* $S S'$] **by** *blast*
next
case (*propagate'* $S S'$) **note** $S = this$
show 1: *no-more-propagation-to-do* S'
using *propagate-no-more-propagation-to-do*[*of* $S S'$] S **by** *blast*
qed

lemma $cdcl_W$ -restart-then-exists- $cdcl_W$ -stgy-step:
assumes
 o : $cdcl_W$ -o $S S'$ **and**
 $alien$: *no-strange-atm* S **and**
 lev : $cdcl_W$ -restart- M -level-inv S
shows $\exists S'. cdcl_W$ -stgy $S S'$
proof –
obtain S'' **where** *full* $cdcl_W$ -cp $S' S''$
using *always-exists-full- $cdcl_W$ -cp-step* $alien$ $cdcl_W$ -restart-*no-strange-atm-inv* $cdcl_W$ -o-*no-more-init-clss*
 o *other* lev **by** (*meson* $cdcl_W$ -restart-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 : *conflicting* $S = Some E$ **and**
 LE : $L \in \# E$ **and**
 L : *get-level* (*trail* S) $L = backtrack$ -lvl S **and**
 D : *get-maximum-level* (*trail* S) (*remove1-mset* $L E$) < *backtrack-lvl* S **and**
 bt : *backtrack-lvl* $S = get$ -maximum-level (*trail* S) E **and**
 $M-L$: $cdcl_W$ -restart- M -level-inv S
shows $\exists S'. cdcl_W$ -o $S S'$
proof –
have $L-D$: *get-level* (*trail* S) $L = get$ -maximum-level (*trail* S) E
using $L D bt$ **by** (*simp* *add*: *get-maximum-level-plus*)
let ? $i = get$ -maximum-level (*trail* S) (*remove1-mset* $L E$)
obtain $K M1 M2$ **where**
 K : (*Decided* $K \# M1, M2$) $\in set$ (*get-all-ann-decomposition* (*trail* S)) **and**
 $lev-K$: *get-level* (*trail* S) $K = Suc$? i
using *backtrack-ex-decomp*[*OF* $M-L$, *of* ? i] $D S$ **by** *auto*
show ?thesis **using** *backtrack-rule*[*OF* $S LE K L$, *of* ? i] $bt L lev-K bj$ **by** (*auto simp*: $cdcl_W$ -bj.simps)
qed

lemma $cdcl_W$ -stgy-final-state-conclusive:
assumes
 $termi$: $\forall S'. \neg cdcl_W$ -stgy $S S'$ **and**
 $decomp$: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) **and**
 $learned$: $cdcl_W$ -restart-learned-clause S **and**
 $level$ -inv: $cdcl_W$ -restart- M -level-inv S **and**
 $alien$: *no-strange-atm* S **and**
 no -dup: *distinct- $cdcl_W$ -restart-state* S **and**
 $confl$: $cdcl_W$ -restart-conflicting S **and**
 $confl-k$: *conflict-is-false-with-level* S

```

shows (conflicting  $S = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$ )
   $\vee (\text{conflicting } S = \text{None} \wedge \text{trail } S \models_{\text{as}} \text{set-mset } (\text{init-clss } S))$ 
proof –
  let  $?M = \text{trail } S$ 
  let  $?N = \text{init-clss } S$ 
  let  $?k = \text{backtrack-lvl } S$ 
  let  $?U = \text{learned-clss } S$ 
  consider
    (None) conflicting  $S = \text{None}$ 
  | (Some-Empty)  $E$  where conflicting  $S = \text{Some } E$  and  $E = \{\#\}$ 
  | (Some)  $E'$  where conflicting  $S = \text{Some } E'$  and
    conflicting  $S = \text{Some } (E')$  and  $E' \neq \{\#\}$ 
  by (cases conflicting S, simp) auto
then show  $?thesis$ 
proof cases
  case (Some-Empty  $E$ )
  then have conflicting  $S = \text{Some } \{\#\}$  by auto
  then have unsatisfiable (set-mset (init-clss  $S$ ))
    using assms(3) unfolding cdclW-restart-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)
  then show  $?thesis$  using Some-Empty by auto
next
case None
  { assume  $\neg ?M \models_{\text{asm}} ?N$ 
    have atm-of ‘ (lits-of-l  $?M$ ) = atms-of-mm  $?N$  (is  $?A = ?B$ )
    proof
      show  $?A \subseteq ?B$  using alien unfolding no-strange-atm-def by auto
      show  $?B \subseteq ?A$ 
      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)
        moreover have conflicting  $S = \text{None}$ 
          using None by auto
        ultimately have  $\exists S'. \text{cdcl}_W\text{-o } S S'$ 
          using cdclW-o.decide decide-rule  $\langle l \in ?B \rangle$  no-strange-atm-def
          by (metis literal.sel(1) state-eq-def)
        then show False
          using termi cdclW-restart-then-exists-cdclW-stgy-step[OF - alien] level-inv by blast
      qed
    }
  qed
obtain  $D$  where  $\neg ?M \models_a D$  and  $D \in \# ?N$ 
  using  $\langle \neg ?M \models_{\text{asm}} ?N \rangle$  unfolding lits-of-def true-annots-def Ball-def by auto
have atms-of  $D \subseteq \text{atm-of } \langle \text{lits-of-l } ?M \rangle$ 
  using  $\langle D \in \# ?N \rangle$  unfolding atm-of ‘ (lits-of-l  $?M$ ) = atms-of-mm  $?N$  atms-of-ms-def
  by (auto simp add: atms-of-def)
then have  $a1: \text{atm-of } \langle \text{set-mset } D \subseteq \text{atm-of } \langle \text{lits-of-l } (\text{trail } S) \rangle$ 
  by (auto simp add: atms-of-def lits-of-def)
have total-over-m (lits-of-l  $?M$ )  $\{D\}$ 
  using  $\langle \text{atms-of } D \subseteq \text{atm-of } \langle \text{lits-of-l } ?M \rangle$ 
  atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set by (fastforce simp: total-over-set-def)
then have  $?M \models_{\text{as}} \text{CNot } D$ 
  using total-not-true-clss-true-clss-CNot  $\langle \neg \text{trail } S \models_a D \rangle$  true-annot-def
  true-annots-true-clss by fastforce

```

```

then have False
proof -
  obtain S' where
    f2: full cdclW-cp S S'
    by (meson alien always-exists-full-cdclW-cp-step level-inv)
  then have S' = S
    using cdclW-stgy.conflict'[of S] by (metis (no-types) full-unfold termi)
  then show ?thesis
    using f2 ⟨D ∈ # init-clss S⟩ None ⟨trail S ⊨as CNot D⟩
    clauses-def full-cdclW-cp-not-any-negated-init-clss by auto
qed
}
then have ?M ⊨asm ?N by blast
then show ?thesis
  using None by auto
next
case (Some E') note conf = this(1) and LD = this(2) and nempty = this(3)
then obtain L D where
  E'[simp]: E' = D + {#L#} and
  lev-L: get-level ?M L = ?k
  by (metis (mono-tags) confl-k insert-DiffM2)
let ?D = D + {#L#}
have ?D ≠ {#} by auto
have ?M ⊨as CNot ?D using confl LD unfolding cdclW-restart-conflicting-def by auto
then have ?M ≠ [] unfolding true-annots-def Ball-def true-annot-def true-clss-def by force
have M: ?M = hd ?M # tl ?M using ⟨?M ≠ []⟩ list.collapse by fastforce

have g-k: get-maximum-level (trail S) D ≤ ?k
  using count-decided-ge-get-maximum-level[of ?M] level-inv
  unfolding cdclW-restart-M-level-inv-def
  by auto
{
  assume decided: is-decided (hd ?M)
  then obtain k' where k': k' + 1 = ?k
    using level-inv M unfolding cdclW-restart-M-level-inv-def
    by (cases hd (trail S); cases trail S) auto
  obtain L' where L': hd ?M = Decided L' using decided by (cases hd ?M) auto
  have *: ∧list. no-dup list ⇒
    - L ∈ lits-of-l list ⇒ atm-of L ∈ atm-of 'lits-of-l list
    by (metis atm-of-uminus imageI)
  have L'-L: L' = -L
  proof (rule ccontr)
    assume ¬ ?thesis
    moreover have -L ∈ lits-of-l ?M using confl LD unfolding cdclW-restart-conflicting-def
    by auto
    ultimately have get-level (hd (trail S) # tl (trail S)) L = get-level (tl ?M) L
      using cdclW-restart-M-level-inv-decomp(1)[OF level-inv] unfolding consistent-interp-def
      by (subst (asm) (2) M) (auto simp add: atm-of-eq-atm-of L')
    moreover
      have count-decided (trail S) = ?k
        using level-inv unfolding cdclW-restart-M-level-inv-def by auto
      then have count: count-decided (tl (trail S)) = ?k - 1
        using level-inv unfolding cdclW-restart-M-level-inv-def
        by (subst (asm) M) (auto simp add: L')
      then have get-level (tl ?M) L < ?k
        using count-decided-ge-get-level[of L tl ?M] unfolding count k'[symmetric]

```

```

    by auto
    finally show False using lev-L M by auto
qed
have L: hd ?M = Decided (-L) using L'-L L' by auto

have get-maximum-level (trail S) D < ?k
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then have get-maximum-level (trail S) D = ?k using M g-k unfolding L by auto
  then obtain L'' where  $L'' \in \# D$  and L-k: get-level ?M L'' = ?k
    using get-maximum-level-exists-lit[of ?k ?M D] unfolding k'[symmetric] by auto
  have  $L \neq L''$  using no-dup <L'' ∈ # D>
    unfolding distinct-cdclW-restart-state-def LD
    by (metis E' add.right-neutral add-diff-cancel-right'
      distinct-mem-diff-mset union-commute union-single-eq-member)
  have  $L'' = -L$ 
  proof (rule ccontr)
    assume  $\neg ?thesis$ 
    then have get-level ?M L'' = get-level (tl ?M) L''
      using M <L ≠ L''> get-level-skip-beginning[of L'' hd ?M tl ?M] unfolding L
      by (auto simp: atm-of-eq-atm-of)
    moreover
      have d: dropWhile (λS. atm-of (lit-of S) ≠ atm-of L) (tl (trail S)) = []
        using level-inv unfolding cdclW-restart-M-level-inv-def apply (subst (asm)(2) M)
        by (auto simp: image-iff L' L'-L)
      have get-level (tl (trail S)) L = 0
        by (auto simp: filter-empty-conv d)
    moreover
      have get-level (tl (trail S)) L'' ≤ count-decided (tl (trail S))
        by auto
      then have get-level (tl (trail S)) L'' < backtrack-lvl S
        using level-inv unfolding cdclW-restart-M-level-inv-def apply (subst (asm)(5) M)
        by (auto simp: image-iff L' L'-L simp del: count-decided-ge-get-level)
    ultimately show False
      apply -
      apply (subst (asm) M, subst (asm)(3) M, subst (asm) L')
      using L-k
      apply (auto simp: L' L'-L split: if-splits)
      apply (subst (asm)(3) M, subst (asm) L')
      using  $\langle L'' \neq -L \rangle$  by (auto simp: L' L'-L split: if-splits)
  qed
  then have taut: tautology (D + {#L#})
    using  $\langle L'' \in \# D \rangle$  by (metis add commute mset-leD mset-le-add-left multi-member-this
      tautology-minus)
  have consistent-interp (lits-of-l ?M)
    using level-inv unfolding cdclW-restart-M-level-inv-def by auto
  then have  $\neg ?M \models_{as} CNot ?D$ 
    using taut by (metis <L'' = - L> <L'' ∈ # D> add commute consistent-interp-def
      diff-union-cancelR in-CNot-implies-uminus(2) in-diffD multi-member-this)
  moreover have  $?M \models_{as} CNot ?D$ 
    using confl no-dup LD unfolding cdclW-restart-conflicting-def by auto
  ultimately show False by blast
qed note H = this
have get-maximum-level (trail S) D < get-maximum-level (trail S) (D + {#L#})
  using H by (auto simp: get-maximum-level-plus lev-L max-def)
moreover have backtrack-lvl S = get-maximum-level (trail S) (D + {#L#})

```

```

    using H by (auto simp: get-maximum-level-plus lev-L max-def)
ultimately have False
    using backtrack-no-decomp[OF conf - lev-L] level-inv termi
    cdclW-restart-then-exists-cdclW-stgy-step[of S] alien unfolding E'
    by (auto simp add: lev-L max-def)
} note not-is-decided = this

moreover {
  let ?D = D + {#L#}
  have ?D ≠ {#} by auto
  have ?M ⊨as CNot ?D using confl LD unfolding cdclW-restart-conflicting-def by auto
  then have ?M ≠ [] unfolding true-annots-def Ball-def true-annot-def true-cls-def by force
  assume nm: ¬is-decided (hd ?M)
  then obtain L' C where L'C: hd-trail S = Propagated L' C using ⟨trail S ≠ []⟩
    by (cases hd-trail S) auto
  then have hd ?M = Propagated L' C
    using ⟨trail S ≠ []⟩ by fastforce
  then have M: ?M = Propagated L' C # tl ?M
    using ⟨?M ≠ []⟩ list.collapse by fastforce
  then obtain C' where C': C = C' + {#L'#}
    using confl unfolding cdclW-restart-conflicting-def by (metis append-Nil diff-single-eq-union)
  { assume -L' ∈ # ?D
    then have Ex (skip S)
      using skip-rule[OF M conf] unfolding E' by auto
    then have False
      using cdclW-restart-then-exists-cdclW-stgy-step[of S] alien level-inv termi
      by (auto dest: cdclW-o.intros cdclW-bj.intros)
  }
  moreover {
    assume L'D: -L' ∈ # ?D
    then obtain D' where D': ?D = D' + {#-L'#} by (metis insert-DiffM2)
    then have get-maximum-level (trail S) D' ≤ ?k
      using count-decided-ge-get-maximum-level[of Propagated L' C # tl ?M] M
      level-inv unfolding cdclW-restart-M-level-inv-def by auto
    then have get-maximum-level (trail S) D' = ?k
      ∨ get-maximum-level (trail S) D' < ?k
      using le-neq-implies-less by blast
    moreover {
      assume g-D'-k: get-maximum-level (trail S) D' = ?k
      then have f1: get-maximum-level (trail S) D' = backtrack-lvl S
        using M by auto
      then have Ex (cdclW-o S)
        using f1 resolve-rule[of S L' C , OF ⟨trail S ≠ []⟩ - - conf] conf g-D'-k
        L'C L'D unfolding C' D' E'
        by (fastforce simp add: D' intro: cdclW-o.intros cdclW-bj.intros)
      then have False
        by (meson alien cdclW-restart-then-exists-cdclW-stgy-step termi level-inv)
    }
  }
  moreover {
    assume a1: get-maximum-level (trail S) D' < ?k
    then have f3: get-maximum-level (trail S) D' < get-level (trail S) (-L')
      using a1 lev-L by (metis D' get-maximum-level-ge-get-level insert-noteq-member
        not-less)
    moreover have backtrack-lvl S = get-level (trail S) L'
      apply (subst M)
      using level-inv unfolding cdclW-restart-M-level-inv-def

```

```

    by (subst (asm)(3) M) (auto simp add: cdclW-restart-M-level-inv-decomp)[]
  moreover
    then have get-level (trail S) L' = get-maximum-level (trail S) (D' + {#- L'#})
      using a1 by (auto simp add: get-maximum-level-plus max-def)
    ultimately have False
      using M backtrack-no-decomp[of S - L', OF conf]
      cdclW-restart-then-exists-cdclW-stgy-step L'D level-inv termi alien
      unfolding D' E' by auto
  }
  ultimately have False by blast
}
ultimately have False by blast
}
ultimately show ?thesis by blast
qed
qed

lemma cdclW-cp-tranclp-cdclW-restart:
  cdclW-cp S S'  $\implies$  cdclW-restart++ S S'
  apply (induct rule: cdclW-cp.induct)
  by (meson cdclW-restart.conflict cdclW-restart.propagate tranclp.r-into-trancl tranclp.trancl-into-trancl)+

lemma tranclp-cdclW-cp-tranclp-cdclW-restart:
  cdclW-cp++ S S'  $\implies$  cdclW-restart++ S S'
  apply (induct rule: tranclp.induct)
  apply (simp add: cdclW-cp-tranclp-cdclW-restart)
  by (meson cdclW-cp-tranclp-cdclW-restart tranclp-trans)

lemma cdclW-stgy-tranclp-cdclW-restart:
  cdclW-stgy S S'  $\implies$  cdclW-restart++ S S'
proof (induct rule: cdclW-stgy.induct)
  case conflict'
  then show ?case
    unfolding full1-def by (simp add: tranclp-cdclW-cp-tranclp-cdclW-restart)
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-restart.other tranclp.r-into-trancl
      tranclp-cdclW-cp-tranclp-cdclW-restart tranclp-trans)
qed

lemma tranclp-cdclW-stgy-tranclp-cdclW-restart:
  cdclW-stgy++ S S'  $\implies$  cdclW-restart++ S S'
  apply (induct rule: tranclp.induct)
  using cdclW-stgy-tranclp-cdclW-restart apply blast
  by (meson cdclW-stgy-tranclp-cdclW-restart tranclp-trans)

lemma rtranclp-cdclW-stgy-rtranclp-cdclW-restart:
  cdclW-stgy** S S'  $\implies$  cdclW-restart** S S'
  using rtranclp-unfold[of cdclW-stgy S S'] tranclp-cdclW-stgy-tranclp-cdclW-restart[of S S'] by auto

lemma not-empty-get-maximum-level-exists-lit:
  assumes n: D  $\neq$  {#}
  and max: get-maximum-level M D = n

```

shows $\exists L \in \#D. \text{get-level } M \ L = n$

proof –

have $f: \text{finite } (\text{insert } 0 \ ((\lambda L. \text{get-level } M \ L) \text{ 'set-mset } D))$ **by** *auto*

then have $n \in ((\lambda L. \text{get-level } M \ L) \text{ 'set-mset } D)$

using $n \text{ max get-maximum-level-exists-lit-of-max-level image-iff}$

unfolding *get-maximum-level-def* **by** *force*

then show $\exists L \in \#D. \text{get-level } M \ L = n$ **by** *auto*

qed

lemma *cdcl_W-o-conflict-is-false-with-level-inv*:

assumes

cdcl_W-o S S' and

lev: cdcl_W-restart-M-level-inv S and

confl-inv: conflict-is-false-with-level S and

n-d: distinct-cdcl_W-restart-state S and

conflicting: cdcl_W-restart-conflicting S

shows *conflict-is-false-with-level S'*

using *assms(1,2)*

proof (*induct rule: cdcl_W-o-induct*)

case (*resolve L C M D T*) **note** $\text{tr-S} = \text{this}(1)$ **and** $\text{confl} = \text{this}(4)$ **and** $\text{LD} = \text{this}(5)$ **and** $T = \text{this}(7)$

have $\text{uL-not-D}: -L \notin \# \text{remove1-mset } (-L) \ D$

using $n\text{-d confl}$ **unfolding** *distinct-cdcl_W-restart-state-def distinct-mset-def*

by (*metis distinct-cdcl_W-restart-state-def distinct-mem-diff-mset multi-member-last n-d*)

moreover have $\text{L-not-D}: L \notin \# \text{remove1-mset } (-L) \ D$

proof (*rule ccontr*)

assume $\neg ?thesis$

then have $L \in \# D$

by (*auto simp: in-remove1-mset-neq*)

moreover have $\text{Propagated } L \ C \ \# \ M \models_{\text{as}} \text{CNot } D$

using *conflicting confl tr-S* **unfolding** *cdcl_W-restart-conflicting-def* **by** *auto*

ultimately have $-L \in \text{lits-of-l } (\text{Propagated } L \ C \ \# \ M)$

using *in-CNot-implies-uminus(2)* **by** *blast*

moreover have $\text{no-dup } (\text{Propagated } L \ C \ \# \ M)$

using *lev tr-S* **unfolding** *cdcl_W-restart-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 ann-lit.sel(2) distinct-consistent-interp*)

qed

ultimately

have $\text{g-D}: \text{get-maximum-level } (\text{Propagated } L \ C \ \# \ M) \ (\text{remove1-mset } (-L) \ D)$

$= \text{get-maximum-level } M \ (\text{remove1-mset } (-L) \ D)$

using *get-maximum-level-skip-first[of L remove1-mset (-L) D C M]*

by (*simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def*)

have $\text{lev-L[simp]}: \text{get-level } M \ L = 0$

apply (*rule atm-of-notin-get-rev-level-eq-0*)

using *lev* **unfolding** *cdcl_W-restart-M-level-inv-def tr-S* **by** (*auto simp: lits-of-def*)

have $D: \text{get-maximum-level } M \ (\text{remove1-mset } (-L) \ D) = \text{backtrack-lvl } S$

using *resolve.hyps(6) LD* **unfolding** *tr-S* **by** (*auto simp: get-maximum-level-plus max-def g-D*)

have $\text{get-maximum-level } M \ (\text{remove1-mset } L \ C) \leq \text{backtrack-lvl } S$

using *count-decided-ge-get-maximum-level[of M] lev* **unfolding** *tr-S cdcl_W-restart-M-level-inv-def*

by *auto*

then have

$\text{get-maximum-level } M \ (\text{remove1-mset } (-L) \ D \ \# \cup \text{remove1-mset } L \ C) =$

$\text{backtrack-lvl } S$

```

    by (auto simp: get-maximum-level-union-mset get-maximum-level-plus max-def D)
  then show ?case
    using tr-S not-empty-get-maximum-level-exists-lit[of
      remove1-mset (- L) D # $\cup$  remove1-mset L C M] T
    by auto
next
case (skip L C' M D T) note tr-S = this(1) and D = this(2) and T = this(5)
then obtain La where
  La  $\in$  # D and
  get-level (Propagated L C' # M) La = backtrack-lvl S
  using skip confl-inv by auto
moreover
  have atm-of La  $\neq$  atm-of L
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    then have La: La = L using  $\langle La \in \# D \rangle \langle - L \notin \# D \rangle$ 
    by (auto simp add: atm-of-eq-atm-of)
    have Propagated L C' # M  $\models_{as}$  CNot D
    using conflicting tr-S D unfolding cdclW-restart-conflicting-def by auto
    then have -L  $\in$  lits-of-l M
    using  $\langle La \in \# D \rangle$  in-CNot-implies-uminus(2)[of L D Propagated L C' # M] unfolding La
    by auto
    then show False using lev tr-S unfolding cdclW-restart-M-level-inv-def consistent-interp-def
  by auto
  qed
  then have get-level (Propagated L C' # M) La = get-level M La by auto
  ultimately show ?case using D tr-S T by auto
next
case backtrack
then show ?case
  by (auto split: if-split-asm simp: cdclW-restart-M-level-inv-decomp lev)
qed auto

```

Strong completeness

lemma *cdcl_W-cp-propagate-conf*:

assumes *cdcl_W-cp* S T
shows *propagate*** S T \vee ($\exists S'. \text{propagate** } S S' \wedge \text{conflict } S' T$)
using *assms* **by** *induction blast+*

lemma *rtranclp-cdcl_W-cp-propagate-conf*:

assumes *cdcl_W-cp*** S T
shows *propagate*** S T \vee ($\exists S'. \text{propagate** } S S' \wedge \text{conflict } S' T$)
by (*simp add: assms rtranclp-cdcl_W-cp-propa-or-propa-conf*)

lemma *propagate-high-levelE*:

assumes *propagate* S T
obtains M' N' U k L C **where**
 state S = (M', N', U, k, None) **and**
 state T = (Propagated L (C + {#L#}) # M', N', U, k, None) **and**
 C + {#L#} \in # *local.clauses* S **and**
 M' \models_{as} CNot C **and**
 undefined-lit (trail S) L

proof –

obtain E L **where**
conf: conflicting S = None **and**


```

E:  $E \in \#$  clauses  $S$  and
LE:  $L \in \#$   $E$  and
|: trail  $S \models_{as} CNot (E - \{\#L\# \})$  and
undef: undefined-lit (trail  $S$ )  $L$  and
T:  $T \sim cons\text{-}trail (Propagated\ L\ E)\ S$ 
using assms by (elim propagateE) simp
obtain  $M\ N\ U\ k$  where
S: state  $S = (M, N, U, k, None)$ 
using conf by auto
show thesis
using that[of  $M\ N\ U\ k\ L\ remove1\text{-}mset\ L\ E$ ]  $S\ T\ LE\ E\ tr\ undef$ 
by auto
qed

lemma cdclW-cp-propagate-completeness:
assumes MN:  $set\ M \models_s set\text{-}mset\ N$  and
cons: consistent-interp (set  $M$ ) and
tot: total-over- $m$  (set  $M$ ) (set-mset  $N$ ) and
lits-of-l (trail  $S$ )  $\subseteq set\ M$  and
init-clss  $S = N$  and
propagate**  $S\ S'$  and
learned-clss  $S = \{\#\}$ 
shows  $length\ (trail\ S) \leq length\ (trail\ S') \wedge lits\text{-}of\text{-}l\ (trail\ S') \subseteq set\ M$ 
using assms(6,4,5,7)
proof (induction rule: rtranclp-induct)
case base
then show ?case by auto
next
case (step  $Y\ Z$ )
note st = this(1) and propa = this(2) and IH = this(3) and lits' = this(4) and NS = this(5) and
learned = this(6)
then have len:  $length\ (trail\ S) \leq length\ (trail\ Y)$  and LM:  $lits\text{-}of\text{-}l\ (trail\ Y) \subseteq set\ M$ 
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:  $C + \{\#L\#\} \in \#$  clauses  $Y$  and
M'-C:  $M' \models_{as} CNot\ C$  and
undefined-lit (trail  $Y$ )  $L$ 
using propa by (auto elim: propagate-high-levelE)
have init-clss  $S = init\text{-}clss\ Y$ 
using st by induction (auto elim: propagateE)
then have [simp]:  $N' = N$  using NS  $Y\ Z$  by simp
have learned-clss  $Y = \{\#\}$ 
using st learned by induction (auto elim: propagateE)
then have [simp]:  $U = \{\#\}$  using  $Y$  by auto
have  $set\ M \models_s CNot\ C$ 
using M'-C LM  $Y$  unfolding true-annots-def Ball-def true-annot-def true-clss-def true-clss-def
by force
moreover
have  $set\ M \models C + \{\#L\#\}$ 
using MN  $C$  learned  $Y\ NS$   $\langle init\text{-}clss\ S = init\text{-}clss\ Y \rangle \langle learned\text{-}clss\ Y = \{\#\} \rangle$ 
unfolding true-clss-def clauses-def by fastforce
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

assumes *propagate*** *S X*

shows

rtrancp-propagate-init-clss: *init-clss X = init-clss S* and

rtrancp-propagate-learned-clss: *learned-clss X = learned-clss S*

using *assms* by (induction rule: *rtrancp-induct*) (auto elim: *propagateE*)

lemma *completeness-is-a-full1-propagation*:

fixes *S* :: 'st and *M* :: 'v literal list

assumes *MN*: *set M* \models *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-l* (*trail S*) \subseteq *set M*

shows $\exists S'. \text{propagate}^{**} S S' \wedge \text{full cdcl}_W\text{-cp } S S'$

proof –

obtain *S'* where *full*: *full cdcl_W-cp S S'*

using *always-exists-full-cdcl_W-cp-step alien* by *blast*

then consider (*propa*) *propagate** S S'*

| (*confl*) $\exists X. \text{propagate}^{**} S X \wedge \text{conflict } X S'$

using *rtrancp-cdcl_W-cp-propagate-confl* unfolding *full-def* by *blast*

then show ?thesis

proof cases

case *propa* then show ?thesis using *full* by *blast*

next

case *confl*

then obtain *X* where

X: *propagate** S X* and

Xconf: *conflict X S'*

by *blast*

have *clsX*: *init-clss X* = *init-clss S*

using *X* by (*blast dest: rtrancp-propagate-init-clss*)

have *learnedX*: *learned-clss X* = {#}

using *X* *learned* by (*auto dest: rtrancp-propagate-learned-clss*)

obtain *E* where

E: *E* \in # *init-clss X* + *learned-clss X* and

Not-E: *trail X* \models *CNot E*

using *Xconf* by (*auto simp add: clauses-def elim!: conflictE*)

have *lits-of-l* (*trail X*) \subseteq *set M*

using *cdcl_W-cp-propagate-completeness*[*OF assms(1–3) lits - X learned*] *learned* by *auto*

then have *MNE*: *set M* \models *CNot E*

using *Not-E*

by (*fastforce simp add: true-annots-def true-annot-def true-clss-def true-clss-def*)

have $\neg \text{set } M \models \text{set-mset } N$

using *E* *consistent-CNot-not*[*OF cons MNE*]

unfolding *learnedX* *true-clss-def* unfolding *clsX* *clsS* by *auto*

then show ?thesis using *MN* by *blast*

qed

qed

See also *rtrancp-cdcl_W-cp-dropWhile-trail*

lemma *rtrancp-propagate-is-trail-append*:

$\text{propagate}^{**} S T \implies \exists c. \text{trail } T = c @ \text{trail } S$
by (induction rule: *rtrancpl-induct*) (auto elim: *propagateE*)

lemma *rtrancpl-propagate-is-update-trail*:
 $\text{propagate}^{**} S T \implies \text{cdcl}_W\text{-restart-}M\text{-level-inv } S \implies$
 $\text{init-clss } S = \text{init-clss } T \wedge \text{learned-clss } S = \text{learned-clss } T \wedge \text{backtrack-lvl } S = \text{backtrack-lvl } T$
 $\wedge \text{conflicting } S = \text{conflicting } T$

proof (induction rule: *rtrancpl-induct*)
case *base*
then show ?case **unfolding** *state-eq-def* **by** (auto simp: *cdcl_W-restart- M -level-inv-decomp*)
next
case (step $T U$) **note** $IH = \text{this}(3)[OF \text{this}(4)]$
moreover have *cdcl_W-restart- M -level-inv* U
using *rtrancpl-cdcl_W-restart-consistent-inv* $\langle \text{propagate}^{**} S T \rangle \langle \text{propagate } T U \rangle$
rtrancpl-mono[of *propagate cdcl_W-restart*] *cdcl_W-cp-consistent-inv* *propagate'*
rtrancpl-propagate-is-rtrancpl-cdcl_W-restart *step.prem*s **by** *blast*
then have *no-dup* (trail U) **unfolding** *cdcl_W-restart- M -level-inv-def* **by** *auto*
ultimately show ?case **using** $\langle \text{propagate } T U \rangle$ **unfolding** *state-eq-def*
by (fastforce simp: elim: *propagateE*)
qed

lemma *cdcl_W-stgy-strong-completeness-n*:
assumes
 $MN: \text{set } M \models_s \text{set-mset } N$ **and**
 $\text{cons: consistent-interp } (\text{set } M)$ **and**
 $\text{tot: total-over-}m (\text{set } M) (\text{set-mset } N)$ **and**
 $\text{atm-incl: atm-of ' } (\text{set } M) \subseteq \text{atms-of-mm } N$ **and**
 $\text{distM: distinct } M$ **and**
 $\text{length: } n \leq \text{length } M$
shows
 $\exists M' k S. \text{length } M' \geq n \wedge$
 $\text{lits-of-l } M' \subseteq \text{set } M \wedge$
 $\text{no-dup } M' \wedge$
 $\text{state } S = (M', N, \{\#\}, k, \text{None}) \wedge$
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$
using *length*
proof (induction n)
case 0
have $\text{state } (\text{init-state } N) = ([], N, \{\#\}, 0, \text{None})$
by (auto simp: *state-eq-def* *simp del: state-simp*)
moreover have
 $0 \leq \text{length } []$ **and**
 $\text{lits-of-l } [] \subseteq \text{set } M$ **and**
 $\text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) (\text{init-state } N)$
and *no-dup* $[]$
by (auto simp: *state-eq-def* *simp del: state-simp*)
ultimately show ?case **using** *state-eq-sym* **by** *blast*
next
case (*Suc* n) **note** $IH = \text{this}(1)$ **and** $n = \text{this}(2)$
then obtain $M' k S$ **where**
 $l\text{-}M': \text{length } M' \geq n$ **and**
 $M': \text{lits-of-l } M' \subseteq \text{set } M$ **and**
 $n\text{-d}[\text{simp}]: \text{no-dup } M'$ **and**
 $S: \text{state } S = (M', N, \{\#\}, k, \text{None})$ **and**
 $st: \text{cdcl}_W\text{-stgy}^{**} (\text{init-state } N) S$
by *auto*

```

have
  M: cdclW-restart-M-level-inv S and
  alien: no-strange-atm S
  using cdclW-restart-M-level-inv-S0-cdclW-restart rtrancpl-cdclW-stgy-consistent-inv st apply blast
using cdclW-restart-M-level-inv-S0-cdclW-restart no-strange-atm-S0 rtrancpl-cdclW-restart-no-strange-atm-inv
rtrancpl-cdclW-stgy-rtrancpl-cdclW-restart st by blast

{ assume no-step: ¬no-step propagate S
  obtain S' where S': propagate** S S' and full: full cdclW-cp S S'
  using completeness-is-a-full1-propagation[OF assms(1-3), of S] alien M' S
  by (auto simp: comp-def)
  have lev: cdclW-restart-M-level-inv S'
  using M S' rtrancpl-cdclW-restart-consistent-inv rtrancpl-propagate-is-rtrancpl-cdclW-restart by
blast
  then have n-d'[simp]: no-dup (trail S')
  unfolding cdclW-restart-M-level-inv-def by auto
  have length (trail S) ≤ length (trail S') ∧ lits-of-l (trail S') ⊆ set M
  using S' full cdclW-cp-propagate-completeness[OF assms(1-3), of S] M' S
  by (auto simp: comp-def)
  moreover
  have full: full1 cdclW-cp S S'
  using full no-step no-step-cdclW-cp-no-conflict-no-propagate(2) unfolding full1-def full-def
  rtrancpl-unfold by blast
  then have cdclW-stgy S S' by (simp add: cdclW-stgy.conflict')
  moreover
  have propa: propagate++ S S' using S' full unfolding full1-def by (metis rtrancplD trancplD)
  have trail S = M'
  using S by (auto simp: comp-def rev-map)
  with propa have length (trail S') > n
  using l-M' propa by (induction rule: trancpl.induct) (auto elim: propagateE)
  moreover
  have stS': cdclW-stgy** (init-state N) S'
  using st cdclW-stgy.conflict'[OF full] by auto
  then have init-clss S' = N
  using stS' rtrancpl-cdclW-stgy-no-more-init-clss by fastforce
  moreover
  have
    [simp]: learned-clss S' = {#} and
    [simp]: init-clss S' = init-clss S and
    [simp]: conflicting S' = None
  using trancpl-into-rtrancpl[OF ⟨propagate++ S S'⟩] S
  rtrancpl-propagate-is-update-trail[of S S'] S M unfolding state-eq-def
  by (auto simp: comp-def)
  have S-S': state S' = (trail S', N, {#}, backtrack-lvl S', None)
  using S by auto
  have cdclW-stgy** (init-state N) S'
  apply (rule rtrancpl.rtrancpl-into-rtrancpl)
  using st apply simp
  using ⟨cdclW-stgy S S'⟩ by simp
  ultimately have ?case
  apply -
  apply (rule exI[of - trail S'], rule exI[of - backtrack-lvl S'], rule exI[of - S'])
  using S-S' by (auto simp: state-eq-def simp del: state-simp)
}
moreover {
  assume no-step: no-step propagate S

```

```

have ?case
proof (cases length M' ≥ Suc n)
  case True
  then show ?thesis using l-M' M' st M alien S n-d by blast
next
case False
then have n': length M' = n using l-M' by auto
have no-conf!: no-step conflict S
proof -
  { fix D
    assume D ∈# N and M' ⊨as CNot D
    then have set M ⊨ D using MN unfolding true-clss-def by auto
    moreover have set M ⊨s CNot D
      using ⟨M' ⊨as CNot D⟩ M'
      by (metis le-iff-sup true-annots-true-cls true-clss-union-increase)
    ultimately have False using cons consistent-CNot-not by blast
  }
  then show ?thesis
    using S by (auto simp: true-clss-def comp-def rev-map
      clauses-def elim!: conflictE)
qed
have lenM: length M = card (set M) using distM by (induction M) auto
have no-dup M' using S M unfolding cdclW-restart-M-level-inv-def by auto
then have card (lits-of-l M') = length M'
  by (induction M') (auto simp add: lits-of-def card-insert-if)
then have lits-of-l M' ⊆ set M
  using n M' n' lenM by auto
then obtain L where L: L ∈ set M and undef-m: L ∉ lits-of-l M' by auto
moreover have undef: undefined-lit M' L
  using M' Decided-Propagated-in-iff-in-lits-of-l calculation(1,2) cons
  consistent-interp-def by (metis (no-types, lifting) subset-eq)
moreover have atm-of L ∈ atms-of-mm (init-clss S)
  using atm-incl calculation S by auto
ultimately
  have dec: decide S (cons-trail (Decided L) (incr-lvl S))
    using decide-rule[of S - cons-trail (Decided L) (incr-lvl S)] S
    by auto
let ?S' = cons-trail (Decided L) (incr-lvl S)
have lits-of-l (trail ?S') ⊆ set M using L M' S undef by auto
moreover have no-strange-atm ?S'
  using alien dec M by (meson cdclW-restart-no-strange-atm-inv decide other)
ultimately obtain S'' where S'': propagate** ?S' S'' and full: full cdclW-cp ?S' S''
  using completeness-is-a-full1-propagation[OF assms(1-3), of ?S'] S undef
  by auto
have cdclW-restart-M-level-inv ?S'
  using M dec rtranclp-mono[of decide cdclW-restart] by (meson cdclW-restart-consistent-inv
decide other)
then have lev'': cdclW-restart-M-level-inv S''
  using S'' rtranclp-cdclW-restart-consistent-inv rtranclp-propagate-is-rtranclp-cdclW-restart by
blast
then have n-d'': no-dup (trail S'')
  unfolding cdclW-restart-M-level-inv-def by auto
have length (trail ?S') ≤ length (trail S'') ∧ lits-of-l (trail S'') ⊆ set M
  using S'' full cdclW-cp-propagate-completeness[OF assms(1-3), of ?S' S''] L M' S undef
  by simp
then have Suc n ≤ length (trail S'') ∧ lits-of-l (trail S'') ⊆ set M

```

```

    using l-M' S undef by auto
  moreover
    have cdclW-restart-M-level-inv (cons-trail (Decided L)
      (update-backtrack-lvl (Suc (backtrack-lvl S)) S))
      using S (cdclW-restart-M-level-inv (cons-trail (Decided L) (incr-lvl S))) by auto
    then have S'':
      state S'' = (trail S'', N, {#}, backtrack-lvl S'', None)
      using rtrancplp-propagate-is-update-trail[OF S''] S undef n-d'' lev''
      by auto
    then have cdclW-stgy** (init-state N) S''
      using cdclW-stgy.intros(2)[OF decide[OF dec] - full] no-step no-conflict st
      by (auto simp: cdclW-cp.simps)
    ultimately show ?thesis using S'' n-d'' by blast
  qed
}
ultimately show ?case by blast
qed

```

theorem 2.9.11 page 84 of Weidenbach's book (with strategy)

lemma *cdcl_W-stgy-strong-completeness*:

assumes

MN: $\text{set } M \models_s \text{set-mset } N$ **and**
cons: *consistent-interp* (set *M*) **and**
tot: *total-over-m* (set *M*) (set-mset *N*) **and**
atm-incl: *atm-of* ' (set *M*) \subseteq *atms-of-mm* *N* **and**
distM: *distinct* *M*

shows

$\exists M' k S.$
lits-of-l *M'* = set *M* \wedge
 state *S* = (*M'*, *N*, {#}, *k*, None) \wedge
 cdcl_W-stgy** (init-state *N*) *S* \wedge
 final-cdcl_W-restart-state *S*

proof –

from *cdcl_W-stgy-strong-completeness-n*[OF *assms*, of length *M*]

obtain *M' k T* **where**

l: length *M* \leq length *M'* **and**
M'-M: *lits-of-l* *M'* \subseteq set *M* **and**
no-dup: *no-dup* *M'* **and**
T: state *T* = (*M'*, *N*, {#}, *k*, None) **and**
st: cdcl_W-stgy** (init-state *N*) *T*
by *auto*

have card (set *M*) = length *M* **using** *distM* **by** (simp add: *distinct-card*)

moreover

have cdcl_W-restart-M-level-inv *T*
using rtrancplp-cdcl_W-stgy-consistent-inv[OF *st*] *T* **by** *auto*
then have card (set ((map ($\lambda l.$ *atm-of* (*lit-of* *l*)) *M'*))) = length *M'*
using *distinct-card no-dup* **by** *fastforce*

moreover have card (*lits-of-l* *M'*) = card (set ((map ($\lambda l.$ *atm-of* (*lit-of* *l*)) *M'*)))

using *no-dup unfolding lits-of-def apply* (*induction* *M'*) **by** (auto simp add: *card-insert-if*)

ultimately have card (set *M*) \leq card (*lits-of-l* *M'*) **using** *l* **unfolding** *lits-of-def* **by** *auto*

then have set *M* = *lits-of-l* *M'*

using *M'-M card-seteq* **by** *blast*

moreover

then have *M'* \models_{asm} *N*

using *MN unfolding true-annots-def Ball-def true-annot-def true-clss-def* **by** *auto*

then have final-cdcl_W-restart-state *T*

using T *no-dup* **unfolding** *final-cdcl_W-restart-state-def* **by** *auto*
 ultimately show *?thesis* using *st* T **by** *blast*
 qed

No conflict with only variables of level less than backtrack level

This invariant is stronger than the previous argument in the sense that it is a property about all possible conflicts.

definition *no-smaller-conf* ($S :: 'st$) \equiv
 $(\forall M K M' D. M' @ Decided K \# M = trail S \longrightarrow D \in \# clauses S$
 $\longrightarrow \neg M \models_{as} CNot D)$

lemma *no-smaller-conf-init-sate*[*simp*]:
no-smaller-conf (*init-state* N) **unfolding** *no-smaller-conf-def* **by** *auto*

lemma *cdcl_W-o-no-smaller-conf-inv*:

fixes $S S' :: 'st$

assumes

cdcl_W-o $S S'$ **and**

lev: *cdcl_W-restart-M-level-inv* S **and**

max-lev: *conflict-is-false-with-level* S **and**

smaller: *no-smaller-conf* S **and**

no-f: *no-clause-is-false* S

shows *no-smaller-conf* S'

using *assms*(1,2) **unfolding** *no-smaller-conf-def*

proof (*induct rule*: *cdcl_W-o-induct*)

case (*decide* $L T$) **note** *conf* = *this*(1) **and** *undef* = *this*(2) **and** $T = this(4)$

have [*simp*]: *clauses* $T = clauses S$

using T *undef* **by** *auto*

show *?case*

proof (*intro allI impI*)

fix $M'' K M' Da$

assume $M'' @ Decided K \# M' = trail T$

and D : $Da \in \# local.clauses T$

then have $tl M'' @ Decided K \# M' = trail S$

$\vee (M'' = [] \wedge Decided K \# M' = Decided L \# trail S)$

using T *undef* **by** (*cases* M'') *auto*

moreover {

assume $tl M'' @ Decided K \# M' = trail S$

then have $\neg M' \models_{as} CNot Da$

using $D T$ *undef* *no-f* *conf* *smaller* **unfolding** *no-smaller-conf-def* *smaller* **by** *fastforce*

}

moreover {

assume $Decided K \# M' = Decided L \# trail S$

then have $\neg M' \models_{as} CNot Da$ **using** *no-f* D *conf* T **by** *auto*

}

ultimately show $\neg M' \models_{as} CNot Da$ **by** *fast*

qed

next

case *resolve*

then show *?case* **using** *smaller* *no-f* *max-lev* **unfolding** *no-smaller-conf-def* **by** *auto*

next

case *skip*

then show *?case* **using** *smaller* *no-f* *max-lev* **unfolding** *no-smaller-conf-def* **by** *auto*

next

```

case (backtrack L D K i M1 M2 T) note confl = this(1) and LD = this(2) and decomp = this(3)
and
  T = this(8)
obtain c where M: trail S = c @ M2 @ Decided K # M1
using decomp by auto

show ?case
proof (intro allI impI)
  fix M ia K' M' Da
  assume M' @ Decided K' # M = trail T
  then have tl M' @ Decided K' # M = M1
    using T decomp lev by (cases M') (auto simp: cdclW-restart-M-level-inv-decomp)
  let ?S' = (cons-trail (Propagated L D)
    (reduce-trail-to M1 (add-learned-cl D
      (update-backtrack-lvl i (update-conflicting None S))))))
  assume D: Da ∈ # clauses T
  moreover{
    assume Da ∈ # clauses S
    then have ¬M ⊨as CNot Da using ⟨tl M' @ Decided K' # M = M1⟩ M confl smaller
      unfolding no-smaller-confl-def by auto
  }
  moreover {
    assume Da: Da = D
    have ¬M ⊨as CNot Da
    proof (rule ccontr)
      assume ¬ ?thesis
      then have −L ∈ lits-of-l M
        using LD unfolding Da by (simp add: in-CNot-implies-uminus(2))
      then have −L ∈ lits-of-l (Propagated L D # M1)
        using UnI2 ⟨tl M' @ Decided K' # M = M1⟩
        by auto
      moreover
        have backtrack S ?S'
          using backtrack-rule[of S] backtrack.hyps
          by (force simp: state-eq-def simp del: state-simp)
        then have cdclW-restart-M-level-inv ?S'
          using cdclW-restart-consistent-inv[OF - lev] other[OF bj] by (auto intro: cdclW-bj.intros)
        then have no-dup (Propagated L D # M1)
          using decomp lev unfolding cdclW-restart-M-level-inv-def by auto
        ultimately show False
          using Decided-Propagated-in-iff-in-lits-of-l defined-lit-map by auto
      qed
    }
  ultimately show ¬M ⊨as CNot Da
    using T decomp lev unfolding cdclW-restart-M-level-inv-def by fastforce
  qed
qed

```

```

lemma conflict-no-smaller-confl-inv:
  assumes conflict S S'
  and no-smaller-confl S
  shows no-smaller-confl S'
  using assms unfolding no-smaller-confl-def by (fastforce elim: conflictE)

```

```

lemma propagate-no-smaller-confl-inv:
  assumes propagate: propagate S S'

```


and $n\text{-l}$: $\text{no-smaller-confli } S$
shows $\text{no-smaller-confli } S'$
unfolding $\text{no-smaller-confli-def}$
proof (intro allI impI)
fix $M' K M'' D$
assume $M': M'' @ \text{Decided } K \# M' = \text{trail } S'$
and $D \in \# \text{ clauses } S'$
obtain $M N U k C L$ **where**
 S : $\text{state } S = (M, N, U, k, \text{None})$ **and**
 S' : $\text{state } S' = (\text{Propagated } L (C + \{\#L\}) \# M, N, U, k, \text{None})$ **and**
 $C + \{\#L\} \in \# \text{ clauses } S$ **and**
 $M \models_{\text{as}} C \text{Not } C$ **and**
 $\text{undefined-lit } M L$
using $\text{propagate by (auto elim: propagate-high-levelE)}$
have $\text{tl } M'' @ \text{Decided } K \# M' = \text{trail } S$ **using** $M' S S'$
by ($\text{metis Pair-inject list.inject list.sel(3) ann-lit.distinct(1) self-append-conv2}$
 tl-append2)
then have $\neg M' \models_{\text{as}} C \text{Not } D$
using $\langle D \in \# \text{ clauses } S' \rangle n\text{-l } S S' \text{ clauses-def}$ **unfolding** $\text{no-smaller-confli-def}$ **by** auto
then show $\neg M' \models_{\text{as}} C \text{Not } D$ **by** auto
qed

lemma $\text{cdcl}_W\text{-cp-no-smaller-confli-inv}$:
assumes $\text{propagate: cdcl}_W\text{-cp } S S'$
and $n\text{-l}$: $\text{no-smaller-confli } S$
shows $\text{no-smaller-confli } S'$
using assms
proof ($\text{induct rule: cdcl}_W\text{-cp.induct}$)
case ($\text{conflict}' S S'$)
then show $?case$ **using** $\text{conflict-no-smaller-confli-inv[of } S S']$ **by** blast
next
case ($\text{propagate}' S S'$)
then show $?case$ **using** $\text{propagate-no-smaller-confli-inv[of } S S']$ **by** fastforce
qed

lemma $\text{rtrancp-cdcl}_W\text{-cp-no-smaller-confli-inv}$:
assumes $\text{propagate: cdcl}_W\text{-cp}^{**} S S'$
and $n\text{-l}$: $\text{no-smaller-confli } S$
shows $\text{no-smaller-confli } S'$
using assms
proof ($\text{induct rule: rtrancp-induct}$)
case base
then show $?case$ **by** simp
next
case ($\text{step } S' S''$)
then show $?case$ **using** $\text{cdcl}_W\text{-cp-no-smaller-confli-inv[of } S' S'']$ **by** fast
qed

lemma $\text{trancp-cdcl}_W\text{-cp-no-smaller-confli-inv}$:
assumes $\text{propagate: cdcl}_W\text{-cp}^{++} S S'$
and $n\text{-l}$: $\text{no-smaller-confli } S$
shows $\text{no-smaller-confli } S'$
using assms
proof ($\text{induct rule: trancp.induct}$)
case ($r\text{-into-tranci } S S'$)
then show $?case$ **using** $\text{cdcl}_W\text{-cp-no-smaller-confli-inv[of } S S']$ **by** blast

```

next
  case (tranc1-into-tranc1 S S' S'')
  then show ?case using cdclW-cp-no-smaller-conf1-inv[of S' S''] by fast
qed

lemma full-cdclW-cp-no-smaller-conf1-inv:
  assumes full cdclW-cp S S'
  and n-l: no-smaller-conf1 S
  shows no-smaller-conf1 S'
  using assms unfolding full-def
  using rtrancp-cdclW-cp-no-smaller-conf1-inv[of S S'] by blast

lemma full1-cdclW-cp-no-smaller-conf1-inv:
  assumes full1 cdclW-cp S S'
  and n-l: no-smaller-conf1 S
  shows no-smaller-conf1 S'
  using assms unfolding full1-def
  using trancp-cdclW-cp-no-smaller-conf1-inv[of S S'] by blast

lemma cdclW-stgy-no-smaller-conf1-inv:
  assumes cdclW-stgy S S'
  and n-l: no-smaller-conf1 S
  and conflict-is-false-with-level S
  and cdclW-restart-M-level-inv S
  shows no-smaller-conf1 S'
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict' S')
  then show ?case using full1-cdclW-cp-no-smaller-conf1-inv[of S S'] by blast
next
  case (other' S' S'')
  have no-smaller-conf1 S'
    using cdclW-o-no-smaller-conf1-inv[OF other'.hyps(1) other'.prems(3,2,1)]
    not-conflict-not-any-negated-init-clss other'.hyps(2) cdclW-cp.simps by auto
  then show ?case using full-cdclW-cp-no-smaller-conf1-inv[of S' S''] other'.hyps by blast
qed

lemma is-conflicting-exists-conflict:
  assumes ¬(∀ D ∈ #init-clss S' + learned-clss S'. ¬ trail S' ⊨as CNot D)
  and conflicting S' = None
  shows ∃ S''. conflict S' S''
  using assms clauses-def not-conflict-not-any-negated-init-clss by fastforce

lemma cdclW-o-conflict-is-no-clause-is-false:
  fixes S S' :: 'st
  assumes
    cdclW-o S S' and
    lev: cdclW-restart-M-level-inv S and
    max-lev: conflict-is-false-with-level S and
    no-f: no-clause-is-false S and
    no-l: no-smaller-conf1 S
  shows no-clause-is-false S'
    ∨ (conflicting S' = None
      → (∀ D ∈ # clauses S'. trail S' ⊨as CNot D
        → (∃ L. L ∈ # D ∧ get-level (trail S') L = backtrack-lvl S')))
  using assms(1,2)

```

```

proof (induct rule: cdclW-o-induct)
  case (decide L T) note  $S = \text{this}(1)$  and  $\text{undef} = \text{this}(2)$  and  $T = \text{this}(4)$ 
  show ?case
  proof (rule HOL.disjI2, clarify)
    fix D
    assume D:  $D \in \# \text{ clauses } T$  and M-D:  $\text{trail } T \models_{\text{as}} \text{CNot } D$ 
    let ?M = trail S
    let ?M' = trail T
    let ?k = backtrack-lvl S
    have  $\neg ?M \models_{\text{as}} \text{CNot } D$ 
      using no-f D S T 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 L'': lits-of-l (Decided L # ?M)  $\models_l L''$ 
          using M-D x T undef unfolding true-annots-def Ball-def true-annot-def CNot-def
            true-cls-def Bex-def by auto
        show  $\exists L \in \# x. \text{lits-of-l } ?M \models_l L$  unfolding Bex-def
          using L'(1) L'(2)  $\langle -L \notin \# D \rangle \langle L'' \in \# x \rangle$ 
             $\langle \text{lits-of-l } (\text{Decided } L \# \text{trail } S) \models_l L'' \rangle$  by auto
        qed
        then show False using  $\langle \neg ?M \models_{\text{as}} \text{CNot } D \rangle$  by auto
      qed
    have atm-of L  $\notin \text{atm-of } \langle \text{lits-of-l } ?M \rangle$ 
      using undef defined-lit-map unfolding lits-of-def by fastforce
    then have get-level (Decided L # ?M)  $(-L) = ?k + 1$ 
      using lev unfolding cdclW-restart-M-level-inv-def by auto
    then have  $-L \in \# D \wedge \text{get-level } ?M' (-L) = \text{backtrack-lvl } T$ 
      using  $\langle -L \in \# D \rangle$  T undef by auto
    then show  $\exists La. La \in \# D \wedge \text{get-level } ?M' La = \text{backtrack-lvl } T$ 
      by blast
    qed
  next
    case resolve
    then show ?case by auto
  next
    case skip
    then show ?case by auto
  next
    case (backtrack L D K i M1 M2 T) note decomp = this(3) and lev-K = this(7) and  $T = \text{this}(8)$ 
    show ?case
    proof (rule HOL.disjI2, clarify)
      fix Da
      assume Da:  $Da \in \# \text{ clauses } T$  and M-D:  $\text{trail } T \models_{\text{as}} \text{CNot } Da$ 
      obtain c where M:  $\text{trail } S = c @ M2 @ \text{Decided } K \# M1$ 
        using decomp by auto
      have tr-T:  $\text{trail } T = \text{Propagated } L D \# M1$ 
        using T decomp lev by (auto simp: cdclW-restart-M-level-inv-decomp)
      have backtrack S T

```

```

    using backtrack-rule[of S] backtrack.hyps T
    by (force simp del: state-simp simp: state-eq-def)
  then have lev': cdclW-restart-M-level-inv T
    using cdclW-restart-consistent-inv lev other cdclW-bj.backtrack cdclW-o.bj by blast
  then have - L ∉ lits-of-l M1
    using lev cdclW-restart-M-level-inv-def tr-T unfolding consistent-interp-def by (metis insert-iff
      list.simps(15) lits-of-insert ann-lit.sel(2))
  { assume Da ∈# clauses S
    then have ¬M1 ⊨as CNot Da using no-l M unfolding no-smaller-conflict-def by auto
  }
  moreover {
    assume Da: Da = D
    have ¬M1 ⊨as CNot Da using ⟨- L ∉ lits-of-l M1⟩ unfolding Da
      using backtrack.hyps(2) in-CNot-implies-uminus(2) by auto
  }
  ultimately have ¬M1 ⊨as CNot Da
    using Da T decomp lev by (fastforce simp: cdclW-restart-M-level-inv-decomp)
  then have -L ∈# Da
    using M-D ⟨- L ∉ lits-of-l M1⟩ T unfolding tr-T true-annots-true-clss true-clss-def
    by (auto simp: uminus-lit-swap)
  have no-dup (Propagated L D # M1)
    using lev lev' T decomp unfolding cdclW-restart-M-level-inv-def by auto
  then have L: atm-of L ∉ atm-of ' lits-of-l M1 unfolding lits-of-def by auto
  have get-level (Propagated L D # M1) (-L) = i
    using lev-K lev unfolding cdclW-restart-M-level-inv-def
    by (simp add: M image-Un atm-lit-of-set-lits-of-l)

  then have -L ∈# Da ∧ get-level (trail T) (-L) = backtrack-lvl T
    using ⟨-L ∈# Da⟩ T decomp lev by (auto simp: cdclW-restart-M-level-inv-def)
  then show ∃ La. La ∈# Da ∧ get-level (trail T) La = backtrack-lvl T
    by blast
qed
qed

```

lemma full1-cdcl_W-cp-exists-conflict-decompose:

```

assumes
  confl: ∃ D ∈# clauses S. trail S ⊨as CNot D and
  full: full cdclW-cp S U and
  no-conf: conflicting S = None and
  lev: cdclW-restart-M-level-inv S
shows ∃ T. propagate** S T ∧ conflict T U
proof -
  consider (propa) propagate** S U
  | (confl) T where propagate** S T and conflict T U
  using full unfolding full-def by (blast dest: rtranclp-cdclW-cp-propa-or-propa-conf)
  then show ?thesis
  proof cases
    case confl
    then show ?thesis by blast
  next
    case propa
    then have conflicting U = None and
      [simp]: learned-clss U = learned-clss S and
      [simp]: init-clss U = init-clss S
      using no-conf rtranclp-propagate-is-update-trail lev by auto
    moreover

```

```

obtain  $D$  where  $D: D \in \# \text{clauses } U$  and
   $trS: \text{trail } S \models_{as} CNot\ D$ 
  using  $\text{confl clauses-def}$  by  $\text{auto}$ 
obtain  $M$  where  $M: \text{trail } U = M @ \text{trail } S$ 
  using  $\text{full rtrancpl-cdcl}_W\text{-cp-dropWhile-trail unfolding full-def}$  by  $\text{meson}$ 
have  $tr-U: \text{trail } U \models_{as} CNot\ D$ 
  apply  $(\text{rule true-annots-mono})$ 
  using  $trS$  unfolding  $M$  by  $\text{simp-all}$ 
have  $\exists V. \text{conflict } U\ V$ 
  using  $(\text{conflicting } U = None) \ D \text{ clauses-def not-conflict-not-any-negated-init-clss } tr-U$ 
  by  $\text{meson}$ 
then have  $False$  using  $\text{full cdcl}_W\text{-cp.conflict' unfolding full-def}$  by  $\text{blast}$ 
then show  $?thesis$  by  $\text{fast}$ 
qed
qed

```

lemma $\text{full1-cdcl}_W\text{-cp-exists-conflict-full1-decompose}$:

```

assumes
   $\text{confl}: \exists D \in \# \text{clauses } S. \text{trail } S \models_{as} CNot\ D$  and
   $\text{full}: \text{full cdcl}_W\text{-cp } S\ U$  and
   $\text{no-confl}: \text{conflicting } S = None$  and
   $\text{lev}: \text{cdcl}_W\text{-restart-M-level-inv } S$ 
shows  $\exists T\ D. \text{propagate}^{**} S\ T \wedge \text{conflict } T\ U$ 
   $\wedge \text{trail } T \models_{as} CNot\ D \wedge \text{conflicting } U = Some\ D \wedge D \in \# \text{clauses } S$ 

```

proof –

```

obtain  $T$  where  $\text{propa}: \text{propagate}^{**} S\ T$  and  $\text{conf}: \text{conflict } T\ U$ 
  using  $\text{full1-cdcl}_W\text{-cp-exists-conflict-decompose}[OF\ \text{assms}]$  by  $\text{blast}$ 
have  $p: \text{learned-clss } T = \text{learned-clss } S\ \text{init-clss } T = \text{init-clss } S$ 
  using  $\text{propa lev rtrancpl-propagate-is-update-trail}$  by  $\text{auto}$ 
have  $c: \text{learned-clss } U = \text{learned-clss } T\ \text{init-clss } U = \text{init-clss } T$ 
  using  $\text{conf}$  by  $(\text{auto elim: conflictE})$ 
obtain  $D$  where  $\text{trail } T \models_{as} CNot\ D \wedge \text{conflicting } U = Some\ D \wedge D \in \# \text{clauses } S$ 
  using  $\text{conf } p\ c$  by  $(\text{fastforce simp: clauses-def elim!: conflictE})$ 
then show  $?thesis$ 
  using  $\text{propa conf}$  by  $\text{blast}$ 
qed

```

lemma $\text{cdcl}_W\text{-stgy-no-smaller-conf}$:

```

assumes
   $\text{cdcl}_W\text{-stgy } S\ S'$  and
   $n-l: \text{no-smaller-conf } S$  and
   $\text{conflict-is-false-with-level } S$  and
   $\text{cdcl}_W\text{-restart-M-level-inv } S$  and
   $\text{no-clause-is-false } S$  and
   $\text{distinct-cdcl}_W\text{-restart-state } S$  and
   $\text{cdcl}_W\text{-restart-conflicting } S$ 
shows  $\text{no-smaller-conf } S'$ 
using  $\text{assms}$ 
proof  $(\text{induct rule: cdcl}_W\text{-stgy.induct})$ 
  case  $(\text{conflict' } S')$ 
  show  $\text{no-smaller-conf } S'$ 
    using  $\text{conflict'.hyps conflict'.prems(1) full1-cdcl}_W\text{-cp-no-smaller-conf-inv}$  by  $\text{blast}$ 
next
  case  $(\text{other' } S'\ S'')$ 
  have  $\text{lev': cdcl}_W\text{-restart-M-level-inv } S'$ 
    using  $\text{cdcl}_W\text{-restart-consistent-inv other other'.hyps(1) other'.prems(3)}$  by  $\text{blast}$ 

```

```

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-restart-M-level-inv  $S$  and
    no-clause-is-false  $S$  and
    distinct-cdclW-restart-state  $S$  and
    cdclW-restart-conflicting  $S$ 
  shows conflict-is-false-with-level  $S'$ 
  using assms
proof (induct rule: cdclW-stgy.induct)
  case (conflict'  $S'$ )
  have no-smaller-confl  $S'$ 
    using conflict'.hyps conflict'.prems(1) full1-cdclW-cp-no-smaller-confl-inv by blast
  moreover have conflict-is-false-with-level  $S'$ 
    using conflict'.hyps conflict'.prems(2-4)
    rtrancpl-cdclW-restart-co-conflict-ex-lit-of-max-level[of  $S$   $S'$ ]
    unfolding full-def full1-def rtrancpl-unfold by presburger
  then show ?case by blast
next
  case (other'  $S'$   $S''$ )
  have lev': cdclW-restart-M-level-inv  $S'$ 
    using cdclW-restart-consistent-inv other other'.hyps(1) other'.prems(3) by blast
  moreover
    have no-clause-is-false  $S'$ 
       $\vee$  (conflicting  $S' = \text{None}$   $\longrightarrow$  ( $\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{\text{as}} C\text{Not } D$ 
         $\longrightarrow$  ( $\exists L. L \in \# D \wedge \text{get-level}(\text{trail } S') L = \text{backtrack-lvl } S')$ ))
      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  $S' = \text{None}$ 
        then have conflict-is-false-with-level  $S'$  by auto
        moreover have full cdclW-cp  $S'$   $S''$ 
          by (metis (no-types) other'.hyps(3))
        ultimately have conflict-is-false-with-level  $S''$ 
          using rtrancpl-cdclW-restart-co-conflict-ex-lit-of-max-level[of  $S'$   $S''$ ] lev' (no-clause-is-false  $S'$ )
          by blast
      }
    }
  moreover
  {
    assume c: conflicting  $S' \neq \text{None}$ 
    have conflicting  $S \neq \text{None}$  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: rtrancpl-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) lev' by (metis (mono-tags, lifting) conflictE state-eq-trail
      trail-update-conflicting)
    obtain M where M: trail S'' = M @ trail S' and nm:  $\forall m \in \text{set } M. \neg \text{is-decided } m$ 
    using rtrancpl-cdclW-cp-dropWhile-trail other'(3) unfolding full-def by meson
    have btS: backtrack-lvl S'' = backtrack-lvl S'
    using other'.hypos(3) unfolding full-def by (metis rtrancpl-cdclW-cp-backtrack-lvl)
    have inv: cdclW-restart-M-level-inv S''
    by (metis (no-types) cdclW-stgy.conflict' cdclW-stgy-consistent-inv full-unfold lev'
      other'.hypos(3))
    then have nd: no-dup (trail S'')
    by (metis (no-types) cdclW-restart-M-level-inv-decomp(2))
    have conflict-is-false-with-level S''
    proof cases
      assume trail S'  $\models_{as} CNot D$ 
      moreover then obtain L where
        L  $\in \# D$  and
        lev-L: get-level (trail S') L = backtrack-lvl S'
        using D-L D by blast
      moreover
        have LS':  $-L \in \text{lits-of-l}(\text{trail } S')$ 
        using  $\langle \text{trail } S' \models_{as} CNot D \rangle \langle L \in \# D \rangle \text{in-}CNot\text{-implies-uminus}(2)$  by blast
      { fix x :: ('v, 'v clause) ann-lit and
        xb :: ('v, 'v clause) ann-lit
        assume a1:  $x \in \text{set}(\text{trail } S')$  and
          a2:  $xb \in \text{set } M$  and
          a3:  $(\lambda l. \text{atm-of}(\text{lit-of } l)) \text{' set } M \cap (\lambda l. \text{atm-of}(\text{lit-of } l)) \text{' set}(\text{trail } S') = \{\}$  and
          a4:  $-L = \text{lit-of } x$  and
          a5:  $\text{atm-of } L = \text{atm-of}(\text{lit-of } xb)$ 
        moreover have atm-of (lit-of x) = atm-of L
        using a4 by (metis (no-types) atm-of-uminus)
        ultimately have False
        using a5 a3 a2 a1 by auto
      }
    }
  }
}

```

```

    }
    then have atm-of L  $\notin$  atm-of ' lits-of-l M
      using nd LS' unfolding M by (auto simp add: lits-of-def)
    then have get-level (trail S'') L = get-level (trail S') L
      unfolding M by (simp add: lits-of-def)
    ultimately show ?thesis using btS <conflicting S'' = Some D> by auto
next
assume  $\neg$ trail S'  $\models_{as}$  CNot D
then obtain L where L  $\in \#$  D and LM:  $\neg$ L  $\in$  lits-of-l M
  using <trail S''  $\models_{as}$  CNot D> unfolding M
  by (auto simp add: true-cls-def M true-annots-def true-annot-def
    split: if-split-asm)
{ fix x :: ('v, 'v clause) ann-lit and
  xb :: ('v, 'v clause) ann-lit
  assume a1: xb  $\in$  set (trail S') and
    a2: x  $\in$  set M and
    a3: atm-of L = atm-of (lit-of xb) and
    a4:  $\neg$  L = lit-of x and
    a5: ( $\lambda$ l. atm-of (lit-of l)) ' set M  $\cap$  ( $\lambda$ l. atm-of (lit-of l)) ' set (trail S')
      = {}
  moreover have atm-of (lit-of xb) = atm-of ( $\neg$  L)
    using a3 by simp
  ultimately have False
    by auto }
then have LS': atm-of L  $\notin$  atm-of ' lits-of-l (trail S')
  using nd <L  $\in \#$  D> LM unfolding M by (auto simp add: lits-of-def)
show ?thesis
proof -
  have atm-of L  $\in$  atm-of ' lits-of-l M
    using < $\neg$ L  $\in$  lits-of-l M>
    by (simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set lits-of-def)
  then have get-level (M @ trail S') L = backtrack-lvl S'
    using lev' LS' nm unfolding cdclW-restart-M-level-inv-def by auto
  then show ?thesis
    using nm <L  $\in \#$  D> <conflicting S'' = Some D>
    unfolding lits-of-def btS M
    by auto
qed
qed
}
ultimately have conflict-is-false-with-level S'' by blast
}
moreover
{
  assume conflicting S'  $\neq$  None
  have no-clause-is-false S' using <conflicting S'  $\neq$  None> by auto
  then have conflict-is-false-with-level S'' using calculation(3) by presburger
}
ultimately show ?case by blast
qed

```

lemma rtrancp-cdcl_W-stgy-no-smaller-confl-inv:

```

assumes
  cdclW-stgy** S S' and
  n-l: no-smaller-confl S and
  cls-false: conflict-is-false-with-level S and

```


lev: *cdcl_W-restart-M-level-inv S* and
no-f: *no-clause-is-false S* and
dist: *distinct-cdcl_W-restart-state S* and
conflicting: *cdcl_W-restart-conflicting S* and
decomp: *all-decomposition-implies-m (init-clss S) (get-all-ann-decomposition (trail S))* and
learned: *cdcl_W-restart-learned-clause S* and
alien: *no-strange-atm S*
shows *no-smaller-confl S' \wedge conflict-is-false-with-level S'*
using *assms(1)*
proof (*induct rule: rtranclp-induct*)
case *base*
then show *?case using n-l cls-false by auto*
next
case (*step S' S''*) **note** *st = this(1)* **and** *cdcl = this(2)* **and** *IH = this(3)*
have *no-smaller-confl S' and conflict-is-false-with-level S'*
using *IH by blast+*
moreover have *cdcl_W-restart-M-level-inv S'*
using *st lev rtranclp-cdcl_W-stgy-rtranclp-cdcl_W-restart*
by (*blast intro: rtranclp-cdcl_W-restart-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-restart-state S'*
using *rtranclp-distinct-cdcl_W-restart-state-inv[of S S'] lev rtranclp-cdcl_W-stgy-rtranclp-cdcl_W-restart[OF*
st]
dist by auto
moreover have *cdcl_W-restart-conflicting S'*
using *rtranclp-cdcl_W-restart-all-inv(6)[of S S'] st alien conflicting decomp dist learned lev*
rtranclp-cdcl_W-stgy-rtranclp-cdcl_W-restart 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

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' = Some $\{\#\}$ \wedge unsatisfiable (set-mset (init-clss S'))*)
 \vee (*conflicting S' = None \wedge trail S' \models_{asm} init-clss S'*)
proof –
let *?S = init-state N*
have
termi: $\forall S''. \neg cdcl_W-stgy S' S''$ and
*step: cdcl_W-stgy** ?S S' using full unfolding full-def by auto*
moreover have
learned: cdcl_W-restart-learned-clause S' and
level-inv: cdcl_W-restart-M-level-inv S' and
alien: no-strange-atm S' and
no-dup: distinct-cdcl_W-restart-state S' and
confl: cdcl_W-restart-conflicting S' and
decomp: all-decomposition-implies-m (init-clss S') (get-all-ann-decomposition (trail S'))
using *no-d tranclp-cdcl_W-stgy-tranclp-cdcl_W-restart[of ?S S'] step rtranclp-cdcl_W-restart-all-inv(1-6)[of*
?S S']
unfolding *rtranclp-unfold by auto*

```

moreover
  have  $\forall D \in \#N. \neg [] \models_{as} CNot\ D$  using no-empty by auto
  then have confl-k: conflict-is-false-with-level S'
    using rtrancp-cdclW-stgy-no-smaller-confl-inv[OF step] no-d by auto
show ?thesis
  using cdclW-stgy-final-state-conclusive[OF termi decomp learned level-inv alien no-dup confl
    confl-k] .
qed

```

```

lemma conflict-is-full1-cdclW-cp:
  assumes cp: conflict S S'
  shows full1 cdclW-cp S S'
proof -
  have cdclW-cp S S' and conflicting S' ≠ None
    using cp cdclW-cp.intros by (auto elim!: conflictE simp: state-eq-def simp del: state-simp)
  then have cdclW-cp++ S S' by blast
  moreover have no-step cdclW-cp S'
    using  $\langle \text{conflicting } S' \neq \text{None} \rangle$  by (metis cdclW-cp-conflicting-not-empty
      option.exhaust)
  ultimately show full1 cdclW-cp S S' unfolding full1-def by blast+
qed

```

```

lemma cdclW-cp-fst-empty-conflicting-false:
  assumes
    cdclW-cp S S' and
    trail S = [] and
    conflicting S ≠ None
  shows False
  using assms by (induct rule: cdclW-cp.induct) (auto elim: propagateE conflictE)

```

```

lemma cdclW-o-fst-empty-conflicting-false:
  assumes cdclW-o S S'
  and trail S = []
  and conflicting S ≠ None
  shows False
  using assms by (induct rule: cdclW-o.induct) auto

```

```

lemma cdclW-stgy-fst-empty-conflicting-false:
  assumes cdclW-stgy S S'
  and trail S = []
  and conflicting S ≠ None
  shows False
  using assms apply (induct rule: cdclW-stgy.induct)
  using trancpD cdclW-cp-fst-empty-conflicting-false unfolding full1-def apply metis
  using cdclW-o-fst-empty-conflicting-false by blast
thm cdclW-cp.induct[split-format(complete)]

```

```

lemma cdclW-cp-conflicting-is-false:
  cdclW-cp S S'  $\implies$  conflicting S = Some {#}  $\implies$  False
  by (induction rule: cdclW-cp.induct) (auto elim: propagateE conflictE)

```

```

lemma rtrancp-cdclW-cp-conflicting-is-false:
  cdclW-cp++ S S'  $\implies$  conflicting S = Some {#}  $\implies$  False
  apply (induction rule: trancp.induct)
  by (auto dest: cdclW-cp-conflicting-is-false)

```

```

lemma cdclW-o-conflicting-is-false:
  cdclW-o S S'  $\implies$  conflicting S = Some {#}  $\implies$  False
  by (induction rule: cdclW-o-induct) auto

lemma cdclW-stgy-conflicting-is-false:
  cdclW-stgy S S'  $\implies$  conflicting S = Some {#}  $\implies$  False
  apply (induction rule: cdclW-stgy.induct)
    unfolding full1-def apply (metis (no-types) cdclW-cp-conflicting-not-empty tranclpD)
  unfolding full-def by (metis conflict-with-false-implies-terminated other)

lemma rtranclp-cdclW-stgy-conflicting-is-false:
  cdclW-stgy** S S'  $\implies$  conflicting S = Some {#}  $\implies$  S' = S
  apply (induction rule: rtranclp-induct)
    apply simp
  using cdclW-stgy-conflicting-is-false by blast

lemma full-cdclW-restart-init-clss-with-false-normal-form:
  assumes
     $\forall m \in \text{set } M. \neg \text{is-decided } m$  and
    E = Some D and
    state S = (M, N, U, 0, E)
    full cdclW-stgy S S' and
    all-decomposition-implies-m (init-clss S) (get-all-ann-decomposition (trail S))
    cdclW-restart-learned-clause S
    cdclW-restart-M-level-inv S
    no-strange-atm S
    distinct-cdclW-restart-state S
    cdclW-restart-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 rtranclp-cdclW-stgy-conflicting-is-false unfolding full-def cdclW-restart-conflicting-def
    by fastforce
next
  case (Cons L M) note IH = this(1) and full = this(8) and E = this(10) and inv = this(2-7) and
    S = this(9) and nm = this(11)
  obtain K p where K: L = Propagated K p
    using nm by (cases L) auto
  have every-mark-is-a-conflict S using inv unfolding cdclW-restart-conflicting-def by auto
  then have MpK: M  $\models_{as}$  CNot (p - {#K#}) and Kp: K  $\in \#$  p
    using S unfolding K by fastforce+
  then have p: p = (p - {#K#}) + {#K#}
    by (auto simp add: multiset-eq-iff)
  then have K': L = Propagated K ((p - {#K#}) + {#K#})
    using K by auto
  obtain p' where
    p': hd-trail S = Propagated K p' and
    pp': p' = p
    using S K by (cases hd-trail S) auto
  have conflicting S = Some D
    using S E by (cases conflicting S) auto
  then have DD: D = D
    using S E by auto

```

```

consider (D)  $D = \{\#\}$  | (D')  $D \neq \{\#\}$  by blast
then show ?case
proof cases
  case D
  then show ?thesis
    using full rtrancpl-cdclW-stgy-conflicting-is-false S unfolding full-def E D by auto
next
  case D'
  then have no-p: no-step propagate S and no-c: no-step conflict S
    using S E by (auto elim: propagateE conflictE)
  then have no-step cdclW-cp S by (auto simp: cdclW-cp.simps)
  have res-skip:  $\exists T. (\text{resolve } S \ T \wedge \text{no-step skip } S \wedge \text{full cdcl}_W\text{-cp } T \ T)$ 
     $\vee (\text{skip } S \ T \wedge \text{no-step resolve } S \wedge \text{full cdcl}_W\text{-cp } T \ T)$ 
  proof cases
    assume  $\neg \text{lit-of } L \notin \# D$ 
    then obtain T where sk: skip S T
      using S D' K skip-rule unfolding E by fastforce
    then have res: no-step resolve S
      using  $\langle \neg \text{lit-of } L \notin \# D \rangle$  S D' K unfolding E
      by (auto elim!: skipE resolveE)
    have full cdclW-cp T T
      using sk by (auto intro!: option-full-cdclW-cp elim: skipE)
    then show ?thesis
      using sk res by blast
  next
    assume LD:  $\neg \neg \text{lit-of } L \notin \# D$ 
    then have D: Some D = Some ((D -  $\{\# \neg \text{lit-of } L \#\}$ ) +  $\{\# \neg \text{lit-of } L \#\}$ )
      by (auto simp add: multiset-eq-iff)

    have  $\bigwedge L. \text{get-level } M \ L = 0$ 
      by (simp add: nm)
    then have get-maximum-level (Propagated K (p -  $\{\#K\#\}$  +  $\{\#K\#\}$ ) # M) (D -  $\{\# \neg$ 
K $\#\}$ ) = 0
      using LD get-maximum-level-exists-lit-of-max-level
    proof -
      obtain L' where get-level (L # M) L' = get-maximum-level (L # M) D
        using LD get-maximum-level-exists-lit-of-max-level[of D L # M] by fastforce
      then show ?thesis by (metis (mono-tags) K' get-level-skip-all-not-decided
        get-maximum-level-exists-lit nm not-gr0)
    qed
    then obtain T where sk: resolve S T
      using resolve-rule[of S K p' D] S p'  $\langle K \in \# p \rangle$  D LD
      unfolding K' D E pp' by auto
    then have res: no-step skip S
      using LD S D' K unfolding E
      by (auto elim!: skipE resolveE)
    have full cdclW-cp T T
      using sk by (auto simp: option-full-cdclW-cp elim: resolveE)
    then show ?thesis
      using sk res by blast
  qed
then have step-s:  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$ 
  using (no-step cdclW-cp S) other' by (meson bj resolve skip)
have get-all-ann-decomposition (L # M) =  $[[\ ], L \# M]$ 
  using nm unfolding K apply (induction M rule: ann-lit-list-induct, simp)
  by (rename-tac L xs, case-tac hd (get-all-ann-decomposition xs), auto)+

```

```

then have no-b: no-step backtrack S
  using nm S by (auto elim: backtrackE)
have no-d: no-step decide S
  using S E by (auto elim: decideE)

have full-S-S: full cdclW-cp S S
  using S E by (auto simp add: option-full-cdclW-cp)
then have no-f: no-step (full1 cdclW-cp) S
  unfolding full-def full1-def rtrancpl-unfold by (meson trancplD)
obtain T where
  s: cdclW-stgy S T and st: cdclW-stgy** T S'
  using full step-s full unfolding full-def by (metis rtrancpl-unfold trancplD)
have resolve S T ∨ skip S T
  using s no-b no-d res-skip full-S-S cdclW-cp-state-eq-compatible resolve-unique
  skip-unique unfolding cdclW-stgy.simps cdclW-o.simps full-unfold
  full1-def by (blast dest!: trancplD elim!: cdclW-bj.cases)+
then obtain D' where T: state T = (M, N, U, 0, Some D')
  using S E by (auto elim!: skipE resolveE simp: state-eq-def simp del: state-simp)

have st-c: cdclW-restart** S T
  using E T rtrancpl-cdclW-stgy-rtrancpl-cdclW-restart s by blast
have cdclW-restart-conflicting T
  using rtrancpl-cdclW-restart-all-inv(6)[OF st-c inv(6,5,4,3,2,1)] .
show ?thesis
  apply (rule IH[of T])
    using rtrancpl-cdclW-restart-all-inv(6)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancpl-cdclW-restart-all-inv(5)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancpl-cdclW-restart-all-inv(4)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancpl-cdclW-restart-all-inv(3)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancpl-cdclW-restart-all-inv(2)[OF st-c inv(6,5,4,3,2,1)] apply blast
    using rtrancpl-cdclW-restart-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[of init-state N] by auto

  then have cdclW-stgy: ∃ S'. cdclW-stgy ?S S'
    using cdclW-cp.conflict'[of ?S] conflict-is-full1-cdclW-cp cdclW-stgy.intros(1) by metis
  have S' ≠ ?S using ⟨no-step cdclW-stgy S'⟩ cdclW-stgy by blast

```

```

then obtain  $St :: 'st$  where  $St: cdcl_W\text{-stgy } ?S \text{ } St$  and  $cdcl_W\text{-stgy}^{**} St S'$ 
  using plus-or-eq by (metis (no-types)  $\langle cdcl_W\text{-stgy}^{**} ?S S' \rangle$  converse-rtranclpE)
have  $st: cdcl_W\text{-restart}^{**} ?S St$ 
  by (simp add: rtranclp-unfold  $\langle cdcl_W\text{-stgy } ?S St \rangle$   $cdcl_W\text{-stgy-tranclp-cdcl_W-restart}$ )

have  $\exists T. \text{conflict } ?S T$ 
  using empty not-conflict-not-any-negated-init-clss[of  $?S$ ] by force
then have  $fullSt: full1 \text{ } cdcl_W\text{-cp } ?S St$ 
  using  $St$  unfolding  $cdcl_W\text{-stgy.simps}$  by blast
then have  $bt: \text{backtrack-lvl } St = (0::nat)$ 
  using  $rtranclp\text{-}cdcl_W\text{-cp-backtrack-lvl}$  unfolding  $full1\text{-def}$ 
  by (fastforce dest!:  $\text{tranclp-into-rtranclp}$ )
have  $cls\text{-}St: \text{init-clss } St = N$ 
  using  $fullSt \text{ } cdcl_W\text{-stgy-no-more-init-clss}$ [OF  $St$ ] by auto
have  $\text{conflicting } St \neq None$ 
  proof (rule ccontr)
    assume  $\text{conf}: \neg ?thesis$ 
    obtain  $E$  where
       $ES: E \in \# \text{ } \text{init-clss } St$  and
       $E: E = \{\#\}$ 
    using empty cls-St by metis
    then have  $\exists T. \text{conflict } St T$ 
      using empty cls-St conflict-rule[of  $St E$ ]  $ES \text{ } \text{conf}$  unfolding  $E$ 
      by (auto simp: clauses-def dest: )
    then show False using  $fullSt$  unfolding  $full1\text{-def}$  by blast
  qed

have 1:  $\forall m \in \text{set } (\text{trail } St). \neg \text{is-decided } m$ 
  using  $fullSt$  unfolding  $full1\text{-def}$  by (auto dest!:  $\text{tranclp-into-rtranclp}$ 
     $rtranclp\text{-}cdcl_W\text{-cp-dropWhile-trail}$ )
have 2:  $full \text{ } cdcl_W\text{-stgy } St S'$ 
  using  $\langle cdcl_W\text{-stgy}^{**} St S' \rangle$   $\langle \text{no-step } cdcl_W\text{-stgy } S' \rangle$   $bt$  unfolding  $full\text{-def}$  by auto
have 3: all-decomposition-implies-m
  ( $\text{init-clss } St$ )
  ( $\text{get-all-ann-decomposition}$ 
    ( $\text{trail } St$ ))
  using  $rtranclp\text{-}cdcl_W\text{-restart-all-inv}(1)$ [OF  $st$ ]  $no\text{-d } bt$  by simp
have 4:  $cdcl_W\text{-restart-learned-clause } St$ 
  using  $rtranclp\text{-}cdcl_W\text{-restart-all-inv}(2)$ [OF  $st$ ]  $no\text{-d } bt$  by simp
have 5:  $cdcl_W\text{-restart-M-level-inv } St$ 
  using  $rtranclp\text{-}cdcl_W\text{-restart-all-inv}(3)$ [OF  $st$ ]  $no\text{-d } bt$  by simp
have 6:  $\text{no-strange-atm } St$ 
  using  $rtranclp\text{-}cdcl_W\text{-restart-all-inv}(4)$ [OF  $st$ ]  $no\text{-d } bt$  by simp
have 7:  $\text{distinct-cdcl_W-restart-state } St$ 
  using  $rtranclp\text{-}cdcl_W\text{-restart-all-inv}(5)$ [OF  $st$ ]  $no\text{-d } bt$  by simp
have 8:  $cdcl_W\text{-restart-conflicting } St$ 
  using  $rtranclp\text{-}cdcl_W\text{-restart-all-inv}(6)$ [OF  $st$ ]  $no\text{-d } bt$  by simp
have  $\text{init-clss } S' = \text{init-clss } St$  and  $\text{conflicting } S' = \text{Some } \{\#\}$ 
  using  $\langle \text{conflicting } St \neq None \rangle$   $full\text{-}cdcl_W\text{-restart-init-clss-with-false-normal-form}$ [OF 1, of - -  $St$ ]
  2 3 4 5 6 7 8 apply (metis  $\langle cdcl_W\text{-stgy}^{**} St S' \rangle$   $rtranclp\text{-}cdcl_W\text{-stgy-no-more-init-clss}$ )
  using  $\langle \text{conflicting } St \neq None \rangle$   $full\text{-}cdcl_W\text{-restart-init-clss-with-false-normal-form}$ [OF 1, of - -  $St$  - -
     $S'$ ] 2 3 4 5 6 7 8 by (metis  $bt \text{ } \text{option.exhaust prod.inject}$ )

moreover have  $\text{init-clss } S' = N$ 
  using  $\langle cdcl_W\text{-stgy}^{**} (\text{init-state } N) S' \rangle$   $rtranclp\text{-}cdcl_W\text{-stgy-no-more-init-clss}$  by fastforce
moreover have  $\text{unsatisfiable } (\text{set-mset } N)$ 

```

```

    by (meson empty satisfiable-def true-clb-empty true-clss-def)
  ultimately show ?thesis by auto
qed

```

theorem 2.9.9 page 83 of Weidenbach's book

```

lemma full-cdclW-stgy-final-state-conclusive:
  fixes S' :: 'st
  assumes full: full cdclW-stgy (init-state N) S' and no-d: distinct-mset-mset N
  shows (conflicting S' = Some {#} ∧ unsatisfiable (set-mset (init-clss S')))
    ∨ (conflicting S' = None ∧ trail S' ⊨asm init-clss S')
  using assms full-cdclW-stgy-final-state-conclusive-is-one-false
  full-cdclW-stgy-final-state-conclusive-non-false by blast

```

theorem 2.9.9 page 83 of Weidenbach's book

```

lemma full-cdclW-stgy-final-state-conclusive-from-init-state:
  fixes S' :: 'st
  assumes full: full cdclW-stgy (init-state N) S'
  and no-d: distinct-mset-mset N
  shows (conflicting S' = Some {#} ∧ unsatisfiable (set-mset N))
    ∨ (conflicting S' = None ∧ trail S' ⊨asm N ∧ satisfiable (set-mset N))
proof -
  have N: init-clss S' = N
    using full unfolding full-def by (auto dest: rtrancp-cdclW-stgy-no-more-init-clss)
  consider
    (confl) conflicting S' = Some {#} and unsatisfiable (set-mset (init-clss S'))
  | (sat) conflicting S' = None and trail S' ⊨asm init-clss S'
  using full-cdclW-stgy-final-state-conclusive[OF assms] by auto
  then show ?thesis
  proof cases
    case confl
    then show ?thesis by (auto simp: N)
  next
    case sat
    have cdclW-restart-M-level-inv (init-state N) by auto
    then have cdclW-restart-M-level-inv S'
      using full rtrancp-cdclW-stgy-consistent-inv unfolding full-def by blast
    then have consistent-interp (lits-of-l (trail S'))
      unfolding cdclW-restart-M-level-inv-def by blast
    moreover have lits-of-l (trail S') ⊨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 conflict-driven-clause-learningW
begin

```

6.1.6 Termination

The condition that no learned clause is a tautology is overkill (in the sense that the no-duplicate condition is enough), but we can reuse *simple-clss*.

The invariant contains all the structural invariants that holds,

definition *cdcl_W-restart-all-struct-inv* **where**

cdcl_W-restart-all-struct-inv $S \longleftrightarrow$
no-strange-atm $S \wedge$
cdcl_W-restart-M-level-inv $S \wedge$
 $(\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s) \wedge$
distinct-cdcl_W-restart-state $S \wedge$
cdcl_W-restart-conflicting $S \wedge$
all-decomposition-implies-m (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) \wedge
cdcl_W-restart-learned-clause S

lemma *cdcl_W-restart-all-struct-inv-inv*:

assumes *cdcl_W-restart* $S \ S'$ **and** *cdcl_W-restart-all-struct-inv* S

shows *cdcl_W-restart-all-struct-inv* S'

unfolding *cdcl_W-restart-all-struct-inv-def*

proof (*intro HOL.conjI*)

show *no-strange-atm* S'

using *cdcl_W-restart-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *auto*

show *cdcl_W-restart-M-level-inv* S'

using *cdcl_W-restart-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *fast*

show *distinct-cdcl_W-restart-state* S'

using *cdcl_W-restart-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *fast*

show *cdcl_W-restart-conflicting* S'

using *cdcl_W-restart-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *fast*

show *all-decomposition-implies-m* (*init-clss* S') (*get-all-ann-decomposition* (*trail* S'))

using *cdcl_W-restart-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *fast*

show *cdcl_W-restart-learned-clause* S'

using *cdcl_W-restart-all-inv*[*OF* *assms*(1)] *assms*(2) **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *fast*

show $\forall s \in \# \text{ learned-clss } S'. \neg \text{tautology } s$

using *assms*(1)[*THEN* *learned-clss-are-not-tautologies*] *assms*(2)

unfolding *cdcl_W-restart-all-struct-inv-def* **by** *fast*

qed

lemma *rtrancp-cdcl_W-restart-all-struct-inv-inv*:

assumes *cdcl_W-restart*^{**} $S \ S'$ **and** *cdcl_W-restart-all-struct-inv* S

shows *cdcl_W-restart-all-struct-inv* S'

using *assms* **by** *induction* (*auto intro: cdcl_W-restart-all-struct-inv-inv*)

lemma *cdcl_W-stgy-cdcl_W-restart-all-struct-inv*:

cdcl_W-stgy $S \ T \implies \text{cdcl}_W\text{-restart-all-struct-inv } S \implies \text{cdcl}_W\text{-restart-all-struct-inv } T$

by (*meson cdcl_W-stgy-trancp-cdcl_W-restart rtrancp-cdcl_W-restart-all-struct-inv-inv rtrancp-unfold*)

lemma *rtrancp-cdcl_W-stgy-cdcl_W-restart-all-struct-inv*:

cdcl_W-stgy^{**} $S \ T \implies \text{cdcl}_W\text{-restart-all-struct-inv } S \implies \text{cdcl}_W\text{-restart-all-struct-inv } T$

by (induction rule: rtrancp-induct) (auto intro: cdcl_W-stgy-cdcl_W-restart-all-struct-inv)

No Relearning of a clause

lemma *cdcl_W-o-new-clause-learned-is-backtrack-step:*

assumes *learned: $D \in \#$ learned-clss T and*

new: $D \notin \#$ learned-clss S and

cdcl_W-restart: cdcl_W-o S T and

lev: cdcl_W-restart-M-level-inv S

shows *backtrack S $T \wedge$ conflicting $S = \text{Some } D$*

using *cdcl_W-restart lev learned new*

proof (induction rule: cdcl_W-o-induct)

case (backtrack L C K i $M1$ $M2$ T) **note** *decomp = this(3) and undef = this(6) and $T = \text{this}(8)$*

and

$D-T = \text{this}(10)$ and $D-S = \text{this}(11)$

then have *$D = C$*

using *not-gr0 lev by (auto simp: cdcl_W-restart-M-level-inv-decomp)*

then show *?case*

using *T backtrack.hyps(1-5) backtrack.intros[OF backtrack.hyps(1,2)] backtrack.hyps(3-7)*

by auto

qed auto

lemma *cdcl_W-cp-new-clause-learned-has-backtrack-step:*

assumes *learned: $D \in \#$ learned-clss T and*

new: $D \notin \#$ learned-clss S and

cdcl_W-restart: cdcl_W-stgy S T and

lev: cdcl_W-restart-M-level-inv S

shows *$\exists S'. \text{backtrack } S$ $S' \wedge \text{cdcl}_W\text{-stgy}^{**} S' T \wedge \text{conflicting } S = \text{Some } D$*

using *cdcl_W-restart learned new*

proof (induction rule: cdcl_W-stgy.induct)

case (conflict' S')

then show *?case*

unfolding *full1-def by (metis (mono-tags, lifting) rtrancp-cdcl_W-cp-learned-clause-inv
trancp-into-rtrancp)*

next

case (other' $S' S''$)

then have *$D \in \#$ learned-clss S'*

unfolding *full-def by (auto dest: rtrancp-cdcl_W-cp-learned-clause-inv)*

then show *?case*

using *cdcl_W-o-new-clause-learned-is-backtrack-step[OF - $\langle D \notin \#$ learned-clss $S \rangle \langle \text{cdcl}_W\text{-o } S S' \rangle$
 $\langle \text{full cdcl}_W\text{-cp } S' S'' \rangle$ lev by (metis cdcl_W-stgy.conflict' full-unfold r-into-rtrancp
rtrancp.rtrancp-refl)*

qed

lemma *rtrancp-cdcl_W-cp-new-clause-learned-has-backtrack-step:*

assumes *learned: $D \in \#$ learned-clss T and*

new: $D \notin \#$ learned-clss S and

*cdcl_W-restart: cdcl_W-stgy^{**} S T and*

lev: cdcl_W-restart-M-level-inv S

shows *$\exists S' S''. \text{cdcl}_W\text{-stgy}^{**} S$ $S' \wedge \text{backtrack } S' S'' \wedge \text{conflicting } S' = \text{Some } D \wedge$
*cdcl_W-stgy^{**} $S'' T$**

using *cdcl_W-restart learned new*

proof (induction rule: rtrancp-induct)

case *base*

then show *?case by blast*

next

```

case (step  $T\ U$ ) note  $st = \text{this}(1)$  and  $o = \text{this}(2)$  and  $IH = \text{this}(3)$  and
 $D\text{-}U = \text{this}(4)$  and  $D\text{-}S = \text{this}(5)$ 
show ?case
proof (cases  $D \in \# \text{ learned-clss } T$ )
  case True
  then obtain  $S'\ S''$  where
     $st'$ :  $\text{cdcl}_W\text{-stgy}^{**} S\ S'$  and
     $bt$ :  $\text{backtrack } S'\ S''$  and
     $\text{confl}$ :  $\text{conflicting } S' = \text{Some } D$  and
     $st''$ :  $\text{cdcl}_W\text{-stgy}^{**} S''\ T$ 
    using  $IH\ D\text{-}S$  by metis
  have  $\text{cdcl}_W\text{-stgy}^{++} S''\ U$ 
    using  $st''\ o$  by force
  then show ?thesis
    by (meson  $bt\ \text{confl}\ \text{rtrancpl-unfold } st'$ )
  next
  case False
  have  $\text{cdcl}_W\text{-restart-}M\text{-level-inv } T$ 
    using  $\text{lev } \text{rtrancpl-cdcl}_W\text{-stgy-consistent-inv } st$  by blast
  then obtain  $S'$  where
     $bt$ :  $\text{backtrack } T\ S'$  and
     $st'$ :  $\text{cdcl}_W\text{-stgy}^{**} S'\ U$  and
     $\text{confl}$ :  $\text{conflicting } T = \text{Some } D$ 
    using  $\text{cdcl}_W\text{-cp-new-clause-learned-has-backtrack-step}[OF\ D\text{-}U\ \text{False } o]$ 
    by metis
  then have  $\text{cdcl}_W\text{-stgy}^{**} S\ T$  and
     $\text{backtrack } T\ S'$  and
     $\text{conflicting } T = \text{Some } D$  and
     $\text{cdcl}_W\text{-stgy}^{**} S'\ U$ 
    using  $o\ st$  by auto
  then show ?thesis by blast
qed
qed

```

lemma *propagate-no-more-Decided-lit*:
assumes $\text{propagate } S\ S'$
shows $\text{Decided } K \in \text{set } (\text{trail } S) \longleftrightarrow \text{Decided } K \in \text{set } (\text{trail } S')$
using *assms* **by** (*auto elim: propagateE*)

lemma *conflict-no-more-Decided-lit*:
assumes $\text{conflict } S\ S'$
shows $\text{Decided } K \in \text{set } (\text{trail } S) \longleftrightarrow \text{Decided } K \in \text{set } (\text{trail } S')$
using *assms* **by** (*auto elim: conflictE*)

lemma $\text{cdcl}_W\text{-cp-no-more-Decided-lit}$:
assumes $\text{cdcl}_W\text{-cp } S\ S'$
shows $\text{Decided } K \in \text{set } (\text{trail } S) \longleftrightarrow \text{Decided } K \in \text{set } (\text{trail } S')$
using *assms* **apply** (*induct rule: cdcl_W-cp.induct*)
using *conflict-no-more-Decided-lit propagate-no-more-Decided-lit* **by** *auto*

lemma $\text{rtrancpl-cdcl}_W\text{-cp-no-more-Decided-lit}$:
assumes $\text{cdcl}_W\text{-cp}^{**} S\ S'$
shows $\text{Decided } K \in \text{set } (\text{trail } S) \longleftrightarrow \text{Decided } K \in \text{set } (\text{trail } S')$
using *assms* **apply** (*induct rule: rtrancpl-induct*)
using $\text{cdcl}_W\text{-cp-no-more-Decided-lit}$ **by** *blast+*

lemma *cdcl_W-o-no-more-Decided-lit*:
assumes *cdcl_W-o S S'* **and** *lev: cdcl_W-restart-M-level-inv S* **and** $\neg \text{decide } S \ S'$
shows *Decided K \in set (trail S') \longrightarrow Decided K \in set (trail S)*
using *assms*

proof (*induct rule: cdcl_W-o-induct*)
case *backtrack* **note** *decomp = this(3)* **and** *undef = this(8)* **and** *T = this(9)*
then show *?case* **using** *lev* **by** (*auto simp: cdcl_W-restart-M-level-inv-decomp*)
next
case (*decide L T*)
then show *?case* **using** *decide-rule[OF decide.hyps]* **by** *blast*
qed *auto*

lemma *cdcl_W-restart-new-decided-at-beginning-is-decide*:
assumes *cdcl_W-stgy S S'* **and**
lev: cdcl_W-restart-M-level-inv S **and**
trail S' = M' @ Decided L # 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-restart-M-level-inv S'*
using *full1-cdcl_W-cp-consistent-inv no-dup st* **by** *blast*
then have *Decided L \in set (trail S')* **and** *Decided L \notin set (trail S)*
using *no-dup unfolding S S' cdcl_W-restart-M-level-inv-def* **by** (*auto simp add: rev-image-eqI*)
then have *False*
using *st rtranclp-cdcl_W-cp-no-more-Decided-lit[of S S']*
unfolding *full1-def rtranclp-unfold* **by** *blast*
then show *?case* **by** *fast*

next
case (*other' T U*) **note** *o = this(1)* **and** *ns = this(2)* **and** *st = this(3)* **and** *no-dup = this(4)* **and**
S' = this(5) **and** *S = this(6)*
have *cdcl_W-restart-M-level-inv U*
by (*metis (full-types) lev cdcl_W-restart.simps cdcl_W-restart-consistent-inv full-def o*
other'.hyps(3) rtranclp-cdcl_W-cp-consistent-inv)
then have *Decided L \in set (trail U)* **and** *Decided L \notin set (trail S)*
using *no-dup unfolding S S' cdcl_W-restart-M-level-inv-def* **by** (*auto simp add: rev-image-eqI*)
then have *Decided L \in set (trail T)*
using *st rtranclp-cdcl_W-cp-no-more-Decided-lit* **unfolding** *full-def* **by** *blast*
then show *?case*
using *cdcl_W-o-no-more-Decided-lit[OF o] (Decided L \notin set (trail S)) ns lev* **by** *meson*
qed

lemma *cdcl_W-o-is-decide*:
assumes *cdcl_W-o S T* **and** *lev: cdcl_W-restart-M-level-inv S*
trail T = drop (length M₀) M' @ Decided L # H @ M **and**
 $\neg (\exists M'. \text{trail } S = M' @ \text{Decided } L \# H @ M)$
shows *decide S T*
using *assms*

proof (*induction rule: cdcl_W-o-induct*)
case (*backtrack L D K i M1 M2 T*)
then obtain *c* **where** *trail S = c @ M2 @ Decided K # M1*
by *auto*
show *?case*
using *backtrack lev*
apply (*cases drop (length M₀) M'*)

```

    apply (auto simp: cdclW-restart-M-level-inv-decomp)
    using ⟨trail S = c @ M2 @ Decided K # M1⟩
    by (auto simp: cdclW-restart-M-level-inv-decomp)
next
  case decide
  show ?case using decide-rule[of S] decide(1-4) by auto
qed auto

lemma rtrancpl-cdclW-restart-new-decided-at-beginning-is-decide:
  assumes cdclW-stgy** R U and
  trail U = M' @ Decided L # H @ M and
  trail R = M and
  cdclW-restart-M-level-inv R
  shows
    ∃ S T T'. cdclW-stgy** R S ∧ decide S T ∧ cdclW-stgy** T U ∧ cdclW-stgy** S U ∧
    no-step cdclW-cp S ∧ trail T = Decided L # H @ M ∧ trail S = H @ M ∧ cdclW-stgy S T' ∧
    cdclW-stgy** T' U
  using assms
proof (induct arbitrary: M H M' i rule: rtrancpl-induct)
  case base
  then show ?case by auto
next
  case (step T U) note st = this(1) and IH = this(3) and s = this(2) and
    U = this(4) and S = this(5) and lev = this(6)
  show ?case
  proof (cases ∃ M'. trail T = M' @ Decided L # H @ M)
    case False
    with s show ?thesis using U s st S
  proof induction
    case (conflict' W) note cp = this(1) and nd = this(2) and W = this(3)
    then obtain M0 where trail W = M0 @ trail T and ndecided: ∀ l ∈ set M0. ¬ is-decided l
      using rtrancpl-cdclW-cp-dropWhile-trail unfolding full1-def rtrancpl-unfold by meson
    then have MV: M' @ Decided L # H @ M = M0 @ trail T unfolding W by simp
    then have V: trail T = drop (length M0) (M' @ Decided L # H @ M)
      by auto
    have takeWhile (Not o is-decided) M' = M0 @ takeWhile (Not o is-decided) (trail T)
      using arg-cong[OF MV, of takeWhile (Not o is-decided)] ndecided
      by (simp add: takeWhile-tail)
    from arg-cong[OF this, of length] have length M0 ≤ length M'
      unfolding length-append by (metis (no-types, lifting) Nat.le-trans le-add1
        length-takeWhile-le)
    then have False using nd V by auto
    then show ?case by fast
  next
    case (other' T' U) note o = this(1) and ns = this(2) and cp = this(3) and nd = this(4)
      and U = this(5) and st = this(6)
    obtain M0 where trail U = M0 @ trail T' and ndecided: ∀ l ∈ set M0. ¬ is-decided l
      using rtrancpl-cdclW-cp-dropWhile-trail cp unfolding full-def by meson
    then have MV: M' @ Decided L # H @ M = M0 @ trail T' unfolding U by simp
    then have V: trail T' = drop (length M0) (M' @ Decided L # H @ M)
      by auto
    have takeWhile (Not o is-decided) M' = M0 @ takeWhile (Not o is-decided) (trail T')
      using arg-cong[OF MV, of takeWhile (Not o is-decided)] ndecided
      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-take While-le)
then have tr-T': trail T' = drop (length M0) M' @ Decided L # H @ M using V by auto
then have LT': Decided L ∈ set (trail T') by auto
moreover
  have cdclW-restart-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-Decided-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'⟩ ⟨Decided L ∈ set (trail T')⟩ nd tr-T'
  by (auto simp add: Cons-eq-append-conv elim: decideE)
have T': drop (length M0) M' @ Decided L # H @ M = Decided L # trail T
  using ⟨decide T T'⟩ ⟨Decided L ∈ set (trail T')⟩ nd tr-T'
  by (auto elim: decideE)
have trail T' = Decided L # trail T
  using ⟨decide T T'⟩ ⟨Decided L ∈ set (trail T')⟩ tr-T'
  by (auto elim: decideE)
then have 5: trail T' = Decided L # 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' = Decided L # trail T'⟩
    ⟨trail T' = drop (length M0) M' @ Decided L # 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' @ Decided L # H @ M by metis
from IH[OF this S lev] obtain S' S'' S''' where
  1: cdclW-stgy** R S' and
  2: decide S' S'' and
  3: cdclW-stgy** S'' T and
  4: no-step cdclW-cp S' and
  6: trail S'' = Decided L # H @ M and
  7: trail S' = H @ M and
  8: cdclW-stgy** S' T and
  9: cdclW-stgy S' S''' and
  10: cdclW-stgy** S''' T
  by blast
have cdclW-stgy** S'' U using s ⟨cdclW-stgy** S'' T⟩ by auto
moreover have cdclW-stgy** S' U using 8 s by auto
moreover have cdclW-stgy** S''' U using 10 s by auto
ultimately show ?thesis apply - apply (rule exI[of - S'], rule exI[of - S''])
  using 1 2 4 6 7 8 9 by blast
qed
qed

```

lemma rtrancpl-cdcl_W-restart-new-decided-at-beginning-is-decide':
 assumes cdcl_W-stgy** R U and

trail $U = M' @ Decided L \# H @ M$ **and**
trail $R = M$ **and**
cdcl_W-restart-M-level-inv R
shows $\exists y y'. cdcl_W-stgy^{**} R y \wedge cdcl_W-stgy y y' \wedge \neg (\exists c. trail y = c @ Decided L \# H @ M)$
 $\wedge (\lambda a b. cdcl_W-stgy a b \wedge (\exists c. trail a = c @ Decided L \# H @ M))^{**} y' U$
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: $trail T = Decided L \# H @ M$ **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 *rtranclp-cdcl_W-restart-new-decided-at-beginning-is-decide*[*OF assms*] **by** *blast*
have $n: \neg (\exists c. trail S' = c @ Decided L \# H @ M)$ **using** *trS'* **by** *auto*
show *?thesis*
using *rtranclp-trans*[*OF st*] *rtranclp-exists-last-with-prop*[*of cdcl_W-stgy S' T' -*
 $\lambda a -. \neg (\exists c. trail a = c @ Decided L \# H @ M), OF S'T' T'U n]$
by *meson*
qed

lemma *beginning-not-decided-invert*:
assumes $A: M @ A = M' @ Decided K \# H$ **and**
nm: $\forall m \in set M. \neg is-decided m$
shows $\exists M. A = M @ Decided K \# H$
proof –
have $A = drop (length M) (M' @ Decided K \# H)$
using *arg-cong*[*OF A, of drop (length M)*] **by** *auto*
moreover have $drop (length M) (M' @ Decided K \# H) = drop (length M) M' @ Decided K \# H$
using *nm* **by** (*metis* (*no-types, lifting*) *A drop-Cons' drop-append ann-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-decided-is-decide-step*:
assumes $cdcl_W-stgy S T$
 $\neg (\exists c. trail S = c @ Decided L \# H @ M)$ **and**
 $(\lambda a b. cdcl_W-stgy a b \wedge (\exists c. trail a = c @ Decided L \# H @ M))^{**} T U$ **and**
 $\exists M'. trail U = M' @ Decided L \# H @ M$ **and**
lev: $cdcl_W-restart-M-level-inv S$
shows $\exists S'. decide S S' \wedge full cdcl_W-cp S' T \wedge no-step cdcl_W-cp S$
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** $M': trail T = M' @ Decided L \# H @ M$ **by** *metis*
obtain M'' **where** $M'': trail T = M'' @ trail S$ **and** *nm*: $\forall m \in set M''. \neg is-decided m$

```

    using cp unfolding full1-def
    by (metis rtrancp-cdclW-cp-dropWhile-trail' trancp-into-rtrancp)
have False
    using beginning-not-decided-invert[of M'' trail S M' L H @ M] M' nm nd unfolding M''
    by fast
then show ?case by fast
next
case (other' T U') note o = this(1) and ns = this(2) and cp = this(3) and nd = this(4)
    and trU' = this(5)
have cdclW-cp** T U' using cp unfolding full-def by blast
from rtrancp-cdclW-cp-dropWhile-trail[OF this]
have  $\exists M'. \text{trail } T = M' @ \text{Decided } L \# H @ M$ 
    using trU' beginning-not-decided-invert[of - trail T - L H @ M] by metis
then obtain M' where M': trail T = M' @ Decided L # H @ M
    by auto
with o lev nd cp ns
show ?case
proof (induction rule: cdclW-o-induct)
case (decide L) note dec = this(1) and cp = this(5) and ns = this(4)
    then have decide S (cons-trail (Decided L) (incr-lvl S))
        using decide.hyps decide.intros[of S] by force
    then show ?case using cp decide.premis by (meson decide-state-eq-compatible ns state-eq-ref
        state-eq-sym)
next
case (backtrack L' D K j M1 M2 T) note decomp = this(3) and undef = this(8) and
    T = this(9) and trT = this(13)
obtain MS3 where MS3: trail S = MS3 @ M2 @ Decided K # M1
    using get-all-ann-decomposition-exists-prepend[OF decomp] by metis
have tl (M' @ Decided L # H @ M) = tl M' @ Decided L # H @ M
    using lev trT T lev undef decomp by (cases M') (auto simp: cdclW-restart-M-level-inv-decomp)
then have M'': M1 = tl M' @ Decided L # H @ M
    using arg-cong[OF trT[simplified], of tl] T decomp undef lev
    by (simp add: cdclW-restart-M-level-inv-decomp)
have False using nd MS3 T undef decomp unfolding M'' by auto
then show ?case by fast
qed auto
qed
qed

```

lemma *rtrancp-cdcl_W-stgy-with-trail-end-has-trail-end:*

assumes $(\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Decided } L \# H @ M))^{**} T U$ **and**
 $\exists M'. \text{trail } U = M' @ \text{Decided } L \# H @ M$
shows $\exists M'. \text{trail } T = M' @ \text{Decided } L \# H @ M$
using *assms* **by** (induction rule: rtrancp-induct) auto

lemma *remove1-mset-eq-remove1-mset-same:*

remove1-mset L D = *remove1-mset* L' D $\implies L \in \# D \implies L = L'$
by (metis diff-single-trivial insert-DiffM multi-drop-mem-not-eq single-eq-single
 union-right-cancel)

lemma *cdcl_W-o-cannot-learn:*

assumes
cdcl_W-o y z **and**
lev: cdcl_W-restart-M-level-inv y **and**
M: trail y = c @ Decided Kh # H and
DL: D $\notin \#$ learned-clss y and

$LD: L \in \# D$ **and**
 $DH: \text{atms-of } (\text{remove1-mset } L D) \subseteq \text{atm-of ' lits-of-l } H$ **and**
 $LH: \text{atm-of } L \notin \text{atm-of ' lits-of-l } H$ **and**
 $\text{learned}: \forall T. \text{conflicting } y = \text{Some } T \longrightarrow \text{trail } y \models_{\text{as}} \text{CNot } T$ **and**
 $z: \text{trail } z = c' @ \text{Decided } Kh \# H$
shows $D \notin \# \text{learned-clss } z$
using $\text{assms}(1-2) \ M \ DL \ DH \ LH \ \text{learned } z$
proof (*induction rule: cdcl_W-o-induct*)
case (*backtrack* $L' D' K j M1 M2 T$) **note** $\text{confl} = \text{this}(1)$ **and** $LD' = \text{this}(2)$ **and** $\text{decomp} = \text{this}(3)$
and $\text{levL} = \text{this}(4)$ **and** $\text{levD} = \text{this}(5)$ **and** $j = \text{this}(6)$ **and** $\text{lev-K} = \text{this}(7)$ **and** $T = \text{this}(8)$ **and**
 $z = \text{this}(15)$
def $i \equiv \text{get-level } (\text{trail } T) \ Kh$
have $\text{levT}: \text{cdcl}_W\text{-restart-M-level-inv } T$
using *backtrack-rule*[*OF* $\text{confl } LD' \ \text{decomp } \text{levL } \text{levD} \ - \ T$] $\text{lev-K } j \ \text{lev}$
by (*metis* *Suc-eq-plus1* *cdcl_W-restart.simps* *cdcl_W-bj.simps* *cdcl_W-restart-consistent-inv* *cdcl_W-o.simps*)
obtain $M3$ **where** $M3: \text{trail } y = M3 @ M2 @ \text{Decided } K \# M1$
using *decomp* *get-all-ann-decomposition-exists-prepend* **by** *metis*
have $c' @ \text{Decided } Kh \# H = \text{Propagated } L' D' \# \text{trail } (\text{reduce-trail-to } M1 \ y)$
using $z \ \text{decomp } T \ \text{lev}$ **by** (*force simp: cdcl_W-restart-M-level-inv-def*)
then obtain d **where** $d: M1 = d @ \text{Decided } Kh \# H$
by (*metis* (*no-types*) *decomp in-get-all-ann-decomposition-trail-update-trail list.inject*
 $\text{list.sel}(3) \ \text{ann-lit.distinct}(1) \ \text{self-append-conv2 } \text{tl-append2}$)

have $\text{atm-of } Kh \notin \text{atm-of ' lits-of-l } c'$
using levT **unfolding** *cdcl_W-restart-M-level-inv-def* z
by (*auto simp: atm-lit-of-set-lits-of-l*)
then have $\text{count-H}: \text{count-decided } H = i - 1 \ i > 0$
unfolding $z \ i\text{-def}$ **by** *auto*
have $n\text{-d-y}: \text{no-dup } (\text{trail } y)$ **and** $\text{bt-y}: \text{backtrack-lvl } y = \text{count-decided } (\text{trail } y)$
using lev **unfolding** *cdcl_W-restart-M-level-inv-def* **by** *auto*
have $\text{tr-T}: \text{trail } T = \text{Propagated } L' D' \# M1$
using *decomp* $T \ n\text{-d-y}$ **by** *auto*

show *?case*
proof
assume $D \in \# \text{learned-clss } T$
then have $DLD': D = D'$
using $DL \ T \ \text{neq0-conv} \ \text{decomp } n\text{-d-y}$ **by** *fastforce*
have $L\text{-cKh}: \text{atm-of } L \in \text{atm-of ' lits-of-l } (c @ [\text{Decided } Kh])$
using $LH \ \text{learned } M \ DLD'[\text{symmetric}] \ \text{confl } LD' \ LD$

apply (*auto simp add: image-iff dest!: in-CNot-implies-uminus*)
apply (*metis atm-of-uminus*) **done**
then consider (Lc) $\text{atm-of } L \in \text{atm-of ' lits-of-l } c$ **and** $\text{atm-of } L \neq \text{atm-of } Kh \mid$
 $(LKh) \ \text{atm-of } L = \text{atm-of } Kh$ **and** $\text{atm-of } L \notin \text{atm-of ' lits-of-l } c$
using $n\text{-d-y } M$ **by** (*auto simp: atm-lit-of-set-lits-of-l*)
then have $\text{lev-L-c-Kh}: \text{get-level } (c @ [\text{Decided } Kh]) \ L \geq 1$
by *cases auto*
have $\text{get-level } (\text{trail } y) \ L = \text{get-level } (c @ [\text{Decided } Kh]) \ L + \text{count-decided } H$
using *get-rev-level-skip-end*[*OF* $L\text{-cKh}, \text{of } H$] **unfolding** M **by** *simp*
then have $\text{get-level } (\text{trail } y) \ L \geq i$
using $\text{count-H } \text{lev-L-c-Kh}$ **by** *linarith*
then have $i\text{-le-bt-y}: i \leq \text{backtrack-lvl } y$
using *cdcl_W-restart-M-level-inv-get-level-le-backtrack-lvl*[*OF* $\text{lev}, \text{of } L$] **by** *linarith*
have $DD'[\text{simp}]: \text{remove1-mset } L D = D' - \{\#L'\# \}$
proof (*rule ccontr*)


```

assume  $DD': \neg ?thesis$ 
then have  $L' \in \# \text{ remove1-mset } L \ D$  using  $DLD' \ LD$  by (metis  $LD'$  in-remove1-mset-neq)
then have  $\text{get-level } (\text{trail } y) \ L' \leq \text{get-maximum-level } (\text{trail } y) \ (\text{remove1-mset } L \ D)$ 
  using  $\text{get-maximum-level-ge-get-level}$  by blast
moreover
have  $\forall x \in \text{atms-of } (\text{remove1-mset } L \ D). \ x \notin \text{atm-of ' lits-of-l } (c \ @ \ \text{Decided } Kh \ \# \ [])$ 
  using  $DH \ n\text{-d-y unfolding } M$  by (auto simp: atm-lit-of-set-lits-of-l)
from  $\text{get-maximum-level-skip-beginning}[OF \ \text{this}, \ \text{of } H]$ 
  have  $\text{get-maximum-level } (\text{trail } y) \ (\text{remove1-mset } L \ D) =$ 
     $\text{get-maximum-level } H \ (\text{remove1-mset } L \ D)$ 
  unfolding  $M$  by (simp add: get-maximum-level-skip-beginning)
moreover
have  $\text{atm-of } Kh \notin \text{atm-of ' lits-of-l } c'$ 
  using  $\text{levT unfolding } \text{cdcl}_W\text{-restart-M-level-inv-def } z$ 
  by (auto simp: atm-lit-of-set-lits-of-l)
then have  $\text{count-decided } H < i$ 
  unfolding  $i\text{-def } z$  by auto
then have  $0 < i - \text{count-decided } H$ 
  by presburger
ultimately have  $\text{get-maximum-level } (\text{trail } y) \ (\text{remove1-mset } L \ D) < i$ 
  by (metis (no-types) count-decided-ge-get-maximum-level diff-is-0-eq diff-le-mono2 not-le)
moreover
have  $L \in \# \text{ remove1-mset } L' \ D'$ 
  using  $DLD'[symmetric] \ DD' \ LD$  by (metis in-remove1-mset-neq)
then have  $\text{get-maximum-level } (\text{trail } y) \ (\text{remove1-mset } L' \ D') \geq$ 
   $\text{get-level } (\text{trail } y) \ L$ 
  using  $\text{get-maximum-level-ge-get-level}$  by blast
moreover
have  $\text{get-maximum-level } (\text{trail } y) \ (\text{remove1-mset } L' \ D')$ 
   $< \text{get-level } (\text{trail } y) \ L$ 
  using  $\langle \text{get-level } (\text{trail } y) \ L' \leq \text{get-maximum-level } (\text{trail } y) \ (\text{remove1-mset } L \ D) \rangle$ 
   $\text{calculation}(1) \ i\text{-le-bt-y levL}$  by linarith
ultimately show False using backtrack.hyps(4) by linarith
qed
then have  $LL': L = L'$ 
  using  $LD \ LD' \ \text{remove1-mset-eq-remove1-mset-same}$  unfolding  $DLD'[symmetric]$  by fast

have [simp]:  $\text{atm-of } K \notin \text{atm-of ' lits-of-l } M2$  and
  [simp]:  $\text{atm-of } K \notin \text{atm-of ' lits-of-l } M3$ 
  using  $\text{lev unfolding } M3 \ \text{cdcl}_W\text{-restart-M-level-inv-def}$  by (auto simp: atm-lit-of-set-lits-of-l)
{ assume  $D: \text{remove1-mset } L \ D' = \{\#\}$ 
  then have  $j0: j = 0$  using  $\text{levD } j$  by (simp add: LL')
  have  $\forall m \in \text{set } M1. \neg \text{is-decided } m$ 
    using  $\text{lev-K unfolding } j0 \ M3$  by (auto simp: atm-lit-of-set-lits-of-l image-Un filter-empty-conv)
  then have False using  $d$  by auto
}
moreover {
  assume  $D[simp]: \text{remove1-mset } L \ D' \neq \{\#\}$ 
  have  $i \leq j$ 
    using  $\text{lev count-H lev-K unfolding } M3 \ d \ \text{cdcl}_W\text{-restart-M-level-inv-def}$  by (auto simp add: atm-lit-of-set-lits-of-l)
  have  $j > 0$  apply (rule ccontr)
    using  $\langle i > 0 \rangle \ \text{lev-K unfolding } M3 \ d$ 
    by (auto simp add: rev-swap[symmetric] dest!: upt-decomp-lt)

```

obtain L'' **where**
 $L'' \in \# \text{ remove1-mset } L \ D'$ **and**
 $L''D'$: $\text{get-level } (\text{trail } y) \ L'' = \text{get-maximum-level } (\text{trail } y)$
 $(\text{remove1-mset } L \ D')$
using $\text{get-maximum-level-exists-lit-of-max-level}[OF \ D, \text{ of trail } y]$ **by** *auto*
have $L''M$: $\text{atm-of } L'' \in \text{atm-of } ' \text{ lits-of-l } (\text{trail } y)$
using $\text{get-level-ge-0-atm-of-in}[of \ 0 \ L'' \text{ trail } y \] \ \langle j > 0 \rangle \text{ levD } L''D'$
 $i\text{-le-bt-}y \text{ levL}$ **by** $(\text{simp add: } LL' \ j)$
then have $L'' \in \text{lits-of-l } (\text{Decided } Kh \ \# \ d)$
proof –
{
 assume $L''H$: $\text{atm-of } L'' \in \text{atm-of } ' \text{ lits-of-l } H$
 then have $\text{atm-of } L'' \notin \text{atm-of } ' \text{ lits-of-l } (c \ @ \ [\text{Decided } Kh])$
 using $n\text{-d-}y \text{ unfolding } M$ **by** $(\text{auto simp: } \text{lits-of-def atm-of-eq-atm-of})$
 then have $\text{get-level } (\text{trail } y) \ L'' = \text{get-level } H \ L''$
 using $L''H \text{ unfolding } M$ **by** *auto*
 moreover have $\text{get-level } H \ L'' \leq \text{count-decided } H$
 by *auto*
 ultimately have *False*
 using $\langle j > 0 \rangle \ \langle i \leq j \rangle \ L''D' \ LL' \ \langle \text{get-level } H \ L'' \leq \text{count-decided } H \rangle \ \text{count-H}(1) \ j$
 unfolding count-H **by** *presburger*
}
moreover
 have $\text{atm-of } L'' \in \text{atm-of } ' \text{ lits-of-l } H$
 using $DD' \ DH \ \langle L'' \in \# \text{ remove1-mset } L \ D' \rangle \text{ atm-of-lit-in-atms-of } LL' \ LD$
 LD' **by** *fastforce*
 ultimately show *?thesis*
 using $DD' \ DH \ \langle L'' \in \# \text{ remove1-mset } L \ D' \rangle \text{ atm-of-lit-in-atms-of}$
 by *auto*
qed
moreover
 have $\text{atm-of } L'' \in \text{atms-of } (\text{remove1-mset } L \ D')$
 using $\langle L'' \in \# \text{ remove1-mset } L \ D' \rangle$ **by** $(\text{auto simp: } \text{atms-of-def})$

 then have $\text{atm-of } L'' \in \text{atm-of } ' \text{ lits-of-l } H$
 using $DH \text{ unfolding } DD' \text{ unfolding } LL'$ **by** *blast*
 ultimately have *False*
 using $n\text{-d-}y \text{ unfolding } M3 \ d \ LL'$ **by** $(\text{auto simp: } \text{lits-of-def})$
}
ultimately show *False* **by** *blast*
qed
qed *auto*

lemma $cdcl_W\text{-stgy-with-trail-end-has-not-been-learned}$:

assumes
 $cdcl_W\text{-stgy } y \ z$ **and**
 $cdcl_W\text{-restart-M-level-inv } y$ **and**
 $\text{trail } y = c \ @ \ \text{Decided } Kh \ \# \ H$ **and**
 $D \notin \# \text{ learned-clss } y$ **and**
 LD : $L \in \# \ D$ **and**
 DH : $\text{atms-of } (\text{remove1-mset } L \ D) \subseteq \text{atm-of } ' \text{ lits-of-l } H$ **and**
 LH : $\text{atm-of } L \notin \text{atm-of } ' \text{ lits-of-l } H$ **and**
 $\forall T. \text{conflicting } y = \text{Some } T \longrightarrow \text{trail } y \models_{as} CNot \ T$ **and**
 $\text{trail } z = c' \ @ \ \text{Decided } Kh \ \# \ H$
shows $D \notin \# \text{ learned-clss } z$
using *assms*

proof *induction*
case *conflict'*
then show *?case*
 unfolding *full1-def* **using** *trancpl-cdcl_W-cp-learned-clause-inv* **by** *auto*
next
case (*other' T U*) **note** *o = this(1) and cp = this(3) and lev = this(4) and trY = this(5) and*
notin = this(6) and LD = this(7) and DH = this(8) and LH = this(9) and confl = this(10) and
trU = this(11)
obtain *c' where c': trail T = c' @ Decided Kh # H*
 using *cp beginning-not-decided-invert[of - trail T c' Kh H]*
 rtrancpl-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 LD DH LH confl c']*
 rtrancpl-cdcl_W-cp-learned-clause-inv cp **unfolding** *full-def* **by** *auto*
qed

lemma *rtrancpl-cdcl_W-stgy-with-trail-end-has-not-been-learned:*

assumes
 ($\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Decided } K \# H @ [])^{**} S z$ **and**
cdcl_W-restart-all-struct-inv S and
trail S = c @ Decided K # H and
D \notin # learned-clss S and
LD: L \in # D and
DH: atm-of (remove1-mset L D) \subseteq atm-of ' lits-of-l H and
LH: atm-of L \notin atm-of ' lits-of-l H and
 $\exists c'. \text{trail } z = c' @ \text{Decided } K \# H$)
shows *D \notin # learned-clss z*
using *assms(1-4,8)*

proof (*induction rule: rtrancpl-induct*)

case *base*
then show *?case by auto[1]*

next

case (*step T U*) **note** *st = this(1) and s = this(2) and IH = this(3)[OF this(4-6)]*
and *lev = this(4) and trS = this(5) and DL-S = this(6) and trU = this(7)*
obtain *c where c: trail T = c @ Decided K # H using s by auto*
obtain *c' where c': trail U = c' @ Decided K # H using trU by blast*
have *cdcl_W-restart^{**} S T*

proof –

have $\forall p \text{ pa. } \exists s \text{ sa. } \forall sb \text{ sc sd se. } (\neg p^{**} (sb::'st) \text{ sc} \vee p \text{ s sa} \vee \text{pa}^{**} sb \text{ sc})$
 $\wedge (\neg \text{pa } s \text{ sa} \vee \neg p^{**} sd \text{ se} \vee \text{pa}^{**} sd \text{ se})$

by (*metis (no-types) mono-rtrancpl*)

then have *cdcl_W-stgy^{**} S T*

using *st by blast*

then show *?thesis*

using *rtrancpl-cdcl_W-stgy-rtrancpl-cdcl_W-restart* **by** *blast*

qed

then have *lev': cdcl_W-restart-all-struct-inv T*

using *rtrancpl-cdcl_W-restart-all-struct-inv-inv[of S T] lev by auto*

then have *confl': $\forall Ta. \text{conflicting } T = \text{Some } Ta \longrightarrow \text{trail } T \models_{as} CNot \text{ Ta}$*

unfolding *cdcl_W-restart-all-struct-inv-def cdcl_W-restart-conflicting-def* **by** *blast*

show *?case*

apply (*rule cdcl_W-stgy-with-trail-end-has-not-been-learned[OF - - c - LD DH LH confl' c']*)

using *s lev' IH c unfolding cdcl_W-restart-all-struct-inv-def* **by** *blast+*

qed

lemma *cdcl_W-stgy-new-learned-clause:*

assumes $cdcl_W\text{-stgy } S \text{ } T$ **and**
 $lev: cdcl_W\text{-restart-}M\text{-level-inv } S$ **and**
 $E \notin \# \text{ learned-clss } S$ **and**
 $E \in \# \text{ learned-clss } T$
shows $\exists S'. \text{ backtrack } S \ S' \wedge \text{ conflicting } S = \text{Some } E \wedge \text{full } cdcl_W\text{-cp } S' \ T$
using *assms*
proof *induction*
case *conflict'*
then show $?case \text{ unfolding full1-def by (auto dest: tranclp-cdcl}_W\text{-cp-learned-clause-inv)}$
next
case (*other'* $T \ U$) **note** $o = \text{this}(1)$ **and** $cp = \text{this}(3)$ **and** $\text{not-yet} = \text{this}(5)$ **and** $\text{learned} = \text{this}(6)$
have $E \in \# \text{ learned-clss } T$
using $\text{learned } cp \ rtranclp\text{-cdcl}_W\text{-cp-learned-clause-inv}$ **unfolding** *full-def* **by** *auto*
then have $\text{backtrack } S \ T$ **and** $\text{conflicting } S = \text{Some } E$
using $cdcl_W\text{-o-new-clause-learned-is-backtrack-step}[OF - \text{not-yet } o]$ lev **by** *blast+*
then show $?case$ **using** cp **by** *blast*
qed

theorem 2.9.7 page 83 of Weidenbach's book

lemma $cdcl_W\text{-stgy-no-relearned-clause}$:

assumes
 $invR: cdcl_W\text{-restart-all-struct-inv } R$ **and**
 $st': cdcl_W\text{-stgy}^{**} R \ S$ **and**
 $bt: \text{backtrack } S \ T$ **and**
 $conft: \text{conflicting } S = \text{Some } E$ **and**
 $\text{already-learned}: E \in \# \text{ clauses } S$ **and**
 $R: \text{trail } R = []$
shows *False*
proof –
have $M\text{-lev}: cdcl_W\text{-restart-}M\text{-level-inv } R$
using $invR$ **unfolding** $cdcl_W\text{-restart-all-struct-inv-def}$ **by** *auto*
have $cdcl_W\text{-restart-}M\text{-level-inv } S$
using $M\text{-lev } assms(2) \ rtranclp\text{-cdcl}_W\text{-stgy-consistent-inv}$ **by** *blast*
with bt **obtain** $L \ K :: 'v \ \text{literal}$ **and** $M1 \ M2\text{-loc} :: ('v, 'v \ \text{clause}) \ \text{ann-lits}$
and $i :: \text{nat}$ **where**
 $T: T \sim \text{cons-trail } (\text{Propagated } L \ E)$
 $(\text{reduce-trail-to } M1 \ (\text{add-learned-cls } E$
 $(\text{update-backtrack-lvl } i \ (\text{update-conflicting } \text{None } S))))$
and
 $\text{decomp}: (\text{Decided } K \ \# \ M1, \ M2\text{-loc}) \in$
 $\text{set } (\text{get-all-ann-decomposition } (\text{trail } S))$ **and**
 $LD: L \in \# \ E$ **and**
 $k: \text{get-level } (\text{trail } S) \ L = \text{backtrack-lvl } S$ **and**
 $\text{level}: \text{get-level } (\text{trail } S) \ L = \text{get-maximum-level } (\text{trail } S) \ E$ **and**
 $\text{conft-S}: \text{conflicting } S = \text{Some } E$ **and**
 $i: i = \text{get-maximum-level } (\text{trail } S) \ (\text{remove1-mset } L \ E)$ **and**
 $\text{lev-K}: \text{get-level } (\text{trail } S) \ K = \text{Suc } i$
using conft **apply** (*induction rule: backtrack.induct*)
apply (*simp del: state-simp*)
by *blast*
obtain $M2$ **where**
 $M: \text{trail } S = M2 \ @ \ \text{Decided } K \ \# \ M1$
using $\text{get-all-ann-decomposition-exists-prepend}[OF \ \text{decomp}]$ **unfolding** i **by** (*metis append-assoc*)
let $?E' = \text{remove1-mset } L \ E$
have $invS: cdcl_W\text{-restart-all-struct-inv } S$
using $invR \ rtranclp\text{-cdcl}_W\text{-restart-all-struct-inv-inv } rtranclp\text{-cdcl}_W\text{-stgy-rtranclp-cdcl}_W\text{-restart } st'$ **by**

blast

then have *conf*: *cdcl_W-restart-conflicting S* **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *blast*
then have *trail S* \models_{as} *CNot E* **unfolding** *cdcl_W-restart-conflicting-def confl-S* **by** *auto*
then have *MD*: *trail S* \models_{as} *CNot E* **by** *auto*
then have *MD'*: *trail S* \models_{as} *CNot ?E'* **using** *true-annot-CNot-diff* **by** *blast*
have *lev'*: *cdcl_W-restart-M-level-inv S* **using** *invS* **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *blast*

have *lev*: *cdcl_W-restart-M-level-inv R* **using** *invR* **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** *blast*
then have *vars-of-D*: *atms-of ?E' \subseteq atm-of ' lits-of-l M1*
using *backtrack-atms-of-D-in-M1[OF lev' - decomp - -, of E - i T] confl-S conf T decomp k*
level lev' lev-K **unfolding** *i cdcl_W-restart-conflicting-def* **by** (*auto simp: cdcl_W-restart-M-level-inv-decomp*)
have *no-dup (trail S)* **using** *lev'* **by** (*auto simp: cdcl_W-restart-M-level-inv-decomp*)
have *vars-in-M1*:
 $\forall x \in \text{atms-of } ?E'. x \notin \text{atm-of ' lits-of-l } (M2 @ [\text{Decided } K])$
unfolding *Set.Ball-def* **apply** (*intro impI allI*)
apply (*rule vars-of-D distinct-atms-of-incl-not-in-other[of*
M2 @ Decided K # [] M1 ?E'])
using (*no-dup (trail S)*) *M vars-of-D* **by** *simp-all*
have *M1-D*: *M1* \models_{as} *CNot ?E'*
using *vars-in-M1 true-annots-remove-if-notin-vars[of M2 @ Decided K # [] M1 CNot ?E']*
MD' M **by** *simp*

have *backtrack-lvl S > 0* **using** *lev'* **unfolding** *cdcl_W-restart-M-level-inv-def M* **by** *auto*

obtain *M1' K' Ls* **where**

M': *trail S = Ls @ Decided K' # M1'* **and**

Ls: $\forall l \in \text{set } Ls. \neg \text{is-decided } l$ **and**

set M1 \subseteq set M1'

proof –

let *?Ls = takeWhile (Not o is-decided) (trail S)*

have *MLs*: *trail S = ?Ls @ dropWhile (Not o is-decided) (trail S)*

by *auto*

have *dropWhile (Not o is-decided) (trail S) $\neq []$* **unfolding** *M* **by** *auto*

moreover

from *hd-dropWhile[OF this]* **have** *is-decided(hd (dropWhile (Not o is-decided) (trail S)))*

by *simp*

ultimately

obtain *K'* **where**

K'k: *dropWhile (Not o is-decided) (trail S)*

$= \text{Decided } K' \# \text{tl (dropWhile (Not o is-decided) (trail S))}$

by (*cases dropWhile (Not o is-decided) (trail S);*
cases hd (dropWhile (Not o is-decided) (trail S)))

simp-all

moreover have $\forall l \in \text{set } ?Ls. \neg \text{is-decided } l$ **using** *set-takeWhileD* **by** *force*

moreover have *set M1 \subseteq set (tl (dropWhile (Not o is-decided) (trail S)))*

unfolding *M* **by** (*induction M2*) *auto*

ultimately show *?thesis* **using** *that[of takeWhile (Not o is-decided) (trail S)*

K' tl (dropWhile (Not o is-decided) (trail S))] MLs **by** *simp*

qed

have *M1'-D*: *M1'* \models_{as} *CNot ?E'* **using** *M1-D (set M1 \subseteq set M1')* **by** (*auto intro: true-annots-mono*)

have $\neg L \in \text{lits-of-l (trail S)}$ **using** *conf confl-S LD* **unfolding** *cdcl_W-restart-conflicting-def*

by (*auto simp: in-CNot-implies-uminus*)

have *L-notin*: *atm-of L \in atm-of ' lits-of-l Ls \vee atm-of L = atm-of K'*

proof (*rule ccontr*)

assume $\neg ?thesis$

```

then have atm-of L  $\notin$  atm-of ' lits-of-l (Decided K' # rev Ls) by simp
then have get-level (trail S) L = get-level M1' L
  unfolding M' by auto
moreover
  have get-level M1' L  $\leq$  count-decided M1'
    by auto
  then have get-level M1' L < backtrack-lvl S
    using lev' unfolding cdclW-restart-M-level-inv-def M'
    by (auto simp del: count-decided-ge-get-level)
ultimately show False using k by linarith
qed
obtain Y Z where
  RY: cdclW-stgy** R Y and
  YZ: cdclW-stgy Y Z and
  nt:  $\neg (\exists c. \text{trail } Y = c @ \text{Decided } K' \# M1' @ [])$  and
  Z:  $(\lambda a b. \text{cdcl}_W\text{-stgy } a b \wedge (\exists c. \text{trail } a = c @ \text{Decided } K' \# M1' @ []))^{**} Z S$ 
  using rtrancpl-cdclW-restart-new-decided-at-beginning-is-decide'[OF st' - lev, of Ls K'
    M1' []] unfolding R M' by auto
have [simp]: cdclW-restart-M-level-inv Y
  using RY lev rtrancpl-cdclW-stgy-consistent-inv by blast
obtain M' where trZ: trail Z = M' @ Decided K' # 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-restart-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-decided-is-decide-step[OF YZ nt Z] M' by auto
have trY: trail Y = M1'
proof -
  obtain M' where M: trail Z = M' @ Decided K' # M1'
    using rtrancpl-cdclW-stgy-with-trail-end-has-trail-end[OF Z] M' by auto
  obtain M'' where M'': trail Z = M'' @ trail Y' and  $\forall m \in \text{set } M''. \neg \text{is-decided } m$ 
    using Y'Z rtrancpl-cdclW-cp-dropWhile-trail' unfolding full-def by blast
  obtain M''' where trail Y' = M''' @ Decided K' # M1'
    using M'' unfolding M
    by (metis (no-types, lifting)  $\forall m \in \text{set } M''. \neg \text{is-decided } m$  beginning-not-decided-invert)
  then show ?thesis using dec nt by (induction M''') (auto elim: decideE)
qed
have Y-CT: conflicting Y = None using (decide Y Y') by (auto elim: decideE)
have cdclW-restart** R Y by (simp add: RY rtrancpl-cdclW-stgy-rtrancpl-cdclW-restart)
then have init-clss Y = init-clss R using rtrancpl-cdclW-restart-init-clss[of R Y] M-lev by auto
{ assume DL: E  $\in$  # clauses Y
  have atm-of L  $\notin$  atm-of ' lits-of-l M1
    apply (rule backtrack-lit-skipped[of - S])
    using decomp i k lev' lev-K unfolding cdclW-restart-M-level-inv-def by auto
  then have LM1: undefined-lit M1 L
    by (metis Decided-Propagated-in-iff-in-lits-of-l 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 Ex (propagate Y)
    using propagate-rule[of Y E L] DL M1'-D L-trY Y-CT trY LD by auto
  then have False using (no-step cdclW-cp Y) propagate' by blast
}

```

```

moreover {
  assume  $DL: E \notin \# \text{ clauses } Y$ 
  have  $lY-lZ: \text{learned-clss } Y = \text{learned-clss } Z$ 
    using  $dec\ Y'Z\ rtranclp\ cdcl_W\ cp\ \text{learned-clause-inv}[of\ Y'\ Z]$  unfolding  $full-def$ 
    by  $(auto\ elim: decideE)$ 
  have  $invZ: cdcl_W\ \text{restart-all-struct-inv } Z$ 
    by  $(meson\ RY\ YZ\ invR\ r\ \text{into-rtranclp}\ rtranclp\ cdcl_W\ \text{restart-all-struct-inv-inv}\ rtranclp\ cdcl_W\ stgy\ rtranclp\ cdcl_W\ \text{restart})$ 
  have  $n: E \notin \# \text{ learned-clss } Z$ 
    using  $DL\ lY-lZ\ YZ$  unfolding  $clauses-def$  by  $auto$ 
  have  $E \notin \# \text{ learned-clss } S$ 
    apply  $(rule\ rtranclp\ cdcl_W\ stgy\ \text{with-trail-end-has-not-been-learned}[OF\ Z\ invZ\ trZ])$ 
      apply  $(simp\ add: n)$ 
      using  $LD$  apply  $simp$ 
      apply  $(metis\ (no-types,\ lifting)\ \langle set\ M1 \subseteq set\ M1' \rangle\ image-mono\ order-trans\ vars-of-D\ lits-of-def)$ 
      using  $L\ \text{notin}\ \langle no-dup\ (trail\ S) \rangle$  unfolding  $M'$  by  $(auto\ simp\ add: image-iff\ lits-of-def)$ 
  then have  $False$ 
    using  $already-learned\ DL\ confl\ st'\ M\ \text{lev}\ rtranclp\ cdcl_W\ stgy\ \text{no-more-init-clss}[of\ R\ S]$ 
    unfolding  $M'$ 
    by  $(simp\ add: \langle init-clss\ Y = init-clss\ R \rangle\ clauses-def\ confl-S\ rtranclp\ cdcl_W\ stgy\ \text{no-more-init-clss})$ 
}
ultimately show  $False$  by  $blast$ 
qed

```

lemma $rtranclp\ cdcl_W\ stgy\ \text{distinct-mset-clauses}:$

```

assumes
   $invR: cdcl_W\ \text{restart-all-struct-inv } R$  and
   $st: cdcl_W\ stgy^{**}\ R\ S$  and
   $dist: \text{distinct-mset } (clauses\ R)$  and
   $R: trail\ R = []$ 
shows  $\text{distinct-mset } (clauses\ S)$ 
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  $(cases\ rule: cdcl_W\ stgy.cases)$ 
      case  $conflict'$ 
      then show  $?thesis$ 
        using  $IH$  unfolding  $full1-def$  by  $(auto\ dest: tranclp\ cdcl_W\ cp\ \text{no-more-clauses})$ 
      next
      case  $(other'\ S')$  note  $o = this(1)$  and  $full = this(3)$ 
      have  $[simp]: clauses\ T = clauses\ S'$ 
        using  $full$  unfolding  $full-def$  by  $(auto\ dest: rtranclp\ cdcl_W\ cp\ \text{no-more-clauses})$ 
      show  $?thesis$ 
        using  $o\ IH$ 
      proof  $(cases\ rule: cdcl_W\ o\ \text{rule-cases})$ 
        case  $backtrack$ 
        moreover
          have  $cdcl_W\ \text{restart-all-struct-inv } S$ 
            using  $invR\ rtranclp\ cdcl_W\ stgy\ cdcl_W\ \text{restart-all-struct-inv } st$  by  $blast$ 
          then have  $cdcl_W\ \text{restart-M-level-inv } S$ 

```

```

      unfolding cdclW-restart-all-struct-inv-def by auto
ultimately obtain E where
  conflicting S = Some E and
  cls-S': clauses S' = {#E#} + clauses S
  using ⟨cdclW-restart-M-level-inv S⟩
  by (induction rule: backtrack.induct) (auto simp: cdclW-restart-M-level-inv-decomp)
then have E ∉ # clauses S
  using cdclW-stgy-no-relearned-clause R invR local.backtrack st by blast
then show ?thesis using IH by (simp add: distinct-mset-add-single cls-S')
qed (auto elim: decideE skipE resolveE)
qed
qed

```

```

lemma cdclW-stgy-distinct-mset-clauses:
  assumes
    st: cdclW-stgy** (init-state N) S and
    no-duplicate-clause: distinct-mset N and
    no-duplicate-in-clause: distinct-mset-mset N
  shows distinct-mset (clauses S)
  using rtranclp-cdclW-stgy-distinct-mset-clauses[OF - st] assms
  by (auto simp: cdclW-restart-all-struct-inv-def distinct-cdclW-restart-state-def)

```

Decrease of a Measure

```

fun cdclW-restart-measure where
  cdclW-restart-measure S =
    [(3::nat) ^ (card (atms-of-mm (init-clss S))) - card (set-mset (learned-clss S)),
     if conflicting S = None then 1 else 0,
     if conflicting S = None then card (atms-of-mm (init-clss S)) - length (trail S)
     else length (trail S)
    ]

```

```

lemma length-model-le-vars-all-inv:
  assumes cdclW-restart-all-struct-inv S
  shows length (trail S) ≤ card (atms-of-mm (init-clss S))
  using assms length-model-le-vars[of S] unfolding cdclW-restart-all-struct-inv-def
  by (auto simp: cdclW-restart-M-level-inv-decomp)
end

```

```

context conflict-driven-clause-learningW
begin

```

```

lemma learned-clss-less-upper-bound:
  fixes S :: 'st
  assumes
    distinct-cdclW-restart-state S and
    ∀ s ∈ # learned-clss S. ¬tautology s
  shows card(set-mset (learned-clss S)) ≤ 3 ^ card (atms-of-mm (learned-clss S))
proof -
  have set-mset (learned-clss S) ⊆ simple-clss (atms-of-mm (learned-clss S))
  apply (rule simplified-in-simple-clss)
  using assms unfolding distinct-cdclW-restart-state-def by auto
then have card(set-mset (learned-clss S))
  ≤ card (simple-clss (atms-of-mm (learned-clss S)))
  by (simp add: simple-clss-finite card-mono)
then show ?thesis

```


by (meson atms-of-ms-finite simple-clss-card finite-set-mset order-trans)
qed

lemma *cdcl_W-restart-measure-decreasing*:

fixes $S :: 'st$

assumes

cdcl_W-restart $S S'$ **and**

no-restart:

$\neg(\text{learned-clss } S \subseteq \# \text{ learned-clss } S' \wedge [] = \text{trail } S' \wedge \text{conflicting } S' = \text{None})$

and

no-forget: $\text{learned-clss } S \subseteq \# \text{ learned-clss } S'$ **and**

no-relearn: $\bigwedge S'. \text{backtrack } S S' \implies \forall T. \text{conflicting } S = \text{Some } T \longrightarrow T \notin \# \text{ learned-clss } S$

and

alien: *no-strange-atm* S **and**

M-level: *cdcl_W-restart-M-level-inv* S **and**

no-taut: $\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$ **and**

no-dup: *distinct-cdcl_W-restart-state* S **and**

confl: *cdcl_W-restart-conflicting* S

shows $(\text{cdcl}_W\text{-restart-measure } S', \text{cdcl}_W\text{-restart-measure } S) \in \text{lexn less-than } 3$

using *assms*(1) *M-level assms*(2,3)

proof (*induct rule*: *cdcl_W-restart-all-induct*)

case (*propagate* $C L$) **note** $\text{conf} = \text{this}(1)$ **and** $\text{undef} = \text{this}(5)$ **and** $T = \text{this}(6)$

have *propa*: *propagate* S (*cons-trail* (*Propagated* $L C$) S)

using *propagate-rule*[*OF* *propagate.hyps*(1,2)] *propagate.hyps* **by** *auto*

then have *no-dup'*: *no-dup* (*Propagated* $L C \# \text{trail } S$)

using *M-level cdcl_W-restart-M-level-inv-decomp*(2) *undef defined-lit-map* **by** *auto*

let $?N = \text{init-clss } S$

have *no-strange-atm* (*cons-trail* (*Propagated* $L C$) S)

using *alien cdcl_W-restart.propagate cdcl_W-restart-no-strange-atm-inv propa M-level* **by** *blast*

then have *atm-of* ' *lits-of-l* (*Propagated* $L C \# \text{trail } S$)

$\subseteq \text{atms-of-mm } (\text{init-clss } S)$

using *undef unfolding no-strange-atm-def* **by** *auto*

then have *card* (*atm-of* ' *lits-of-l* (*Propagated* $L C \# \text{trail } S$))

$\leq \text{card } (\text{atms-of-mm } (\text{init-clss } S))$

by (*meson atms-of-ms-finite card-mono finite-set-mset*)

then have *length* (*Propagated* $L C \# \text{trail } S$) $\leq \text{card } (\text{atms-of-mm } ?N)$

using *no-dup-length-eq-card-atm-of-lits-of-l no-dup'* **by** *fastforce*

then have H : $\text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S)$

$= \text{Suc } (\text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{Suc } (\text{length } (\text{trail } S)))$

by *simp*

show *?case using conf T undef* **by** (*auto simp: H lexn3-conv*)

next

case (*decide* L) **note** $\text{conf} = \text{this}(1)$ **and** $\text{undef} = \text{this}(2)$ **and** $T = \text{this}(4)$

moreover

have *dec*: *decide* S (*cons-trail* (*Decided* L) (*incr-lvl* S))

using *decide-rule decide.hyps* **by** *force*

then have *cdcl_W-restart:cdcl_W-restart* S (*cons-trail* (*Decided* L) (*incr-lvl* S))

using *cdcl_W-restart.simps cdcl_W-o.intros* **by** *blast*

moreover

have *lev*: *cdcl_W-restart-M-level-inv* (*cons-trail* (*Decided* L) (*incr-lvl* S))

using *cdcl_W-restart M-level cdcl_W-restart-consistent-inv*[*OF* *cdcl_W-restart*] **by** *auto*

then have *no-dup*: *no-dup* (*Decided* $L \# \text{trail } S$)

using *undef unfolding cdcl_W-restart-M-level-inv-def* **by** *auto*

have *no-strange-atm* (*cons-trail* (*Decided* L) (*incr-lvl* S))

```

    using M-level alien calculation(4) cdclW-restart-no-strange-atm-inv by blast
  then have length (Decided L # (trail S))
    ≤ card (atms-of-mm (init-clss S))
    using no-dup undef
    length-model-le-vars[of cons-trail (Decided L) (incr-lvl S)]
    by fastforce
  ultimately show ?case using conf by (simp add: leW3-conv)
next
case (skip L C' M D) note tr = this(1) and conf = this(2) and T = this(5)
show ?case using conf T by (simp add: tr leW3-conv)
next
case conflict
then show ?case by (simp add: leW3-conv)
next
case resolve
then show ?case using finite by (simp add: leW3-conv)
next
case (backtrack L D K i M1 M2 T) note conf = this(1) and decomp = this(3) and T = this(8) and
lev = this(9)
have bt: backtrack S T
  using backtrack-rule[OF backtrack.hyps] by auto
have D ∉ # learned-clss S
  using no-relearn conf bt by auto
then have card-T:
  card (set-mset ({#D#} + learned-clss S)) = Suc (card (set-mset (learned-clss S)))
  by simp
have distinct-cdclW-restart-state T
  using bt M-level distinct-cdclW-restart-state-inv no-dup other cdclW-o.intros cdclW-bj.intros by blast
moreover have ∀ s ∈ # learned-clss T. ¬ tautology s
  using learned-clss-are-not-tautologies[OF cdclW-restart.other[OF cdclW-o.bj[OF
    cdclW-bj.backtrack[OF bt]]]] M-level no-taut confl by auto
ultimately have card (set-mset (learned-clss T)) ≤ 3 ^ card (atms-of-mm (learned-clss T))
  by (auto simp: learned-clss-less-upper-bound)
then have H: card (set-mset ({#D#} + learned-clss S))
  ≤ 3 ^ card (atms-of-mm ({#D#} + learned-clss S))
  using T decomp M-level by (simp add: cdclW-restart-M-level-inv-decomp)
moreover
have atms-of-mm ({#D#} + learned-clss S) ⊆ atms-of-mm (init-clss S)
  using alien conf unfolding no-strange-atm-def by auto
then have card-f: card (atms-of-mm ({#D#} + learned-clss S))
  ≤ card (atms-of-mm (init-clss S))
  by (meson atms-of-mm-finite card-mono finite-set-mset)
then have (3::nat) ^ card (atms-of-mm ({#D#} + learned-clss S))
  ≤ 3 ^ card (atms-of-mm (init-clss S)) by simp
ultimately have (3::nat) ^ card (atms-of-mm (init-clss S))
  ≥ card (set-mset ({#D#} + learned-clss S))
  using le-trans by blast
then show ?case using decomp diff-less-mono2 card-T T M-level
  by (auto simp: cdclW-restart-M-level-inv-decomp leW3-conv)
next
case restart
then show ?case using alien by (auto simp: state-eq-def simp del: state-simp)
next
case (forget C T) note no-forget = this(9)
then have C ∈ # learned-clss S and C ∉ # learned-clss T
  using forget.hyps by auto

```

then have $\neg \text{learned-clss } S \subseteq \# \text{ learned-clss } T$
 by (auto simp add: mset-leD)
 then show ?case using no-forget by blast
 qed

lemma *propagate-measure-decreasing*:
 fixes $S :: 'st$
 assumes *propagate* $S \ S'$ and *cdcl_W-restart-all-struct-inv* S
 shows $(\text{cdcl}_W\text{-restart-measure } S', \text{cdcl}_W\text{-restart-measure } S) \in \text{lexn less-than } 3$
 apply (rule *cdcl_W-restart-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: *cdcl_W-restart-all-struct-inv-def*)
 done

lemma *conflict-measure-decreasing*:
 fixes $S :: 'st$
 assumes *conflict* $S \ S'$ and *cdcl_W-restart-all-struct-inv* S
 shows $(\text{cdcl}_W\text{-restart-measure } S', \text{cdcl}_W\text{-restart-measure } S) \in \text{lexn less-than } 3$
 apply (rule *cdcl_W-restart-measure-decreasing*)
 using *assms*(1) *conflict* **apply** blast
 using *assms*(1) **apply** (auto simp: *state-eq-def* *simp* *del*: *state-simp* *elim*!: *conflictE*)[3]
 using *assms*(2) **apply** (auto simp add: *cdcl_W-restart-all-struct-inv-def* *elim*: *conflictE*)
 done

lemma *decide-measure-decreasing*:
 fixes $S :: 'st$
 assumes *decide* $S \ S'$ and *cdcl_W-restart-all-struct-inv* S
 shows $(\text{cdcl}_W\text{-restart-measure } S', \text{cdcl}_W\text{-restart-measure } S) \in \text{lexn less-than } 3$
 apply (rule *cdcl_W-restart-measure-decreasing*)
 using *assms*(1) *decide* **other** **apply** blast
 using *assms*(1) **apply** (auto simp: *state-eq-def* *simp* *del*: *state-simp* *elim*!: *decideE*)[3]
 using *assms*(2) **apply** (auto simp add: *cdcl_W-restart-all-struct-inv-def* *elim*: *decideE*)
 done

lemma *cdcl_W-cp-measure-decreasing*:
 fixes $S :: 'st$
 assumes *cdcl_W-cp* $S \ S'$ and *cdcl_W-restart-all-struct-inv* S
 shows $(\text{cdcl}_W\text{-restart-measure } S', \text{cdcl}_W\text{-restart-measure } S) \in \text{lexn less-than } 3$
 using *assms*
proof *induction*
 case *conflict'*
 then show ?case using *conflict-measure-decreasing* by blast
next
 case *propagate'*
 then show ?case using *propagate-measure-decreasing* by blast
qed

lemma *tranclp-cdcl_W-cp-measure-decreasing*:
 fixes $S :: 'st$
 assumes *cdcl_W-cp⁺⁺* $S \ S'$ and *cdcl_W-restart-all-struct-inv* S
 shows $(\text{cdcl}_W\text{-restart-measure } S', \text{cdcl}_W\text{-restart-measure } S) \in \text{lexn less-than } 3$
 using *assms*
proof *induction*
 case *base*
 then show ?case using *cdcl_W-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-restart-measure* *T*, *cdcl_W-restart-measure* *S*) ∈ *learn less-than 3* **by** *blast*

moreover have (*cdcl_W-restart-measure* *U*, *cdcl_W-restart-measure* *T*) ∈ *learn less-than 3*
using *cdcl_W-cp-measure-decreasing*[*OF step*] *rtranclp-cdcl_W-restart-all-struct-inv-inv inv*
tranclp-cdcl_W-cp-tranclp-cdcl_W-restart[*OF st*]
unfolding *trans-def rtranclp-unfold*
by *blast*

ultimately show ?*case using learn-transI*[*OF trans-less-than*] **unfolding** *trans-def* **by** *blast*
qed

lemma *cdcl_W-stgy-step-decreasing*:

fixes *R S T* :: '*st*

assumes *cdcl_W-stgy* *S T* **and**

cdcl_W-stgy^{**} *R S*

trail *R* = [] **and**

cdcl_W-restart-all-struct-inv *R*

shows (*cdcl_W-restart-measure* *T*, *cdcl_W-restart-measure* *S*) ∈ *learn less-than 3*

proof –

have *cdcl_W-restart-all-struct-inv* *S*

using *assms*

by (*metis rtranclp-unfold rtranclp-cdcl_W-restart-all-struct-inv-inv tranclp-cdcl_W-stgy-tranclp-cdcl_W-restart*)

with *assms* **show** ?*thesis*

proof *induction*

case (*conflict'* *V*) **note** *cp* = *this*(1) **and** *inv* = *this*(5)

show ?*case*

using *tranclp-cdcl_W-cp-measure-decreasing*[*OF HOL.conjunct1*[*OF cp*[*unfolded full1-def*]]] *inv*

.

next

case (*other'* *T* *U*) **note** *st* = *this*(1) **and** *H* = *this*(4,5,6,7) **and** *cp* = *this*(3)

have *cdcl_W-restart-all-struct-inv* *T*

using *cdcl_W-restart-all-struct-inv-inv other other'.hyps*(1) *other'.prems*(4) **by** *blast*

from *tranclp-cdcl_W-cp-measure-decreasing*[*OF - this*]

have *le-or-eq*: (*cdcl_W-restart-measure* *U*, *cdcl_W-restart-measure* *T*) ∈ *learn less-than 3* ∨

cdcl_W-restart-measure *U* = *cdcl_W-restart-measure* *T*

using *cp* **unfolding** *full-def rtranclp-unfold* **by** *blast*

moreover

have *cdcl_W-restart-M-level-inv* *S*

using *cdcl_W-restart-all-struct-inv-def other'.prems*(4) **by** *blast*

with *st* **have** (*cdcl_W-restart-measure* *T*, *cdcl_W-restart-measure* *S*) ∈ *learn less-than 3*

proof (*induction rule:cdcl_W-o-induct*)

case (*decide* *T*)

then show ?*case using decide-measure-decreasing H decide.intros*[*OF decide.hyps*] **by** *blast*

next

case (*backtrack* *L D K i M1 M2 T*) **note** *conf* = *this*(1) **and** *decomp* = *this*(3) **and**

undef = *this*(8) **and** *T* = *this*(9)

have *bt*: *backtrack* *S T*

apply (*rule backtrack-rule*)

using *backtrack.hyps* **by** *auto*

then have *no-relearn*: ∀ *T*. *conflicting* *S* = *Some T* → *T* ∉ *learned-clss S*

using *cdcl_W-stgy-no-relearned-clause*[*of R S T*] *H conf*

unfolding *cdcl_W-restart-all-struct-inv-def clauses-def* **by** *auto*

have *inv*: *cdcl_W-restart-all-struct-inv* *S*

using ⟨*cdcl_W-restart-all-struct-inv S*⟩ **by** *blast*

show ?*case*

```

    apply (rule cdclW-restart-measure-decreasing)
      using bt cdclW-bj.backtrack cdclW-o.bj other apply simp
      using bt T undef decomp inv unfolding cdclW-restart-all-struct-inv-def
        cdclW-restart-M-level-inv-def apply auto[]
      using bt T undef decomp inv unfolding cdclW-restart-all-struct-inv-def
        cdclW-restart-M-level-inv-def apply auto[]
      using bt no-relearn apply auto[]
      using inv unfolding cdclW-restart-all-struct-inv-def apply simp
      using inv unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def apply
simp
      using inv unfolding cdclW-restart-all-struct-inv-def apply simp
      using inv unfolding cdclW-restart-all-struct-inv-def apply simp
      using inv unfolding cdclW-restart-all-struct-inv-def by simp
  next
    case skip
    then show ?case by (auto simp: lern3-conv)
  next
    case resolve
    then show ?case by (auto simp: lern3-conv)
  qed
ultimately show ?case
  by (metis (full-types) lern-transI transD trans-less-than)
qed
qed

```

Roughly corresponds to theorem 2.9.15 page 86 of Weidenbach's book (using a different bound)

lemma *tranclp-cdcl_W-stgy-decreasing*:

```

fixes R S T :: 'st
assumes cdclW-stgy++ R S
trail R = [] and
cdclW-restart-all-struct-inv R
shows (cdclW-restart-measure S, cdclW-restart-measure R) ∈ lern less-than 3
using assms
apply induction
  using cdclW-stgy-step-decreasing[of R - R] apply blast
  using cdclW-stgy-step-decreasing[of - - R] tranclp-into-rtranclp[of cdclW-stgy R]
  lern-transI[OF trans-less-than, of 3] unfolding trans-def by blast

```

lemma *tranclp-cdcl_W-stgy-S0-decreasing*:

```

fixes R S T :: 'st
assumes
  pl: cdclW-stgy++ (init-state N) S and
  no-dup: distinct-mset-mset N
shows (cdclW-restart-measure S, cdclW-restart-measure (init-state N)) ∈ lern less-than 3

```

proof –

```

have cdclW-restart-all-struct-inv (init-state N)
  using no-dup unfolding cdclW-restart-all-struct-inv-def by auto
then show ?thesis using pl tranclp-cdclW-stgy-decreasing init-state-trail by blast

```

qed

lemma *wf-tranclp-cdcl_W-stgy*:

```

wf {(S::'st, init-state N)|
  S N. distinct-mset-mset N ∧ cdclW-stgy++ (init-state N) S}
apply (rule wf-wf-if-measure'-notation2[of lern less-than 3 - - cdclW-restart-measure])
apply (simp add: wf wf-lern)
using tranclp-cdclW-stgy-S0-decreasing by blast

```

```

lemma cdclW-cp-wf-all-inv:
  wf {(S', S). cdclW-restart-all-struct-inv S ∧ cdclW-cp S S'}
  (is wf ?R)
proof (rule wf-bounded-measure[of -
  λS. card (atms-of-mm (init-cls S))+1
  λS. length (trail S) + (if conflicting S = None then 0 else 1)], goal-cases)
case (1 S S')
then have cdclW-restart-all-struct-inv S and cdclW-cp S S' by auto
moreover then have cdclW-restart-all-struct-inv S'
using cdclW-cp.simps cdclW-restart-all-struct-inv-inv conflict cdclW-restart.intros cdclW-restart-all-struct-inv-inv
by blast+
ultimately show ?case
by (auto simp: cdclW-cp.simps state-eq-def simp del: state-simp elim!: conflictE propagateE
  dest: length-model-le-vars-all-inv)
qed

end

end

```

6.2 Merging backjump rules

```

theory CDCL-W-Merge
imports CDCL-W-Termination
begin

```

Before showing that Weidenbach's CDCL is included in NOT's CDCL, we need to work on a variant of Weidenbach's calculus: NOT's backjump assumes the existence of a clause that is suitable to backjump. This clause is obtained in W's CDCL by applying:

1. *conflict-driven-clause-learning_W.conflict* to find the conflict
2. the conflict is analysed by repetitive application of *conflict-driven-clause-learning_W.resolve* and *conflict-driven-clause-learning_W.skip*,
3. finally *conflict-driven-clause-learning_W.backtrack* is used to backtrack.

We show that this new calculus has the same final states than Weidenbach's CDCL if the calculus starts in a state such that the invariant holds and no conflict has been found yet. The latter condition holds for initial states.

6.2.1 Inclusion of the states

```

context conflict-driven-clause-learningW
begin
declare cdclW-restart.intros[intro] cdclW-bj.intros[intro] cdclW-o.intros[intro]

```

```

lemma backtrack-no-cdclW-bj:
  assumes cdcl: cdclW-bj T U and inv: cdclW-restart-M-level-inv V
  shows  $\neg$ backtrack V T
  using cdcl inv
  apply (induction rule: cdclW-bj.induct)
  apply (elim skipE, force elim!: backtrackE simp: cdclW-restart-M-level-inv-def)

```

```

  apply (elim resolveE, force elim!: backtrackE simp: cdclW-restart-M-level-inv-def)
  apply standard
  apply (elim backtrackE)
  apply (force simp del: state-simp simp add: state-eq-def cdclW-restart-M-level-inv-decomp)
done

```

skip-or-resolve corresponds to the *analyze* function in the code of MiniSAT.

```

inductive skip-or-resolve :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  s-or-r-skip[intro]: skip S T  $\Rightarrow$  skip-or-resolve S T |
  s-or-r-resolve[intro]: resolve S T  $\Rightarrow$  skip-or-resolve S T

```

```

lemma rtrancpl-cdclW-bj-skip-or-resolve-backtrack:
  assumes cdclW-bj** S U and inv: cdclW-restart-M-level-inv S
  shows skip-or-resolve** S U  $\vee$  ( $\exists$  T. skip-or-resolve** S T  $\wedge$  backtrack T U)
  using assms
proof (induction)
  case base
  then show ?case by simp
next
  case (step U V) note st = this(1) and bj = this(2) and IH = this(3)[OF this(4)]
  consider
    (SU) S = U
    | (SUP) cdclW-bj++ S U
  using st unfolding rtrancpl-unfold by blast
  then show ?case
  proof cases
  case SUP
  have  $\bigwedge$  T. skip-or-resolve** S T  $\Rightarrow$  cdclW-restart** S T
    using mono-rtrancpl[of skip-or-resolve cdclW-restart]
    by (blast intro: skip-or-resolve.cases)
  then have skip-or-resolve** S U
    using bj IH inv backtrack-no-cdclW-bj rtrancpl-cdclW-restart-consistent-inv[OF - inv] by meson
  then show ?thesis
    using bj by (auto simp: cdclW-bj.simps dest!: skip-or-resolve.intros)
  next
  case SU
  then show ?thesis
    using bj by (auto simp: cdclW-bj.simps dest!: skip-or-resolve.intros)
  qed
qed

```

```

lemma rtrancpl-skip-or-resolve-rtrancpl-cdclW-restart:
  skip-or-resolve** S T  $\Rightarrow$  cdclW-restart** S T
  by (induction rule: rtrancpl-induct)
  (auto dest!: cdclW-bj.intros cdclW-restart.intros cdclW-o.intros simp: skip-or-resolve.simps)

```

```

definition backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  backjump-l-cond  $\equiv$   $\lambda$  C C' L' S T. True

```

```

definition invNOT :: 'st  $\Rightarrow$  bool where
  invNOT  $\equiv$   $\lambda$  S. no-dup (trail S)

```

```

declare invNOT-def[simp]
end

```

```

context conflict-driven-clause-learningW

```

begin

6.2.2 More lemmas conflict-propagate and backjumping

Termination

lemma *cdcl_W-cp-normalized-element-all-inv*:
assumes *inv*: *cdcl_W-restart-all-struct-inv S*
obtains *T* **where** *full cdcl_W-cp S T*
using *assms cdcl_W-cp-normalized-element unfolding cdcl_W-restart-all-struct-inv-def* **by** *blast*
thm *backtrackE*

lemma *cdcl_W-bj-measure*:
assumes *cdcl_W-bj S T* **and** *cdcl_W-restart-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-ann-decomposition-exists-prepend
elim!: backtrackE skipE resolveE
simp: cdcl_W-restart-M-level-inv-def)+

lemma *wf-cdcl_W-bj*:
wf {(b,a). cdcl_W-bj a b ∧ cdcl_W-restart-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** *simp*

lemma *cdcl_W-bj-exists-normal-form*:
assumes *lev: cdcl_W-restart-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-restart-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-restart** S T*
using *mono-rtrancp[of cdcl_W-bj cdcl_W-restart]* **by** *blast*
then have *cdcl_W-restart-M-level-inv T*
using *rtrancp-cdcl_W-restart-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
 $\exists M. \text{trail } S = M @ \text{trail } T \wedge (\forall m \in \text{set } M. \neg \text{is-decided } m)$
init-clss S = init-clss T
learned-clss S = learned-clss T
backtrack-lvl S = backtrack-lvl T
conflicting S = conflicting T
using *assms* **by** (*induction rule: rtrancp-induct*)
(auto simp del: state-simp simp: state-eq-def elim!: skipE)

More backjumping

Backjumping after skipping or jump directly lemma *rtrancpl-skip-backtrack-backtrack*:

```

assumes
  skip** S T and
  backtrack T W and
  cdclW-restart-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-restart-all-struct-inv V
  using rtrancpl-mono[of skip cdclW-restart] assms(3) rtrancpl-cdclW-restart-all-struct-inv-inv mono-rtrancpl
  by (auto dest!: bj other cdclW-bj.skip)
then have cdclW-restart-M-level-inv V
  unfolding cdclW-restart-all-struct-inv-def by auto
then obtain K i M1 M2 L D where
  conf: conflicting V = Some D and
  LD: L ∈ # D and
  decomp: (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (trail V)) and
  lev-L: get-level (trail V) L = backtrack-lvl V and
  max: get-level (trail V) L = get-maximum-level (trail V) D and
  max-D: get-maximum-level (trail V) (remove1-mset L D) ≡ i and
  lev-k: get-level (trail V) K = Suc i and
  W: W ∼ cons-trail (Propagated L D)
    (reduce-trail-to M1
      (add-learned-cls D
        (update-backtrack-lvl i
          (update-conflicting None V))))
using bt inv by (elim backtrackE) metis+
obtain L' C' M E where
  tr: trail T = Propagated L' C' # M and
  raw: conflicting T = Some E and
  LE:  $-L' \notin \# E$  and
  E: E ≠ {#} and
  V: V ∼ tl-trail T
  using skip by (elim skipE) metis
let ?M = Propagated L' C' # trail V
have tr-M: trail T = ?M
  using tr V by auto
have MT: M = tl (trail T) and MV: M = trail V
  using tr V by auto
have DE[simp]: D = E
  using V conf raw by (auto simp add: state-eq-def simp del: state-simp)
have cdclW-restart** S T using bj cdclW-bj.skip mono-rtrancpl[of skip cdclW-restart S T] other st
by meson
then have inv': cdclW-restart-all-struct-inv T
  using rtrancpl-cdclW-restart-all-struct-inv-inv inv by blast
have M-lev: cdclW-restart-M-level-inv T using inv' unfolding cdclW-restart-all-struct-inv-def by
auto

```

```

then have  $n\text{-}d'$ :  $\text{no-dup } ?M$ 
  using  $\text{tr-}M$  unfolding  $\text{cdcl}_W\text{-restart-}M\text{-level-inv-def}$  by auto
let  $?k = \text{backtrack-lvl } T$ 
have  $[simp]$ :
   $\text{backtrack-lvl } V = ?k$ 
  using  $V$  by simp
have  $?k > 0$ 
  using decomp  $M\text{-lev } V$   $\text{tr}$  unfolding  $\text{cdcl}_W\text{-restart-}M\text{-level-inv-def}$  by auto
then have  $\text{atm-of } L \in \text{atm-of ' lits-of-l (trail } V)$ 
  using  $\text{lev-}L$   $\text{get-level-ge-0-atm-of-in[of } 0 \ L \ \text{trail } V]$  by auto
then have  $L\text{-}L'$ :  $\text{atm-of } L \neq \text{atm-of } L'$ 
  using  $n\text{-}d'$  unfolding  $\text{lits-of-def}$  by auto
have  $L'\text{-}M$ :  $\text{atm-of } L' \notin \text{atm-of ' lits-of-l (trail } V)$ 
  using  $n\text{-}d'$  unfolding  $\text{lits-of-def}$  by auto
have  $?M \models_{\text{as}} C\text{Not } D$ 
  using  $\text{inv' raw}$  unfolding  $\text{cdcl}_W\text{-restart-conflicting-def}$   $\text{cdcl}_W\text{-restart-all-struct-inv-def}$   $\text{tr-}M$  by auto
then have  $L' \notin \# (\text{remove1-mset } L \ D)$ 
  using  $L\text{-}L' \ L'\text{-}M$   $\langle \text{Propagated } L' \ C' \ \# \ \text{trail } V \models_{\text{as}} C\text{Not } D \rangle$ 
  unfolding  $\text{true-annots-true-clss true-clss-def}$ 
  by (auto simp: uminus-lit-swap atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set dest!: in-diffD)
have  $[simp]$ :  $\text{trail (reduce-trail-to } M1 \ T) = M1$ 
  using decomp  $\text{tr } W \ V$  by auto
have  $\text{skip}^{**} S \ V$ 
  using st skip by auto
have  $\text{no-dup (trail } S)$ 
  using inv unfolding  $\text{cdcl}_W\text{-restart-all-struct-inv-def}$   $\text{cdcl}_W\text{-restart-}M\text{-level-inv-def}$  by auto
then have  $[simp]$ :  $\text{init-clss } S = \text{init-clss } V$  and  $[simp]$ :  $\text{learned-clss } S = \text{learned-clss } V$ 
  using  $\text{rtranclp-skip-state-decomp[OF } \langle \text{skip}^{**} S \ V \rangle]$   $V$ 
  by (auto simp del: state-simp simp: state-eq-def)
then have
   $W\text{-}S$ :  $W \sim \text{cons-trail (Propagated } L \ E) (\text{reduce-trail-to } M1$ 
     $(\text{add-learned-clss } E (\text{update-backtrack-lvl } i (\text{update-conflicting } \text{None } T))))$ 
  using  $W \ V \ M\text{-lev}$  decomp  $\text{tr}$ 
  by (auto simp del: state-simp simp: state-eq-def cdcl}_W\text{-restart-}M\text{-level-inv-def)

obtain  $M2'$  where
   $\text{decomp'}$ :  $(\text{Decided } K \ \# \ M1, \ M2') \in \text{set (get-all-ann-decomposition (trail } T))$ 
  using decomp  $V$  unfolding  $\text{tr-}M$  by (cases  $\text{hd (get-all-ann-decomposition (trail } V))$ ,
    cases  $\text{get-all-ann-decomposition (trail } V))$  auto)
moreover
  from  $L\text{-}L'$  have  $\text{get-level } ?M \ L = ?k$ 
    using  $\text{lev-}L \ V$  by (auto split: if-split-asm)
moreover
  have  $\text{atm-of } L' \notin \text{atms-of } D$ 
    by (metis  $DE \ LE \ L\text{-}L' \ \langle L' \notin \# (\text{remove1-mset } L \ D) \rangle \text{ in-remove1-mset-neq}$ 
       $\text{atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def}$ )
  then have  $\text{get-level } ?M \ L = \text{get-maximum-level } ?M \ D$ 
    using  $\text{calculation}(2) \ \text{lev-}L \ \text{max}$  by auto
moreover
  have  $\text{atm-of } L' \notin \text{atms-of } ((\text{remove1-mset } L \ D))$ 
    by (metis  $DE \ LE \ \langle L' \notin \# (\text{remove1-mset } L \ D) \rangle$ 
       $\text{atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set atms-of-def in-remove1-mset-neq}$ 
       $\text{in-atms-of-remove1-mset-in-atms-of}$ )
  have  $i = \text{get-maximum-level } ?M \ ((\text{remove1-mset } L \ D))$ 
    using  $\text{max-}D \ \langle \text{atm-of } L' \notin \text{atms-of } ((\text{remove1-mset } L \ D)) \rangle$  by auto
moreover have  $\text{atm-of } L' \neq \text{atm-of } K$ 

```

```

using inv' get-all-ann-decomposition-exists-prepend[OF decomp]
unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def tr MV by auto
ultimately have backtrack T W
apply –
apply (rule backtrack-rule[of T - L K M1 M2' i W, OF raw])
unfolding tr-M[symmetric]
  using LD apply simp
    apply simp
    apply simp
    apply simp
    apply auto[]
  using W-S lev-k tr MV apply auto[]
using W-S lev-k apply auto[]
done
then show ?thesis using IH inv by blast
qed

```

See also theorem *rtrancpl-skip-backtrack-backtrack*

lemma *rtrancpl-skip-backtrack-backtrack-end*:

```

assumes
  skip: skip** S T and
  bt: backtrack S W and
  inv: cdclW-restart-all-struct-inv S
shows backtrack T W
using assms
proof –
have M-lev: cdclW-restart-M-level-inv S
  using bt inv unfolding cdclW-restart-all-struct-inv-def by (auto elim!: backtrackE)
then obtain K i M1 M2 L D where
  S: conflicting S = Some D and
  LD: L ∈ # D and
  decomp: (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (trail S)) and
  lev-l: get-level (trail S) L = backtrack-lvl S and
  lev-l-D: get-level (trail S) L = get-maximum-level (trail S) D and
  i: get-maximum-level (trail S) (remove1-mset L D) ≡ i and
  lev-K: get-level (trail S) K = Suc i and
  W: W ~ cons-trail (Propagated L D)
    (reduce-trail-to M1
      (add-learned-cls D
        (update-backtrack-lvl i
          (update-conflicting None S))))
  using bt by (elim backtrackE)
  (simp-all add: cdclW-restart-M-level-inv-decomp state-eq-def del: state-simp)
let ?D = remove1-mset L D

```

```

have [simp]: no-dup (trail S)
  using M-lev by (auto simp: cdclW-restart-M-level-inv-decomp)
have cdclW-restart-all-struct-inv T
  using mono-rtrancpl[of skip cdclW-restart] by (smt bj cdclW-bj.skip inv local.skip other
    rtrancpl-cdclW-restart-all-struct-inv-inv)
then have [simp]: no-dup (trail T)
  unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def by auto

```

```

obtain MS MT where M: trail S = MS @ MT and MT: MT = trail T and nm: ∀ m ∈ set MS.
¬is-decided m
  using rtrancpl-skip-state-decomp(1)[OF skip] S M-lev by auto

```

```

have T: state T = (MT, init-clss S, learned-clss S, backtrack-lvl S, Some D)
  using MT rtrancpl-skip-state-decomp[of S T] skip S
  by (auto simp del: state-simp simp: state-eq-def)

have cdclW-restart-all-struct-inv T
  apply (rule rtrancpl-cdclW-restart-all-struct-inv-inv[OF - inv])
  using bj cdclW-bj.skip local.skip other rtrancpl-mono[of skip cdclW-restart] by blast
then have MT ⊨as CNot D
  unfolding cdclW-restart-all-struct-inv-def cdclW-restart-conflicting-def using T by blast
then have ∀ L ∈ #D. atm-of L ∈ atm-of ' lits-of-l MT
  by (meson atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set
      true-annots-true-cls-def-iff-negation-in-model)
moreover have no-dup (trail S)
  using inv unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def by auto
ultimately have ∀ L ∈ #D. atm-of L ∉ atm-of ' lits-of-l MS
  unfolding M unfolding lits-of-def by auto
then have H: ∧ L. L ∈ #D ⇒ get-level (trail S) L = get-level MT L
  unfolding M by (fastforce simp: lits-of-def)
have [simp]: get-maximum-level (trail S) D = get-maximum-level MT D
  using ⟨MT ⊨as CNot D⟩ M nm ⟨∀ L ∈ #D. atm-of L ∉ atm-of ' lits-of-l MS⟩
  by (auto simp: get-maximum-level-skip-un-decided-not-present)

have lev-l': get-level MT L = backtrack-lvl S
  using lev-l LD 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-decided-invert
      get-all-ann-decomposition-exists-prepend reduce-trail-to-trail-tl-trail-decomp)
have W: W ∼ cons-trail (Propagated L D) (reduce-trail-to M1
  (add-learned-cls D (update-backtrack-lvl i (update-conflicting None T))))
  using W T i decomp by (auto simp del: state-simp simp: state-eq-def)
have lev-l-D': get-level MT L = get-maximum-level MT D
  using lev-l-D LD by (auto simp: H)
have [simp]: get-maximum-level (trail S) ?D = get-maximum-level MT ?D
  by (smt H get-maximum-level-exists-lit get-maximum-level-ge-get-level in-diffD le-antisym
      not-gr0 not-less)
then have i': i = get-maximum-level MT ?D
  using i by auto
have Decided K # M1 ∈ set (map fst (get-all-ann-decomposition (trail S)))
  using Set.imageI[OF decomp, of fst] by auto
then have Decided K # M1 ∈ set (map fst (get-all-ann-decomposition MT))
  using fst-get-all-ann-decomposition-prepend-not-decided[OF nm] unfolding M by auto
then obtain M2' where decomp': (Decided K # M1, M2') ∈ set (get-all-ann-decomposition MT)
  by auto
moreover
  have atm-of K ∉ atm-of ' lits-of-l MS
    using ⟨no-dup (trail S)⟩ decomp' unfolding M MT
    by (auto simp: lits-of-def)
  then have get-level (trail T) K = get-level (trail S) K
    unfolding M MT by auto
ultimately show backtrack T W
  apply -
  apply (rule backtrack.intros[of T D])
  using T lev-l' lev-l-D' i' W LD lev-K i apply auto[7]
  using T W unfolding i'[symmetric]
  by (auto simp del: state-simp simp: state-eq-def)
qed

```

```

lemma cdclW-bj-decomp-resolve-skip-and-bj:
  assumes cdclW-bj** S T and inv: cdclW-restart-M-level-inv S
  shows (skip-or-resolve** S T
     $\vee (\exists U. \text{skip-or-resolve** } S \ U \wedge \text{backtrack } U \ T)$ )
  using assms
proof induction
  case base
  then show ?case by simp
next
  case (step T U) note st = this(1) and bj = this(2) and IH = this(3)
  have IH: skip-or-resolve** S T
  proof –
    { assume  $\exists U. \text{skip-or-resolve** } S \ U \wedge \text{backtrack } U \ T$ 
      then obtain V where
        bt: backtrack V T and
        skip-or-resolve** S V
        by blast
      have cdclW-restart** S V
        using  $\langle \text{skip-or-resolve** } S \ V \rangle$  rtrancp-skip-or-resolve-rtrancp-cdclW-restart by blast
      then have cdclW-restart-M-level-inv V and cdclW-restart-M-level-inv S
        using rtrancp-cdclW-restart-consistent-inv inv by blast+
      with bj bt have False using backtrack-no-cdclW-bj by simp
    }
    then show ?thesis using IH inv by blast
  qed
show ?case
  using bj
  proof (cases rule: cdclW-bj.cases)
    case backtrack
    then show ?thesis using IH by blast
  qed (metis (no-types, lifting) IH rtrancp.simps skip-or-resolve.simps)+
qed

lemma resolve-skip-deterministic:
  resolve S T  $\implies$  skip S U  $\implies$  False
  by (auto elim!: skipE resolveE)

lemma list-same-level-decomp-is-same-decomp:
  assumes M-K: M = M1 @ Decided K # M2 and M-K': M = M1' @ Decided K' # M2' and
  lev-KK': get-level M K = get-level M K' and
  n-d: no-dup M
  shows K = K' and M1 = M1' and M2 = M2'
proof –
  {
    fix j j' K K' M1 M1' M2 M2'
    assume
      M-K: M = M1 @ Decided K # M2 and
      M-K': M = M1' @ Decided K' # M2' and
      lev-KK': get-level M K = get-level M K' and
      j: M ! j = Decided K and j-M: j < length M and
      j': M ! j' = Decided K' and j'-M: j' < length M and
      jj: j' > j
    have j  $\geq$  length M1
    proof (rule ccontr)
      assume  $\neg \text{length } M1 \leq j$ 

```

```

    then have  $j < \text{length } M1$ 
      by auto
    then have  $\text{Decided } K \in \text{set } M1$ 
      using  $j$  unfolding  $M-K$ 
      by (auto simp: nth-append in-set-conv-nth split: if-splits)
    from  $\text{Set.imageI}[OF \text{ this, of } \lambda L. \text{atm-of } (\text{lit-of } L)]$ 
    show  $\text{False}$  using  $n-d$  unfolding  $M-K$  by auto
  qed
moreover then have  $j' - \text{Suc } (\text{length } M1) < \text{length } M2$ 
  using  $j'-M$   $jj$   $M-K$  unfolding  $M-K'$  by (metis One-nat-def Suc-eq-plus1 add.left-commute
    le-less-trans length-append less-diff-conv2 list.size(4) not-less not-less-eq)
ultimately have  $\text{dec}: \text{Decided } K' \in \text{set } M2$ 
  using  $jj$   $j$   $j'$   $j'-M$  unfolding  $M-K$  by (auto simp: nth-append in-set-conv-nth List.nth-Cons')
obtain  $xs$   $ys$  where
   $M2: M2 = xs @ \text{Decided } K' \# ys$ 
  using  $\text{List.split-list}[OF \text{ dec}]$  by auto
have  $[\text{simp}]: \text{atm-of } K \neq \text{atm-of } K'$ 
  using  $n-d$  unfolding  $M-K$   $M2$  by auto
have  $\text{atm-of } K \notin \text{atm-of ' lits-of-l } M1$  and  $\text{atm-of } K' \notin \text{atm-of ' lits-of-l } M1$  and
 $\text{atm-of } K' \notin \text{atm-of ' lits-of-l } xs$ 
  using  $n-d$   $\text{Set.imageI}[OF \text{ dec, of } \lambda L. \text{atm-of } (\text{lit-of } L)]$  unfolding  $M-K$ 
  using  $n-d$  unfolding  $M-K$   $M2$ 
  by (auto simp: lits-of-def)
then have  $\text{False}$ 
  using  $M2$   $\text{levKK'}$  unfolding  $M-K$  by (auto simp: split: if-splits )
} note  $H = \text{this}$ 
have  $\text{Decided } K \in \text{set } M$  and  $\text{Decided } K' \in \text{set } M$ 
  using  $M-K$  apply  $\text{simp}$ 
  using  $M-K'$  by  $\text{simp}$ 
then obtain  $j$   $j'$  where
   $j: M ! j = \text{Decided } K$  and  $j-M: j < \text{length } M$  and
   $j': M ! j' = \text{Decided } K'$  and  $j'-M: j' < \text{length } M$ 
  using  $\text{in-set-conv-nth}$  by metis

have  $[\text{simp}]: j = j'$  using  $H[OF \text{ } M-K \text{ } M-K' - j \text{ } j-M \text{ } j' \text{ } j'-M]$ 
 $H[OF \text{ } M-K' \text{ } M-K - j' \text{ } j'-M \text{ } j \text{ } j-M]$   $\text{lev-KK'}$  by  $\text{presburger}$ 
then show  $KK': K = K'$  using  $j$   $j'$  by auto

have  $j-M1: j = \text{length } M1$ 
  proof (rule ccontr)
    assume  $j \neq \text{length } M1$ 
    moreover then have  $j - \text{Suc } (\text{length } M1) < \text{length } M2 \vee j < \text{length } M1$ 
      using  $j-M$   $M-K$  unfolding  $M-K'$  by force
    ultimately have  $\text{Decided } K \in \text{set } (M1 @ M2)$ 
      using  $j$  unfolding  $M-K$  by (auto simp: nth-append in-set-conv-nth split: if-splits)
    from  $\text{Set.imageI}[OF \text{ this, of } \lambda L. \text{atm-of } (\text{lit-of } L)]$ 
    show  $\text{False}$  using  $n-d$  unfolding  $M-K$  by auto
  qed
have  $j-M2: j' = \text{length } M1'$ 
  proof (rule ccontr)
    assume  $j' \neq \text{length } M1'$ 
    moreover then have  $j' - \text{Suc } (\text{length } M1') < \text{length } M2' \vee j' < \text{length } M1'$ 
      using  $j'-M$   $M-K'$  unfolding  $M-K$  by force
    ultimately have  $\text{Decided } K' \in \text{set } (M1' @ M2')$ 
      using  $j'$  unfolding  $M-K'$  by (auto simp: nth-append in-set-conv-nth split: if-splits)
    from  $\text{Set.imageI}[OF \text{ this, of } \lambda L. \text{atm-of } (\text{lit-of } L)]$ 

```

```

    show False using n-d unfolding M-K' by auto
qed

show M1 = M1' M2 = M2'
  using arg-cong[OF M-K, of take j] j-M1 arg-cong[OF M-K', of take j'] j-M2
  using arg-cong[OF M-K, of drop (j+1)] j-M1 arg-cong[OF M-K', of drop (j'+1)] j-M2
  by auto
qed

lemma backtrack-unique:
  assumes
    bt-T: backtrack S T and
    bt-U: backtrack S U and
    inv: cdclW-restart-all-struct-inv S
  shows T ~ U
proof -
  have lev: cdclW-restart-M-level-inv S
  using inv unfolding cdclW-restart-all-struct-inv-def by auto
  then obtain K i M1 M2 L D where
    S: conflicting S = Some D and
    LD: L ∈ # D and
    decomp: (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (trail S)) and
    lev-l: get-level (trail S) L = backtrack-lvl S and
    lev-l-D: get-level (trail S) L = get-maximum-level (trail S) D and
    i: get-maximum-level (trail S) (remove1-mset L D) ≡ i and
    lev-K: get-level (trail S) K = Suc i and
    T: T ~ cons-trail (Propagated L D)
      (reduce-trail-to M1
        (add-learned-cls D
          (update-backtrack-lvl i
            (update-conflicting None S))))
  using bt-T by (elim backtrackE) (force simp: cdclW-restart-M-level-inv-def)+

  obtain K' i' M1' M2' L' D' where
    S': conflicting S = Some D' and
    LD': L' ∈ # D' and
    decomp': (Decided K' # M1', M2') ∈ set (get-all-ann-decomposition (trail S)) and
    lev-l': get-level (trail S) L' = backtrack-lvl S and
    lev-l-D': get-level (trail S) L' = get-maximum-level (trail S) D' and
    i': get-maximum-level (trail S) (remove1-mset L' D') ≡ i' and
    lev-K': get-level (trail S) K' = Suc i' and
    U: U ~ cons-trail (Propagated L' D')
      (reduce-trail-to M1'
        (add-learned-cls D'
          (update-backtrack-lvl i'
            (update-conflicting None S))))
  using bt-U lev by (elim backtrackE) (force simp: cdclW-restart-M-level-inv-def)+
  obtain c where M: trail S = c @ M2 @ Decided K # M1
  using decomp by auto
  obtain c' where M': trail S = c' @ M2' @ Decided K' # M1'
  using decomp' by auto
  have n-d: no-dup (trail S) and bt: backtrack-lvl S = count-decided (trail S)
  using lev unfolding cdclW-restart-M-level-inv-def by auto
  then have atm-of K ∉ atm-of ' lits-of-l (c @ M2)
  by (auto simp: lits-of-def M)
  then have i < backtrack-lvl S

```

```

using lev-K unfolding M bt by (auto simp add: image-Un)

have [simp]: L' = L
proof (rule ccontr)
  assume  $\neg$  ?thesis
  then have  $L' \in \# \text{ remove1-mset } L D$ 
    using S S' LD LD' by (simp add: in-remove1-mset-neg)
  then have  $\text{get-maximum-level } (\text{trail } S) (\text{remove1-mset } L D) \geq \text{backtrack-lvl } S$ 
    using  $\langle \text{get-level } (\text{trail } S) L' = \text{backtrack-lvl } S \rangle \text{ get-maximum-level-ge-get-level}$ 
    by metis
  then show False using  $i' i \langle i < \text{backtrack-lvl } S \rangle$  by auto
qed
then have [simp]: D' = D
  using S S' by auto
have [simp]: i' = i
  using i i' by auto
have [simp]: K = K' and [simp]: M1 = M1'
  apply (rule list-same-level-decomp-is-same-decomp[of trail S c @ M2 K M1
    c' @ M2' K' M1'])
  using lev-K lev-K' M M' n-d apply (auto)[4]
  apply (rule list-same-level-decomp-is-same-decomp[of trail S c @ M2 K M1
    c' @ M2' K' M1'])
  using lev-K lev-K' M M' n-d apply (auto)[4]
done
show ?thesis using T U inv decomp by (auto simp del: state-simp simp: state-eq-def
  cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-decomp)
qed

lemma if-can-apply-backtrack-no-more-resolve:
  assumes
    skip: skip** S U and
    bt: backtrack S T and
    inv: cdclW-restart-all-struct-inv S
  shows  $\neg \text{resolve } U V$ 
proof (rule ccontr)
  assume resolve:  $\neg \neg \text{resolve } U V$ 

  obtain L E D where
    U: trail U  $\neq []$  and
    tr-U: hd-trail U = Propagated L E and
    LE: L  $\in \#$  E and
    confl-U: conflicting U = Some D and
    LD:  $-L \in \#$  D and
    get-maximum-level (trail U) ((remove1-mset (-L) D)) = backtrack-lvl U and
    V:  $V \sim \text{update-conflicting } (\text{Some } (\text{resolve-cls } L D E)) (\text{tl-trail } U)$ 
  using resolve by (auto elim!: resolveE)
  have inv-U: cdclW-restart-all-struct-inv U
    using mono-rtrancp[of skip cdclW-restart] by (meson bj cdclW-bj.skip inv local.skip other
      rtrancp-cdclW-restart-all-struct-inv-inv)
  then have [iff]: no-dup (trail S) cdclW-restart-M-level-inv S and [iff]: no-dup (trail U)
    using inv unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def by blast+
  have inv-V: cdclW-restart-all-struct-inv V
    using mono-rtrancp[of resolve cdclW-restart] inv-U resolve cdclW-restart.simps cdclW-restart-all-struct-inv-inv
    cdclW-bj.resolve cdclW-o.simps by blast
  have
    S: init-clss U = init-clss S

```



```

    learned-clss U = learned-clss S
    backtrack-lvl U = backtrack-lvl S
    backtrack-lvl V = backtrack-lvl S
    conflicting S = Some D
  using rtrncplp-skip-state-decomp[OF skip] U confl-U V
  by (auto simp del: state-simp simp: state-eq-def)
obtain M0 where
  tr-S: trail S = M0 @ trail U and
  nm:  $\forall m \in \text{set } M_0. \neg \text{is-decided } m$ 
  using rtrncplp-skip-state-decomp[OF skip] by blast

obtain K' i' M1' M2' L' D' where
  S': conflicting S = Some D' and
  LD': L'  $\in \#$  D' and
  decomp': (Decided K'  $\#$  M1', M2')  $\in$  set (get-all-ann-decomposition (trail S)) and
  lev-l: get-level (trail S) L' = backtrack-lvl S and
  lev-l-D: get-level (trail S) L' = get-maximum-level (trail S) D' and
  i': get-maximum-level (trail S) (remove1-mset L' D')  $\equiv$  i' and
  lev-K': get-level (trail S) K' = Suc i' and
  R: T  $\sim$  cons-trail (Propagated L' D')
    (reduce-trail-to M1'
      (add-learned-cls D'
        (update-backtrack-lvl i'
          (update-conflicting None S))))
  using bt by (elim backtrackE) metis
obtain c where M: trail S = c @ M2' @ Decided K'  $\#$  M1'
  using get-all-ann-decomposition-exists-prepend[OF decomp'] by auto
have i' < backtrack-lvl S
  using count-decided-ge-get-level[of K' trail S] inv
  unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def lev-K'
  by linarith

have U: trail U = Propagated L E  $\#$  trail V
  using tr-S U S V tr-U  $\langle$ trail U  $\neq []$  $\rangle$  by (cases trail U) (auto simp: lits-of-def)
have DD'[simp]: D' = D
  using U S' S by auto
have [simp]: L' = -L
  proof (rule ccontr)
    assume  $\neg$  ?thesis
    then have -L  $\in \#$  remove1-mset L' D'
      using DD' LD' LD by (simp add: in-remove1-mset-neq)
    moreover
      have M': trail S = M0 @ Propagated L E  $\#$  trail V
        using tr-S unfolding U by auto
      have no-dup (trail S)
        using inv U unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def by auto
      then have atm-L-notin-M: atm-of L  $\notin$  atm-of ' (lits-of-l (trail V))
        using M' U S by (auto simp: lits-of-def)
      have get-lev-L:
        get-level(Propagated L E  $\#$  trail V) L = backtrack-lvl V
        using inv-V unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def by auto
      have atm-of L  $\notin$  atm-of ' (lits-of-l (rev M0))
        using  $\langle$ no-dup (trail S) $\rangle$  M' by (auto simp: lits-of-def)
      then have get-level (trail S) L = backtrack-lvl S
        using get-lev-L S unfolding M' by auto
    ultimately

```

```

    have get-maximum-level (trail S) (remove1-mset L' D') ≥ backtrack-lvl S
      by (metis get-maximum-level-ge-get-level get-level-uminus)
  then show False
    using ⟨i' < backtrack-lvl S⟩ i' by auto
qed
have cdclW-restart** S U
  using bj cdclW-bj.skip local.skip mono-rtrancpl[of skip cdclW-restart S U] other by meson
then have cdclW-restart-all-struct-inv U
  using inv rtrancpl-cdclW-restart-all-struct-inv-inv by blast
then have Propagated L E # trail V ⊨as CNot D'
  using U confl-U unfolding cdclW-restart-all-struct-inv-def cdclW-restart-conflicting-def by auto
then have ∀ L' ∈ # (remove1-mset L' D') .
  atm-of L' ∈ atm-of ' lits-of-l (Propagated L E # trail U)
  using U atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set in-CNot-implies-uminus(2)
  by (fastforce dest: in-diffD)
then have ∀ L' ∈ # (remove1-mset L' D') .
  atm-of L' ∉ atm-of ' lits-of-l M0
  using ⟨no-dup (trail S)⟩ unfolding tr-S U by (fastforce simp: lits-of-def image-image)
then have get-maximum-level (trail S) (remove1-mset L' D') = backtrack-lvl S
  using get-maximum-level-skip-un-decided-not-present[of remove1-mset L' D'
    M0 trail U] tr-S nm U
  ⟨get-maximum-level (trail U) ((remove1-mset (− L) D)) = backtrack-lvl U⟩
  by (auto simp: S)
then show False
  using i' ⟨i' < backtrack-lvl S⟩ by auto
qed

```

lemma *if-can-apply-resolve-no-more-backtrack:*

```

assumes
  skip: skip** S U and
  resolve: resolve S T and
  inv: cdclW-restart-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-restart-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 skip-or-resolve.simps)

```

lemma *cdcl_W-bj-bj-decomp:*

```

assumes cdclW-bj** S W and cdclW-restart-all-struct-inv S
shows
  (∃ T U V. (λS T. skip-or-resolve S T ∧ no-step backtrack S)** S T
    ∧ (λT U. resolve T U ∧ no-step backtrack T) T U
    ∧ skip** U V ∧ backtrack V W)
  ∨ (∃ T U. (λS T. skip-or-resolve S T ∧ no-step backtrack S)** S T
    ∧ (λT U. resolve T U ∧ no-step backtrack T) T U ∧ skip** U W)
  ∨ (∃ T. skip** S T ∧ backtrack T W)
  ∨ skip** S W (is ?RB S W ∨ ?R S W ∨ ?SB S W ∨ ?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-cdclW-bj
      cdclW-restart-all-struct-inv-def cdclW-restart-all-struct-inv-inv cdclW-o.bj local.bj other
      resolve rtranclp-cdclW-restart-all-struct-inv-inv rtranclp-skip-backtrack-backtrack
      rtranclp-skip-or-resolve-rtranclp-cdclW-restart step.prem)
next
case 2
then show ?case by (meson assms(2) cdclW-restart-all-struct-inv-def backtrack-no-cdclW-bj
    local.bj rtranclp-skip-backtrack-backtrack)
qed
then have IH: ?R S W  $\vee$  ?S S W using IH by blast

have cdclW-restart** S W using mono-rtranclp[of cdclW-bj cdclW-restart] st by blast
then have inv-W: cdclW-restart-all-struct-inv W by (simp add: rtranclp-cdclW-restart-all-struct-inv-inv
    step.prem)
consider
  (BT) X' where backtrack W X'
  | (skip) no-step backtrack W and skip W X
  | (resolve) no-step backtrack W and resolve W X
using bj cdclW-bj.cases by meson
then show ?case
proof cases
  case (BT X')
  then consider
    (bt) backtrack W X
    | (sk) skip W X
  using bj if-can-apply-backtrack-no-more-resolve[of W W X' X] inv-W cdclW-bj.cases by fast
then show ?thesis
proof cases
  case bt
  then show ?thesis using IH by auto
next
  case sk
  then show ?thesis using IH by (meson rtranclp-trans r-into-rtranclp)
qed
next
case skip
then show ?thesis using IH by (meson rtranclp.rtrancl-into-rtrancl)
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-restart** S T
  using RS(1) cdclW-bj.resolve cdclW-o.bj other skip
  mono-rtrancpl[of (λS T. skip-or-resolve S T ∧ no-step backtrack S) cdclW-restart S T]
  by (meson skip-or-resolve.cases)
  then have cdclW-restart-all-struct-inv U
  by (meson RS(2) cdclW-restart-all-struct-inv-inv cdclW-bj.resolve cdclW-o.bj other
    rtrancpl-cdclW-restart-all-struct-inv-inv step.prem)
  { fix U'
    assume skip** U U' and skip** U' W
    have cdclW-restart-all-struct-inv U'
    using ⟨cdclW-restart-all-struct-inv U⟩ ⟨skip** U U'⟩ rtrancpl-cdclW-restart-all-struct-inv-inv
      cdclW-o.bj rtrancpl-mono[of skip cdclW-restart] other skip by blast
    then have no-step backtrack U'
    using if-can-apply-backtrack-no-more-resolve[OF ⟨skip** U' W⟩] res by blast
  }
  with ⟨skip** U W⟩
  have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** U W
  proof induction
    case base
    then show ?case by simp
  next
  case (step V W) note st = this(1) and skip = this(2) and IH = this(3) and H = this(4)
  have ∧ U'. skip** U' V ⇒ skip** U' W
  using skip by auto
  then have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** U V
  using IH H by blast
  moreover have (λS T. skip-or-resolve S T ∧ no-step backtrack S)** V W
  by (simp add: local.skip r-into-rtrancpl st step.prem skip-or-resolve.intros)
  ultimately show ?case by simp
qed
then show ?thesis
proof -
  have f1: ∀ p pa pb pc. ¬ p (pa) pb ∨ ¬ p** pb pc ∨ p** pa pc
  by (meson converse-rtrancpl-into-rtrancpl)
  have skip-or-resolve T U ∧ no-step backtrack T
  using RS(2) RS(3) by force
  then have (λp pa. skip-or-resolve p pa ∧ no-step backtrack p)** T W
  proof -
    have (∃ vr19 vr16 vr17 vr18. vr19 (vr16::'st) vr17 ∧ vr19** vr17 vr18
      ∧ ¬ vr19** vr16 vr18)
      ∨ ¬ (skip-or-resolve T U ∧ no-step backtrack T)
      ∨ ¬ (λuu uua. skip-or-resolve uu uua ∧ no-step backtrack uu)** U W
      ∨ (λuu uua. skip-or-resolve uu uua ∧ no-step backtrack uu)** T W
    by force
    then show ?thesis
    by (metis (no-types) ⟨(λS T. skip-or-resolve S T ∧ no-step backtrack S)** U W⟩
      ⟨skip-or-resolve T U ∧ no-step backtrack T⟩ f1)
  qed
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-restart** S U'
    using mono-rtrancpl[of skip cdclW-restart S U'] by (simp add: cdclW-o.bj other skip)
  then have cdclW-restart-all-struct-inv U'
    by (metis (no-types, hide-lams)  $\langle$ cdclW-restart-all-struct-inv S $\rangle$ 
      rtrancpl-cdclW-restart-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 skip-or-resolve.intros)
    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-restart-all-struct-inv S
  shows  $T \sim V \vee \text{cdcl}_W\text{-bj}^{**} T V$ 
  using assms(2)
proof induction
  case base
  then show ?case by (simp add: assms(1))
next
case (step T U) note st = this(1) and s-o-r = this(2) and IH = this(3)
have cdclW-restart** S T
  using st mono-rtrancpl[of cdclW-bj cdclW-restart] other by blast
then have lev-T: cdclW-restart-M-level-inv T
  using inv rtrancpl-cdclW-restart-consistent-inv[of S T]

```

unfolding *cdcl_W-restart-all-struct-inv-def* **by** *auto*

consider

(*TV*) $T \sim V$
 | (*bj-TV*) *cdcl_W-bj*** $T V$

using *IH* **by** *blast*

then show *?case*

proof *cases*

case *TV*

have *no-step cdcl_W-bj* T

using $\langle \text{cdcl}_W\text{-restart-M-level-inv } T \rangle$ *n-s cdcl_W-bj-state-eq-compatible*[*of T - V*] *TV*
by (*meson backtrack-state-eq-compatible cdcl_W-bj.simps resolve-state-eq-compatible skip-state-eq-compatible state-eq-ref*)

then show *?thesis*

using *s-o-r* **by** *auto*

next

case *bj-TV*

then obtain U' **where**

$T-U'$: *cdcl_W-bj* $T U'$ **and**
*cdcl_W-bj*** $U' V$

using *IH n-s s-o-r* **by** (*metis rtranclp-unfold tranclpD*)

have *cdcl_W-restart*** $S T$

by (*metis (no-types, hide-lams) bj mono-rtranclp[of cdcl_W-bj cdcl_W-restart] other st*)

then have *inv-T*: *cdcl_W-restart-all-struct-inv* T

by (*metis (no-types, hide-lams) inv rtranclp-cdcl_W-restart-all-struct-inv-inv*)

have *lev-U*: *cdcl_W-restart-M-level-inv* U

using *s-o-r cdcl_W-restart-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*]
cdcl_W-bj-decomp-resolve-skip-and-bj

by (*meson bj-TV cdcl_W-bj.backtrack inv-T lev-T n-s rtranclp-skip-backtrack-backtrack-end*)

then have *cdcl_W-bj*** $T V0$ **and** *cdcl_W-bj* $V0 V$

using *rtranclp-mono*[*of skip cdcl_W-bj*] **by** *blast+*

then show *?thesis*

using $\langle \text{backtrack } V0 V \rangle$ $\langle \text{skip** } T V0 \rangle$ *backtrack-unique inv-T local.backtrack rtranclp-skip-backtrack-backtrack* **by** *auto*

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 rtranclp.rtrancl-refl*)

then show *?thesis*

using $\langle \text{cdcl}_W\text{-bj** } U' V \rangle$ **unfolding** *rtranclp-unfold*

by (*meson T-U' bj cdcl_W-restart-consistent-inv lev-T other state-eq-ref state-eq-sym tranclp-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  $cdcl_W$ -restart-all-inv(3)  $cdcl_W$ -restart-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
    have  $cdcl_W$ -bj  $U U'$ 
      by (meson  $\langle skip^{++} T U \rangle$  backtrack  $bt$  inv- $T$   $rtrancpl$ -skip-backtrack-backtrack-end
       $trancpl$ -into- $rtrancpl$ )
    then have  $cdcl_W$ -bj**  $U V$ 
      using  $\langle cdcl_W$ -bj**  $U' V \rangle$  by auto
    then show ?thesis
      by (meson  $trancpl$ -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:  $cdcl_W$ -restart-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:  $cdcl_W$ -restart-all-struct-inv  $S$ 
  shows  $T \sim U$ 
    using  $cdcl_W$ -bj-unique-normal-form assms unfolding  $full$ -def by blast

```

6.2.3 CDCL with Merging

inductive $cdcl_W$ -merge-restart :: ' $st \Rightarrow 'st \Rightarrow bool$ **where**

fw - r -propagate: $propagate S S' \Longrightarrow cdcl_W$ -merge-restart $S S' \mid$

fw - r -conflict: $conflict S T \Longrightarrow full\ cdcl_W$ -bj $T U \Longrightarrow cdcl_W$ -merge-restart $S U \mid$

fw - r -decide: $decide S S' \Longrightarrow cdcl_W$ -merge-restart $S S' \mid$

fw - r -rf: $cdcl_W$ -rf $S S' \Longrightarrow cdcl_W$ -merge-restart $S S'$

lemma $rtrancpl$ - $cdcl_W$ -bj- $rtrancpl$ - $cdcl_W$ -restart:

$cdcl_W$ -bj** $S T \Longrightarrow cdcl_W$ -restart** $S T$

```

using mono-rtrancpl[of cdclW-bj cdclW-restart] by blast

lemma cdclW-merge-restart-cdclW-restart:
  assumes cdclW-merge-restart S T
  shows cdclW-restart** S T
  using assms
proof induction
  case (fw-r-conflict S T U) note confl = this(1) and bj = this(2)
  have cdclW-restart S T using confl by (simp add: cdclW-restart.intros r-into-rtrancpl)
  moreover
    have cdclW-bj** T U using bj unfolding full-def by auto
    then have cdclW-restart** T U using rtrancpl-cdclW-bj-rtrancpl-cdclW-restart by blast
  ultimately show ?case by auto
qed (simp-all add: cdclW-o.intros cdclW-restart.intros r-into-rtrancpl)

lemma cdclW-merge-restart-conflicting-true-or-no-step:
  assumes cdclW-merge-restart S T
  shows conflicting T = None  $\vee$  no-step cdclW-restart T
  using assms
proof induction
  case (fw-r-conflict S T U) note confl = this(1) and n-s = this(2)
  { fix D V
    assume cdclW-restart U V and conflicting U = Some D
    then have False
      using n-s unfolding full-def
      by (induction rule: cdclW-restart-all-rules-induct)
        (auto dest!: cdclW-bj.intros elim: decideE propagateE conflictE forgetE restartE)
  }
  then show ?case by (cases conflicting U) fastforce+
qed (auto simp add: cdclW-rf.simps elim: propagateE decideE restartE forgetE)

inductive cdclW-merge :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  fw-propagate: propagate S S'  $\Longrightarrow$  cdclW-merge S S' |
  fw-conflict: conflict S T  $\Longrightarrow$  full cdclW-bj T U  $\Longrightarrow$  cdclW-merge S U |
  fw-decide: decide S S'  $\Longrightarrow$  cdclW-merge S S' |
  fw-forget: forget S S'  $\Longrightarrow$  cdclW-merge S S'

lemma cdclW-merge-cdclW-merge-restart:
  cdclW-merge S T  $\Longrightarrow$  cdclW-merge-restart S T
  by (meson cdclW-merge.cases cdclW-merge-restart.simps forget)

lemma rtrancpl-cdclW-merge-trancpl-cdclW-merge-restart:
  cdclW-merge** S T  $\Longrightarrow$  cdclW-merge-restart** S T
  using rtrancpl-mono[of cdclW-merge cdclW-merge-restart] cdclW-merge-cdclW-merge-restart by blast

lemma cdclW-merge-rtrancpl-cdclW-restart:
  cdclW-merge S T  $\Longrightarrow$  cdclW-restart** S T
  using cdclW-merge-cdclW-merge-restart cdclW-merge-restart-cdclW-restart by blast

lemma rtrancpl-cdclW-merge-rtrancpl-cdclW-restart:
  cdclW-merge** S T  $\Longrightarrow$  cdclW-restart** S T
  using rtrancpl-mono[of cdclW-merge cdclW-restart**] cdclW-merge-rtrancpl-cdclW-restart by auto

lemmas rulesE =
  skipE resolveE backtrackE propagateE conflictE decideE restartE forgetE

```


lemma *cdcl_W-restart-all-struct-inv-tranclp-cdcl_W-merge-tranclp-cdcl_W-merge-cdcl_W-restart-all-struct-inv:*

assumes

inv: *cdcl_W-restart-all-struct-inv* *b*

cdcl_W-merge⁺⁺ *b a*

shows $(\lambda S T. \text{cdcl}_W\text{-restart-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T)^{++} b a$

using *assms*(2)

proof *induction*

case *base*

then show ?*case* **using** *inv* **by** *auto*

next

case (*step c d*) **note** *st* = *this*(1) **and** *fw* = *this*(2) **and** *IH* = *this*(3)

have *cdcl_W-restart-all-struct-inv* *c*

using *tranclp-into-rtranclp*[*OF st*] *cdcl_W-merge-rtranclp-cdcl_W-restart*

assms(1) *rtranclp-cdcl_W-restart-all-struct-inv-inv* *rtranclp-mono*[*of cdcl_W-merge cdcl_W-restart*^{**}] **by**

fastforce

then have $(\lambda S T. \text{cdcl}_W\text{-restart-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T)^{++} c d$

using *fw* **by** *auto*

then show ?*case* **using** *IH* **by** *auto*

qed

lemma *backtrack-is-full1-cdcl_W-bj:*

assumes *bt*: *backtrack* *S T* **and** *inv*: *cdcl_W-restart-M-level-inv* *S*

shows *full1 cdcl_W-bj* *S T*

using *bt inv backtrack-no-cdcl_W-bj unfolding full1-def* **by** *blast*

lemma *rtrancl-cdcl_W-restart-conflicting-true-cdcl_W-merge-restart:*

assumes *cdcl_W-restart*^{**} *S V* **and** *inv*: *cdcl_W-restart-M-level-inv* *S* **and** *conflicting* *S* = *None*

shows (*cdcl_W-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** *cdcl_W-restart* = *this*(2) **and** *IH* = *this*(3)[*OF this*(4-)] **and**

confl[*simp*] = *this*(5) **and** *inv* = *this*(4)

from *cdcl_W-restart*

show ?*case*

proof (*cases*)

case *propagate*

moreover then have *conflicting* *U* = *None* **and** *conflicting* *V* = *None*

by (*auto elim*: *propagateE*)

ultimately show ?*thesis* **using** *IH cdcl_W-merge-restart.fw-r-propagate*[*of U V*] **by** *auto*

next

case *conflict*

moreover then have *conflicting* *U* = *None* **and** *conflicting* *V* \neq *None*

by (*auto elim*!: *conflictE simp del*: *state-simp simp*: *state-eq-def*)

ultimately show ?*thesis* **using** *IH* **by** *auto*

next

case *other*

then show ?*thesis*

proof *cases*

case *decide*

then show ?*thesis* **using** *IH cdcl_W-merge-restart.fw-r-decide*[*of U V*] **by** (*auto elim*: *decideE*)

next

case *bj*

```

moreover {
  assume skip-or-resolve  $U\ V$ 
  have  $f1: \text{cdcl}_W\text{-bj}^{++}\ U\ V$ 
  by (simp add: local.bj tranclp.r-into-trancl)
  obtain  $T\ T' :: 'st$  where
     $f2: \text{cdcl}_W\text{-merge-restart}^{**}\ S\ U$ 
     $\vee \text{cdcl}_W\text{-merge-restart}^{**}\ S\ T \wedge \text{conflicting}\ U \neq \text{None}$ 
     $\wedge \text{conflict}\ T\ T' \wedge \text{cdcl}_W\text{-bj}^{**}\ T'\ U$ 
  using IH confl by blast
  have  $\text{conflicting}\ V \neq \text{None} \wedge \text{conflicting}\ U \neq \text{None}$ 
  using (skip-or-resolve  $U\ V$ )
  by (auto simp: skip-or-resolve.simps state-eq-def elim!: skipE resolveE
    simp del: state-simp)
  then have ?thesis
  by (metis (full-types) IH f1 rtranclp-trans tranclp-into-rtranclp)
}
moreover {
  assume backtrack  $U\ V$ 
  then have  $\text{conflicting}\ U \neq \text{None}$  by (auto elim: backtrackE)
  then obtain  $T\ T'$  where
     $\text{cdcl}_W\text{-merge-restart}^{**}\ S\ T$  and
     $\text{conflicting}\ U \neq \text{None}$  and
     $\text{conflict}\ T\ T'$  and
     $\text{cdcl}_W\text{-bj}^{**}\ T'\ U$ 
  using IH confl by meson
  have  $\text{inv}U: \text{cdcl}_W\text{-restart-M-level-inv}\ U$ 
  using inv rtranclp-cdclW-restart-consistent-inv step.hyps(1) by blast
  then have  $\text{conflicting}\ V = \text{None}$ 
  using (backtrack  $U\ V$ ) inv by (auto elim: backtrackE
    simp: cdclW-restart-M-level-inv-decomp)
  have full  $\text{cdcl}_W\text{-bj}\ T'\ V$ 
  apply (rule rtranclp-fullI[of cdclW-bj T' U V])
  using ( $\text{cdcl}_W\text{-bj}^{**}\ T'\ U$ ) apply fast
  using (backtrack  $U\ V$ ) backtrack-is-full1-cdclW-bj invU unfolding full1-def full-def
  by blast
  then have ?thesis
  using  $\text{cdcl}_W\text{-merge-restart.fw-r-conflict[of } T\ T'\ V]$  (conflict  $T\ T'$ )
  ( $\text{cdcl}_W\text{-merge-restart}^{**}\ S\ T$ ) ( $\text{conflicting}\ V = \text{None}$ ) by auto
}
ultimately show ?thesis by (auto simp: cdclW-bj.simps)
qed
next
case rf
moreover then have  $\text{conflicting}\ U = \text{None}$  and  $\text{conflicting}\ V = \text{None}$ 
by (auto simp: cdclW-rf.simps elim: restartE forgetE)
ultimately show ?thesis using IH cdclW-merge-restart.fw-r-rf[of U V] by auto
qed
qed

```

lemma *no-step-cdcl_W-restart-no-step-cdcl_W-merge-restart: no-step cdcl_W-restart $S \implies$ no-step cdcl_W-merge-restart S*
by (*auto simp: cdcl_W-restart.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-restart:*
assumes
conflicting $S = \text{None}$ **and**

```

    cdclW-restart-M-level-inv S and
    no-step cdclW-merge-restart S
  shows no-step cdclW-restart S
proof -
  { fix S'
    assume conflict S S'
    then have cdclW-restart S S' using cdclW-restart.conflict by auto
    then have cdclW-restart-M-level-inv S'
      using assms(2) cdclW-restart-consistent-inv by blast
    then obtain S'' where full cdclW-bj S' S''
      using cdclW-bj-exists-normal-form[of S'] by auto
    then have False
      using ⟨conflict S S'⟩ assms(3) fw-r-conflict by blast
  }
  then show ?thesis
    using assms unfolding cdclW-restart.simps cdclW-merge-restart.simps cdclW-o.simps cdclW-bj.simps
    by (auto elim: skipE resolveE backtrackE conflictE decideE restartE)
qed

```

```

lemma cdclW-merge-restart-no-step-cdclW-bj:
  assumes
    cdclW-merge-restart S T
  shows no-step cdclW-bj T
  using assms
  by (induction rule: cdclW-merge-restart.induct)
    (force simp: cdclW-bj.simps cdclW-rf.simps cdclW-merge-restart.simps full-def
      elim!: rulesE)+

```

```

lemma rtranclp-cdclW-merge-restart-no-step-cdclW-bj:
  assumes
    cdclW-merge-restart** S T and
    conflicting S = None
  shows no-step cdclW-bj T
  using assms unfolding rtranclp-unfold
  apply (elim disjE)
  apply (force simp: cdclW-bj.simps cdclW-rf.simps elim!: rulesE)
  by (auto simp: tranclp-unfold-end simp: cdclW-merge-restart-no-step-cdclW-bj)

```

If $\text{conflicting } S \neq \text{None}$, 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-cdclW-restart-iff-full-cdclW-merge:
  assumes confl: conflicting S = None and lev: cdclW-restart-M-level-inv S
  shows full cdclW-restart S V  $\longleftrightarrow$  full cdclW-merge-restart S V
proof
  assume full: full cdclW-merge-restart S V
  then have st: cdclW-restart** S V
    using rtranclp-mono[of cdclW-merge-restart cdclW-restart**] cdclW-merge-restart-cdclW-restart
    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 rtranclp-cdclW-merge-restart-no-step-cdclW-bj confl full unfolding full-def by auto
  have  $\bigwedge S'. \text{conflict } V S' \implies \text{cdcl}_W\text{-restart-M-level-inv } S'$ 

```

using $cdcl_W\text{-restart.conflict } cdcl_W\text{-restart-consistent-inv lev rtrancp-cdcl}_W\text{-restart-consistent-inv st}$
by *blast*
then have $\bigwedge S'. \text{ conflict } V S' \implies \text{False}$
using $n\text{-s } n\text{-s-bj } cdcl_W\text{-bj-exists-normal-form } cdcl_W\text{-merge-restart.simps}$ **by** *meson*
then have $n\text{-s-cdcl}_W\text{-restart: no-step } cdcl_W\text{-restart } V$
using $n\text{-s } n\text{-s-bj}$ **by** $(\text{auto simp: } cdcl_W\text{-restart.simps } cdcl_W\text{-o.simps } cdcl_W\text{-merge-restart.simps})$
then show $\text{full } cdcl_W\text{-restart } S V$ **using** *st* **unfolding** *full-def* **by** *auto*
next
assume *full: full cdcl_W-restart S V*
have $\text{no-step } cdcl_W\text{-merge-restart } V$
using *full no-step-cdcl_W-restart-no-step-cdcl_W-merge-restart* **unfolding** *full-def* **by** *blast*
moreover
consider
 $(fw) \text{ } cdcl_W\text{-merge-restart}^{**} S V \text{ and conflicting } V = \text{None}$
 $| (bj) T U \text{ where}$
 $cdcl_W\text{-merge-restart}^{**} S T \text{ and}$
 $\text{conflicting } V \neq \text{None} \text{ and}$
 $\text{conflict } T U \text{ and}$
 $cdcl_W\text{-bj}^{**} U V$
using *full rtranc-cdcl_W-restart-conflicting-true-cdcl_W-merge-restart confl lev* **unfolding** *full-def*
by *meson*
then have $cdcl_W\text{-merge-restart}^{**} S V$
proof *cases*
case *fw*
then show *?thesis* **by** *fast*
next
case $(bj) T U$
have $\text{no-step } cdcl_W\text{-bj } V$
using *full unfolding full-def* **by** $(\text{meson } cdcl_W\text{-o.bj other})$
then have $\text{full } cdcl_W\text{-bj } U V$
using $\langle cdcl_W\text{-bj}^{**} U V \rangle$ **unfolding** *full-def* **by** *auto*
then have $cdcl_W\text{-merge-restart } T V$
using $\langle \text{conflict } T U \rangle cdcl_W\text{-merge-restart.fw-r-conflict}$ **by** *blast*
then show *?thesis* **using** $\langle cdcl_W\text{-merge-restart}^{**} S T \rangle$ **by** *auto*
qed
ultimately show $\text{full } cdcl_W\text{-merge-restart } S V$ **unfolding** *full-def* **by** *fast*
qed

lemma *init-state-true-full-cdcl_W-restart-iff-full-cdcl_W-merge:*
shows $\text{full } cdcl_W\text{-restart } (\text{init-state } N) V \longleftrightarrow \text{full } cdcl_W\text{-merge-restart } (\text{init-state } N) V$
by $(\text{rule } \text{conflicting-true-full-cdcl}_W\text{-restart-iff-full-cdcl}_W\text{-merge}) \text{ auto}$

6.2.4 CDCL with Merge and Strategy

The intermediate step

inductive $cdcl_W\text{-s}' :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**
 $\text{conflict}': \text{full1 } cdcl_W\text{-cp } S S' \implies cdcl_W\text{-s}' S S' \mid$
 $\text{decide}': \text{decide } S S' \implies \text{no-step } cdcl_W\text{-cp } S \implies \text{full } cdcl_W\text{-cp } S' S'' \implies cdcl_W\text{-s}' S S'' \mid$
 $\text{bj}': \text{full1 } cdcl_W\text{-bj } S S' \implies \text{no-step } cdcl_W\text{-cp } S \implies \text{full } cdcl_W\text{-cp } S' S'' \implies cdcl_W\text{-s}' S S''$

inductive-cases $cdcl_W\text{-s}'E: cdcl_W\text{-s}' S T$

lemma *rtrancp-cdcl_W-bj-full1-cdclp-cdcl_W-stgy:*
 $cdcl_W\text{-bj}^{**} S S' \implies \text{full } cdcl_W\text{-cp } S' S'' \implies cdcl_W\text{-stgy}^{**} S S''$
proof $(\text{induction rule: converse-rtrancp-induct})$

```

case base
then show ?case by (metis cdclW-stgy.conflict' full-unfold rtrancpl.simps)
next
case (step T U) note st = this(2) and bj = this(1) and IH = this(3)[OF this(4)]
have no-step cdclW-cp T
  using bj by (auto simp add: cdclW-bj.simps cdclW-cp.simps elim!: rulesE)
consider
  (U) U = S'
| (U') U' where cdclW-bj U U' and cdclW-bj** U' S'
  using st by (metis converse-rtrancplE)
then show ?case
proof cases
case U
  then show ?thesis
    using (no-step cdclW-cp T) cdclW-o.bj local.bj other' step.prem by (meson r-into-rtrancpl)
next
case U' note U' = this(1)
have no-step cdclW-cp U
  using U' by (fastforce simp: cdclW-cp.simps cdclW-bj.simps elim: rulesE)
then have full cdclW-cp U U
  by (simp add: full-unfold)
then have cdclW-stgy T U
  using (no-step cdclW-cp T) cdclW-stgy.simps local.bj cdclW-o.bj by meson
then show ?thesis using IH by auto
qed
qed

```

```

lemma cdclW-s'-is-rtrancpl-cdclW-stgy:
  cdclW-s' S T  $\implies$  cdclW-stgy** S T
  apply (induction rule: cdclW-s'.induct)
  apply (auto intro: cdclW-stgy.intros)[]
  apply (meson decide other' r-into-rtrancpl)
  by (metis full1-def rtrancpl-cdclW-bj-full1-cdclp-cdclW-stgy trancpl-into-rtrancpl)

```

```

lemma cdclW-cp-cdclW-bj-bissimulation:

```

```

  assumes
    full cdclW-cp T U and
    cdclW-bj** T T' and
    cdclW-restart-all-struct-inv T and
    no-step cdclW-bj T'
  shows full cdclW-cp T' U
     $\vee (\exists U' U''. \text{full cdcl}_{\text{W}}\text{-cp } T' U'' \wedge \text{full1 cdcl}_{\text{W}}\text{-bj } U U' \wedge \text{full cdcl}_{\text{W}}\text{-cp } U' U''$ 
       $\wedge \text{cdcl}_{\text{W}}\text{-s}^{**} U U'')$ 
  using assms(2,1,3,4)
proof (induction rule: rtrancpl-induct)
case base
  then show ?case by blast
next
case (step T' T'') note st = this(1) and bj = this(2) and IH = this(3)[OF this(4,5)] and
  full = this(4) and inv = this(5)
have cdclW-bj** T T''
  using local.bj st by auto
then have cdclW-restart** T T''
  using rtrancpl-cdclW-bj-rtrancpl-cdclW-restart by blast
then have inv-T'': cdclW-restart-all-struct-inv T''
  using inv rtrancpl-cdclW-restart-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
  using ⟨cdclW-bj++ T T'⟩ by (blast dest: tranclpD)
  { assume cdclW-cp++ T U
    then obtain Z' where cdclW-cp T Z'
    by (meson tranclpD)
    then have False
    using ⟨cdclW-bj T Z⟩ by (fastforce simp: cdclW-bj.simps cdclW-cp.simps
      elim: rulesE)
  }
  then show ?thesis
    using full unfolding full-def rtranclp-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'⟩ rtranclp-cdclW-bj-full1-cdclp-cdclW-stgy rtranclp-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-rtranclp)
qed
ultimately show ?case
  using ⟨full1 cdclW-bj T T'⟩ ⟨full cdclW-cp T'' U''⟩ unfolding ⟨T = U⟩ by blast
qed

lemma cdclW-cp-cdclW-bj-bissimulation':
  assumes
    full cdclW-cp T U and
    cdclW-bj** T T' and
    cdclW-restart-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: rtranclp-induct)
  case base
  then show ?case by blast
next
  case (step T' T'') note st = this(1) and bj = this(2) and IH = this(3)[OF this(4,5)] and
    full = this(4) and inv = this(5)
  have cdclW-restart** T T''
  by (metis local.bj rtranclp.simps rtranclp-cdclW-bj-rtranclp-cdclW-restart st)
  then have inv-T'': cdclW-restart-all-struct-inv T''

```

```

    using inv rtrancp-cdclW-restart-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
    using ⟨cdclW-bj++ T T'⟩ by (blast dest: trancpD)
    { 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 elim: rulesE)
    }
    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 *cdcl_W-stgy-cdcl_W-s'-connected:*

```

assumes cdclW-stgy S U and cdclW-restart-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-restart-all-struct-inv T
  using cdclW-restart-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-restart-all-struct-inv-def
  by (metis full-unfold)
then show ?thesis
proof cases
  case cp
  then show ?thesis
  proof -
    obtain ss :: 'st ⇒ 'st where
      f1: ∀ s sa sb. (¬ full1 cdclW-bj s sa ∨ cdclW-cp s (ss s) ∨ ¬ full cdclW-cp sa sb)
      ∨ cdclW-s' s sb
    using bj' by moura
    have full1 cdclW-bj S T
    by (simp add: cp(2) full1-def local.bj tranclp.r-into-trancl)
    then show ?thesis
    using f1 full n-s by blast
  qed
next
case (fbj U')
then have full1 cdclW-bj S U'
  using bj unfolding full1-def by auto
moreover have no-step cdclW-cp S
  using n-s by blast
moreover have T = U
  using full fbj unfolding full1-def full-def rtranclp-unfold
  by (force dest!: tranclpD simp:cdclW-bj.simps elim: rulesE)
ultimately show ?thesis using cdclW-s'.bj'[of S U'] using fbj by blast
qed
qed
qed

lemma cdclW-stgy-cdclW-s'-connected':
  assumes cdclW-stgy S U and cdclW-restart-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  $cdcl_W-s'.simps$  full  $n-s$  by blast
  next
    case bj
      have  $cdcl_W$ -restart-all-struct-inv  $T$ 
        using  $cdcl_W$ -restart-all-struct-inv-inv  $o$  other other'.prems by blast
      then obtain  $T'$  where  $T'$ : full  $cdcl_W$ -bj  $T$   $T'$ 
        using  $cdcl_W$ -bj-exists-normal-form unfolding full-def  $cdcl_W$ -restart-all-struct-inv-def by metis
      then have full  $cdcl_W$ -bj  $S$   $T'$ 
        proof -
          have  $f1$ :  $cdcl_W$ -bj**  $T$   $T' \wedge$  no-step  $cdcl_W$ -bj  $T'$ 
            by (metis (no-types)  $T'$  full-def)
          then have  $cdcl_W$ -bj**  $S$   $T'$ 
            by (meson converse-rtranclp-into-rtranclp local.bj)
          then show ?thesis
            using  $f1$  by (simp add: full-def)
        qed
      have  $cdcl_W$ -bj**  $T$   $T'$ 
        using  $T'$  unfolding full-def by simp
      have  $cdcl_W$ -restart-all-struct-inv  $T$ 
        using  $cdcl_W$ -restart-all-struct-inv-inv  $o$  other other'.prems by blast
      then consider
        ( $T'U$ ) full  $cdcl_W$ -cp  $T'$   $U$ 
        | ( $U$ )  $U'$   $U''$  where
          full  $cdcl_W$ -cp  $T'$   $U''$  and
          full1  $cdcl_W$ -bj  $U$   $U'$  and
          full  $cdcl_W$ -cp  $U'$   $U''$  and
           $cdcl_W$ -s**  $U$   $U''$ 
        using  $cdcl_W$ -cp- $cdcl_W$ -bj-bissimulation[OF full  $\langle cdcl_W$ -bj**  $T$   $T' \rangle$ ]  $T'$  unfolding full-def
        by blast
      then show ?thesis by (metis  $T'$   $cdcl_W$ -s'.simps full-full1 local.bj  $n-s$ )
    qed
  qed

```

lemma $cdcl_W$ -stgy- $cdcl_W$ -s'-no-step:

```

  assumes  $cdcl_W$ -stgy  $S$   $U$  and  $cdcl_W$ -restart-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:  $cdcl_W$ -restart-M-level-inv  $S$ 
  shows  $cdcl_W$ -s**  $S$   $U \vee (\exists T. cdcl_W$ -s**  $S$   $T \wedge cdcl_W$ -bj**  $T$   $U \wedge$  conflicting  $U \neq$  None)
  using assms(1)
  proof induction
    case base
      then show ?case by simp
  next
    case (step  $T$   $V$ ) note  $st = this(1)$  and  $o = this(2)$  and  $IH = this(3)$ 
    from  $o$  show ?case
      proof cases
        case conflict'
          then have  $f2$ :  $cdcl_W$ -s'  $T$   $V$ 

```

```

    using  $cdcl_W-s'.conflict'$  by blast
  obtain  $ss :: 'st$  where
     $f3: S = T \vee cdcl_W-stgy^{**} S ss \wedge cdcl_W-stgy ss T$ 
    by (metis (full-types) rtranclp.simps st)
  obtain  $ssa :: 'st$  where
     $ssa: cdcl_W-cp T ssa$ 
    using  $conflict'$  by (metis (no-types) full1-def tranclpD)
  have  $\forall s. \neg full\ cdcl_W-cp\ s\ T$ 
    by (meson ssa full-def)
  then have  $S = T$ 
    by (metis (full-types) f3 ssa  $cdcl_W-stgy.cases$  full1-def)
  then show ?thesis
    using f2 by blast
next
case (other' U) note  $o = this(1)$  and  $n-s = this(2)$  and  $full = this(3)$ 
then show ?thesis
  using o
  proof (cases rule:  $cdcl_W-o-rule-cases$ )
    case decide
    then have  $cdcl_W-s'^{**} S T$ 
      using IH by (auto elim: rulesE)
    then show ?thesis
      by (meson decide decide' full n-s rtranclp.rtrancl-into-rtrancl)
  next
  case backtrack
  consider
    ( $s'$ )  $cdcl_W-s'^{**} S T$ 
    | ( $bj$ )  $S'$  where  $cdcl_W-s'^{**} S S'$  and  $cdcl_W-bj^{++} S' T$  and conflicting  $T \neq None$ 
    using IH by blast
  then show ?thesis
    proof cases
      case  $s'$ 
      moreover
      have  $cdcl_W-restart-M-level-inv T$ 
        using  $inv\ local.step(1)\ rtranclp-cdcl_W-stgy-consistent-inv$  by auto
      then have  $full1\ cdcl_W-bj\ T\ U$ 
        using  $backtrack-is-full1-cdcl_W-bj\ backtrack$  by blast
      then have  $cdcl_W-s' T V$ 
        using  $full\ bj'\ n-s$  by blast
      ultimately show ?thesis by auto
    next
    case ( $bj\ S'$ ) note  $S-S' = this(1)$  and  $bj-T = this(2)$ 
    have  $no-step\ cdcl_W-cp\ S'$ 
      using  $bj-T$  by (fastforce simp:  $cdcl_W-cp.simps\ cdcl_W-bj.simps\ dest!: tranclpD$ 
        elim: rulesE)
    moreover
    have  $cdcl_W-restart-M-level-inv T$ 
      using  $inv\ local.step(1)\ rtranclp-cdcl_W-stgy-consistent-inv$  by auto
    then have  $full1\ cdcl_W-bj\ T\ U$ 
      using  $backtrack-is-full1-cdcl_W-bj\ backtrack$  by blast
    then have  $full1\ cdcl_W-bj\ S'\ U$ 
      using  $bj-T\ unfolding\ full1-def$  by fastforce
    ultimately have  $cdcl_W-s'\ S'\ V$  using full by (simp add:  $bj'$ )
    then show ?thesis using  $S-S'$  by auto
  qed
next
qed
next

```

```

case skip
then have [simp]:  $U = V$ 
  using full converse-rtrancplE unfolding full-def by (fastforce elim: rulesE)
then have confl-V: conflicting  $V \neq \text{None}$ 
  using skip by (auto elim!: rulesE simp del: state-simp simp: state-eq-def)
consider
  (s')  $\text{cdcl}_W\text{-}s'^{**} S T$ 
  | (bj) S' where  $\text{cdcl}_W\text{-}s'^{**} S S'$  and  $\text{cdcl}_W\text{-}bj^{++} S' T$  and conflicting  $T \neq \text{None}$ 
  using IH by blast
then show ?thesis
proof cases
  case s'
  show ?thesis using s' confl-V skip by force
next
  case (bj S') note  $S\text{-}S' = \text{this}(1)$  and  $bj\text{-}T = \text{this}(2)$ 
  have  $\text{cdcl}_W\text{-}bj^{++} S' V$ 
  using skip bj-T by (metis (U = V) cdcl_W-bj.skip trancpl.simps)
  then show ?thesis using S-S' confl-V by auto
qed
next
case resolve
then have [simp]:  $U = V$ 
  using full unfolding full-def rtrancpl-unfold
  by (auto elim!: rulesE dest!: trancplD
    simp del: state-simp simp: state-eq-def cdcl_W-cp.simps)
have confl-V: conflicting  $V \neq \text{None}$ 
  using resolve by (auto elim!: rulesE simp del: state-simp simp: state-eq-def)

consider
  (s')  $\text{cdcl}_W\text{-}s'^{**} S T$ 
  | (bj) S' where  $\text{cdcl}_W\text{-}s'^{**} S S'$  and  $\text{cdcl}_W\text{-}bj^{++} S' T$  and conflicting  $T \neq \text{None}$ 
  using IH by blast
then show ?thesis
proof cases
  case s'
  have  $\text{cdcl}_W\text{-}bj^{++} T V$ 
  using resolve by force
  then show ?thesis using s' confl-V by auto
next
  case (bj S') note  $S\text{-}S' = \text{this}(1)$  and  $bj\text{-}T = \text{this}(2)$ 
  have  $\text{cdcl}_W\text{-}bj^{++} S' V$ 
  using resolve bj-T by (metis (U = V) cdcl_W-bj.resolve trancpl.simps)
  then show ?thesis using confl-V S-S' by auto
qed
qed
qed
qed

lemma n-step-cdcl_W-stgy-iff-no-step-cdcl_W-restart-cl-cdcl_W-o:
  assumes inv: cdcl_W-restart-all-struct-inv S
  shows no-step cdcl_W-s' S  $\longleftrightarrow$  no-step cdcl_W-cp S  $\wedge$  no-step cdcl_W-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: cdcl_W-s'.simps full1-def trancpl-unfold-begin)
next

```

```

assume  $n-s$ :  $?S' S$ 
have  $?C S$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $S'$  where  $cdcl_W\text{-}cp S S'$ 
    by auto
  then obtain  $T$  where  $full1\ cdcl_W\text{-}cp S T$ 
    using  $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv\ inv$  by (metis (no-types, lifting) full-unfold)
  then show  $False$  using  $n-s\ cdcl_W\text{-}s'.conflict'$  by blast
qed
moreover have  $?O S$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $S'$  where  $cdcl_W\text{-}o S S'$ 
    by auto
  then obtain  $T$  where  $full1\ cdcl_W\text{-}cp S' T$ 
    using  $cdcl_W\text{-}cp\text{-}normalized\text{-}element\text{-}all\text{-}inv\ inv$ 
    by (meson cdcl_W-restart-all-struct-inv-def n-s
       $cdcl_W\text{-}stgy\text{-}cdcl_W\text{-}s'\text{-}connected'\ cdcl_W\text{-}restart\text{-}then\text{-}exists\text{-}cdcl_W\text{-}stgy\text{-}step$  )
  then show  $False$  using  $n-s$  by (meson  $\langle cdcl_W\text{-}o S S' \rangle cdcl_W\text{-}restart\text{-}all\text{-}struct\text{-}inv\text{-}def$ 
       $cdcl_W\text{-}stgy\text{-}cdcl_W\text{-}s'\text{-}connected'\ cdcl_W\text{-}restart\text{-}then\text{-}exists\text{-}cdcl_W\text{-}stgy\text{-}step\ inv$ )
qed
ultimately show  $?C S \wedge ?O S$  by auto
qed

```

```

lemma  $cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W\text{-}restart$ :
   $cdcl_W\text{-}s' S S' \implies cdcl_W\text{-}restart^{++} S S'$ 
proof (induct rule: cdcl_W-s'.induct)
  case conflict'
  then show  $?case$ 
    by (simp add: full1-def trancp-cdcl_W-cp-trancp-cdcl_W-restart)
next
  case decide'
  then show  $?case$ 
    using  $cdcl_W\text{-}stgy.simps\ cdcl_W\text{-}stgy\text{-}trancp\text{-}cdcl_W\text{-}restart$  by (meson cdcl_W-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) trancpD)
  have  $cdcl_W\text{-}bj^{++}\ Sa\ S'a \wedge no\text{-}step\ cdcl_W\text{-}bj\ S'a$ 
    using  $a2$  by (simp add: full1-def)
  then have  $cdcl_W\text{-}bj\ Sa\ (ss\ S'a\ Sa\ cdcl_W\text{-}bj) \wedge cdcl_W\text{-}bj^{**}\ (ss\ S'a\ Sa\ cdcl_W\text{-}bj)\ S'a$ 
    using  $f3$  by auto
  then show  $cdcl_W\text{-}restart^{++}\ Sa\ S''$ 
    using  $a1\ n-s$  by (meson bj other rtrancp-cdcl_W-bj-full1-cdclp-cdcl_W-stgy
       $rtrancp\text{-}cdcl_W\text{-}stgy\text{-}rtrancp\text{-}cdcl_W\text{-}restart\ rtrancp\text{-}into\text{-}trancp2$ )
qed

```

```

lemma  $trancp\text{-}cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W\text{-}restart$ :
   $cdcl_W\text{-}s'^{++} S S' \implies cdcl_W\text{-}restart^{++} S S'$ 
apply (induct rule: trancp.induct)
  using  $cdcl_W\text{-}s'\text{-}trancp\text{-}cdcl_W\text{-}restart$  apply blast
by (meson cdcl_W-s'-trancp-cdcl_W-restart trancp-trans)

```

lemma *rtrancpl-cdcl_W-s'-rtrancpl-cdcl_W-restart*:
 $cdcl_W-s'^{**} S S' \implies cdcl_W-restart^{**} S S'$
using *rtrancpl-unfold*[of *cdcl_W-s' S S'*] *trancpl-cdcl_W-s'-trancpl-cdcl_W-restart*[of *S S'*] **by** *auto*

lemma *full-cdcl_W-stgy-iff-full-cdcl_W-s'*:
assumes *inv*: *cdcl_W-restart-all-struct-inv S*
shows *full cdcl_W-stgy S T \longleftrightarrow full cdcl_W-s' S T* (**is** *?S \longleftrightarrow ?S'*)

proof
assume *?S'*
then have *cdcl_W-restart^{**} S T*
using *rtrancpl-cdcl_W-s'-rtrancpl-cdcl_W-restart*[of *S T*] **unfolding** *full-def* **by** *blast*
then have *inv'*: *cdcl_W-restart-all-struct-inv T*
using *rtrancpl-cdcl_W-restart-all-struct-inv-inv inv* **by** *blast*
have *cdcl_W-stgy^{**} S T*
using $\langle ?S' \rangle$ **unfolding** *full-def*
using *cdcl_W-s'-is-rtrancpl-cdcl_W-stgy rtrancpl-mono*[of *cdcl_W-s' cdcl_W-stgy^{**}*] **by** *auto*
then show *?S*
using $\langle ?S' \rangle$ *inv' cdcl_W-stgy-cdcl_W-s'-connected'* **unfolding** *full-def* **by** *blast*

next
assume *?S*
then have *inv-T:cdcl_W-restart-all-struct-inv T*
by (*metis assms full-def rtrancpl-cdcl_W-restart-all-struct-inv-inv rtrancpl-cdcl_W-stgy-rtrancpl-cdcl_W-restart*)

consider
 $(s') cdcl_W-s'^{**} S T$
 $| (st) S' \text{ where } cdcl_W-s'^{**} S S' \text{ and } cdcl_W-bj^{++} S' T \text{ and } conflicting\ T \neq None$
using *rtrancpl-cdcl_W-stgy-connected-to-rtrancpl-cdcl_W-s'*[of *S T*] *inv $\langle ?S \rangle$*
unfolding *full-def cdcl_W-restart-all-struct-inv-def*
by *blast*

then show *?S'*
proof cases
case *s'*
have *no-step cdcl_W-s' T*
using $\langle full\ cdcl_W-stgy\ S\ T \rangle$ **unfolding** *full-def*
by (*meson cdcl_W-restart-all-struct-inv-def cdcl_W-s'E cdcl_W-stgy.conflict'*
cdcl_W-restart-then-exists-cdcl_W-stgy-step inv-T n-step-cdcl_W-stgy-iff-no-step-cdcl_W-restart-cl-cdcl_W-o)
then show *?thesis*
using *s'* **unfolding** *full-def* **by** *blast*

next
case $(st\ S')$
have *full cdcl_W-cp T T*
using *option-full-cdcl_W-cp st(3)* **by** *blast*

moreover
have *n-s: no-step cdcl_W-bj T*
by (*metis $\langle full\ cdcl_W-stgy\ S\ T \rangle$ bj inv-T cdcl_W-restart-all-struct-inv-def*
cdcl_W-restart-then-exists-cdcl_W-stgy-step full-def)
then have *full1 cdcl_W-bj S' T*
using *st(2)* **unfolding** *full1-def* **by** *blast*

moreover have *no-step cdcl_W-cp S'*
using *st(2)* **by** (*fastforce dest!: trancplD simp: cdcl_W-cp.simps cdcl_W-bj.simps*
elim: rulesE)

ultimately have *cdcl_W-s' S' T*
using *cdcl_W-s'.bj'*[of *S' T T*] **by** *blast*

then have *cdcl_W-s'^{**} S T*
using *st(1)* **by** *auto*

```

moreover have no-step cdclW-s' T
  using inv-T  $\langle \text{full } cdcl_W\text{-cp } T \ T \rangle$   $\langle \text{full } cdcl_W\text{-stgy } S \ T \rangle$  unfolding full-def
  by (metis cdclW-restart-all-struct-inv-def cdclW-restart-then-exists-cdclW-stgy-step
    n-step-cdclW-stgy-iff-no-step-cdclW-restart-cl-cdclW-o)
ultimately show ?thesis
  unfolding full-def by blast
qed
qed

lemma conflict-step-cdclW-stgy-step:
  assumes
    conflict S T
    cdclW-restart-all-struct-inv S
  shows  $\exists T. \text{cdcl}_W\text{-stgy } S \ T$ 
proof –
  obtain U where full cdclW-cp S U
    using cdclW-cp-normalized-element-all-inv assms by blast
  then have full1 cdclW-cp S U
    by (metis cdclW-cp.conflict' assms(1) full-unfold)
  then show ?thesis using cdclW-stgy.conflict' by blast
qed

lemma decide-step-cdclW-stgy-step:
  assumes
    decide S T
    cdclW-restart-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-restart-all-struct-inv-inv
      cdclW-cp-normalized-element-all-inv decide other)
  then show ?thesis
    by (metis assms cdclW-cp-normalized-element-all-inv cdclW-stgy.conflict' decide full-unfold
      other')
qed

lemma rtrancpl-cdclW-cp-conflicting-Some:
  cdclW-cp** S T  $\implies$  conflicting S = Some D  $\implies$  S = T
  using rtrancplD trancplD by fastforce

inductive cdclW-merge-cp :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where
  conflict': conflict S T  $\implies$  full cdclW-bj T U  $\implies$  cdclW-merge-cp S U |
  propagate': propagate++ S S'  $\implies$  cdclW-merge-cp S S'

lemma cdclW-merge-restart-cases[consumes 1, case-names conflict propagate]:
  assumes
    cdclW-merge-cp S U and
     $\bigwedge T. \text{conflict } S \ T \implies \text{full } cdcl_W\text{-bj } T \ U \implies P$  and
    propagate++ S U  $\implies$  P
  shows P
  using assms unfolding cdclW-merge-cp.simps by auto

lemma cdclW-merge-cp-trancpl-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 *trancpl-mono*[*of propagate cdcl_W-merge*] *fw-propagate* **by** *blast*

lemma *rtrancpl-cdcl_W-merge-cp-rtrancpl-cdcl_W-restart*:

*cdcl_W-merge-cp*** *S T* \implies *cdcl_W-restart*** *S T*

apply (*induction rule*: *rtrancpl-induct*)

apply *simp*

unfolding *cdcl_W-merge-cp.simps* **by** (*meson cdcl_W-merge-restart-cdcl_W-restart fw-r-conflict*
rtrancpl-propagate-is-rtrancpl-cdcl_W-restart rtrancpl-trans trancpl-into-rtrancpl)

lemma *full1-cdcl_W-bj-no-step-cdcl_W-bj*:

full1 cdcl_W-bj S T \implies *no-step cdcl_W-cp S*

unfolding *full1-def* **by** (*metis rtrancpl-unfold cdcl_W-cp-conflicting-not-empty option.exhaust*
rtrancpl-cdcl_W-merge-restart-no-step-cdcl_W-bj trancplD)

Full Transformation

inductive *cdcl_W-s'-without-decide* **where**

conflict'-without-decide[*intro*]: *full1 cdcl_W-cp S S'* \implies *cdcl_W-s'-without-decide S S'* |

bj'-without-decide[*intro*]: *full1 cdcl_W-bj S S'* \implies *no-step cdcl_W-cp S* \implies *full cdcl_W-cp S' S''*
 \implies *cdcl_W-s'-without-decide S S''*

lemma *rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W-restart*:

*cdcl_W-s'-without-decide*** *S T* \implies *cdcl_W-restart*** *S T*

apply (*induction rule*: *rtrancpl-induct*)

apply *simp*

by (*meson cdcl_W-s'.simps cdcl_W-s'-trancpl-cdcl_W-restart cdcl_W-s'-without-decide.simps*
rtrancpl-trancpl-trancpl trancpl-into-rtrancpl)

lemma *rtrancpl-cdcl_W-s'-without-decide-rtrancpl-cdcl_W-s'*:

*cdcl_W-s'-without-decide*** *S T* \implies *cdcl_W-s'*** *S T*

proof (*induction rule*: *rtrancpl-induct*)

case *base*

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

next

case (*step y z*) **note** *a2 = this(2)* **and** *a1 = this(3)*

have *cdcl_W-s' y z*

using *a2* **by** (*metis (no-types) bj' cdcl_W-s'.conflict' cdcl_W-s'-without-decide.cases*)

then show *cdcl_W-s'*** *S z*

using *a1* **by** (*meson r-into-rtrancpl rtrancpl-trans*)

qed

lemma *rtrancpl-cdcl_W-merge-cp-is-rtrancpl-cdcl_W-s'-without-decide*:

assumes

*cdcl_W-merge-cp*** *S V*

conflicting S = None

shows

(*cdcl_W-s'-without-decide*** *S V*)

$\vee (\exists T. \text{cdcl}_{W\text{-s'-without-decide}}^{**} S T \wedge \text{propagate}^{++} T V)$

$\vee (\exists T U. \text{cdcl}_{W\text{-s'-without-decide}}^{**} S T \wedge \text{full1 cdcl}_{W\text{-bj}} T U \wedge \text{propagate}^{**} U V)$

using *assms*

proof (*induction rule*: *rtrancpl-induct*)

case *base*

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

next

case (*step U V*) **note** *st = this(1)* **and** *cp = this(2)* **and** *IH = this(3)[OF this(4)]*

from *cp* **show** *?case*

```

proof (cases rule: cdclW-merge-restart-cases)
  case propagate
  then show ?thesis using IH by (meson rtrancpl-trancpl-trancpl trancpl-into-rtrancpl)
next
  case (conflict U') note confl = this(1) and bj = this(2)
  have full1-U-U': full1 cdclW-cp U U'
    by (simp add: conflict-is-full1-cdclW-cp local.conflict(1))
  consider
    (s') cdclW-s'-without-decide** S U
  | (propa) T' where cdclW-s'-without-decide** S T' and propagate++ T' U
  | (bj-prop) T' T'' where
    cdclW-s'-without-decide** S T' and
    full1 cdclW-bj T' T'' and
    propagate** T'' U
  using IH by blast
then show ?thesis
proof cases
  case s'
  have cdclW-s'-without-decide U U'
    using full1-U-U' conflict'-without-decide by blast
  then have cdclW-s'-without-decide** S U'
    using ⟨cdclW-s'-without-decide** S U⟩ by auto
  moreover have U' = V ∨ full1 cdclW-bj U' V
    using bj by (meson full-unfold)
  ultimately show ?thesis by blast
next
  case propa note s' = this(1) and T'-U = this(2)
  have full1 cdclW-cp T' U'
    using rtrancpl-mono[of propagate cdclW-cp] T'-U cdclW-cp.propagate' full1-U-U'
    rtrancpl-full1I[of cdclW-cp T'] by (metis (full-types) predicate2D predicate2I
      trancpl-into-rtrancpl)
  have cdclW-s'-without-decide** S U'
    using ⟨full1 cdclW-cp T' U'⟩ conflict'-without-decide s' by force
  have full1 cdclW-bj U' V ∨ V = U' using bj unfolding full-unfold by blast
  then show ?thesis
    using ⟨cdclW-s'-without-decide** S U'⟩ by blast
next
  case bj-prop note s' = this(1) and bj-T' = this(2) and T''-U = this(3)
  have no-step cdclW-cp T'
    using bj-T' full1-cdclW-bj-no-step-cdclW-bj by blast
  moreover have full1 cdclW-cp T'' U'
    using rtrancpl-mono[of propagate cdclW-cp] T''-U cdclW-cp.propagate' full1-U-U'
    rtrancpl-full1I[of cdclW-cp T''] by blast
  ultimately have cdclW-s'-without-decide T' U'
    using bj'-without-decide[of T' T'' U'] bj-T' by (simp add: full-unfold)
  then have cdclW-s'-without-decide** S U'
    using s' rtrancpl.intros(2)[of - S T' U'] by blast
  then show ?thesis
    using local.bj unfolding full-unfold by blast
qed
qed
qed

```

lemma rtrancpl-cdcl_W-s'-without-decide-is-rtrancpl-cdcl_W-merge-cp:
assumes
 cdcl_W-s'-without-decide** S V **and**


```

  confl: conflicting  $S = \text{None}$ 
shows
  ( $\text{cdcl}_W\text{-merge-cp}^{**} S V \wedge \text{conflicting } V = \text{None}$ )
 $\vee (\text{cdcl}_W\text{-merge-cp}^{**} S V \wedge \text{conflicting } V \neq \text{None} \wedge \text{no-step } \text{cdcl}_W\text{-cp } V \wedge \text{no-step } \text{cdcl}_W\text{-bj } V)$ 
 $\vee (\exists T. \text{cdcl}_W\text{-merge-cp}^{**} S T \wedge \text{conflict } T V)$ 
using assms(1)
proof (induction)
  case base
  then show ?case using confl by auto
next
case (step  $U V$ ) note st = this(1) and s = this(2) and IH = this(3)
from s show ?case
proof (cases rule: cdclW-s'-without-decide.cases)
  case conflict'-without-decide
  then have rt:  $\text{cdcl}_W\text{-cp}^{++} U V$  unfolding full1-def by fast
  then have conflicting  $U = \text{None}$ 
    using trancpl-cdclW-cp-propagate-with-conflict-or-not[of U V]
    conflict by (auto dest!: trancplD simp: rtrancpl-unfold elim: rulesE)
  then have  $\text{cdcl}_W\text{-merge-cp}^{**} S U$  using IH by (auto elim: rulesE
    simp del: state-simp simp: state-eq-def)
  consider
    (propa)  $\text{propagate}^{++} U V$ 
    | (confl')  $\text{conflict } U V$ 
    | (propa-confl')  $U' \text{ where } \text{propagate}^{++} U U' \text{ conflict } U' V$ 
  using trancpl-cdclW-cp-propagate-with-conflict-or-not[OF rt] unfolding rtrancpl-unfold
  by fastforce
  then show ?thesis
  proof cases
    case propa
    then have  $\text{cdcl}_W\text{-merge-cp } U V$ 
      by (auto intro: cdclW-merge-cp.intros)
    moreover have conflicting  $V = \text{None}$ 
      using propa unfolding trancpl-unfold-end by (auto elim: rulesE)
    ultimately show ?thesis using  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  by (auto elim!: rulesE
      simp del: state-simp simp: state-eq-def)
    next
    case confl'
    then show ?thesis using  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  by auto
    next
    case propa-confl' note propa = this(1) and confl' = this(2)
    then have  $\text{cdcl}_W\text{-merge-cp } U U'$  by (auto intro: cdclW-merge-cp.intros)
    then have  $\text{cdcl}_W\text{-merge-cp}^{**} S U'$  using  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  by auto
    then show ?thesis using  $\langle \text{cdcl}_W\text{-merge-cp}^{**} S U \rangle$  confl' by auto
  qed
next
case (bj'-without-decide U) note full-bj = this(1) and cp = this(3)
then have conflicting  $U \neq \text{None}$ 
  using full-bj unfolding full1-def by (fastforce dest!: trancplD simp: cdclW-bj.simps
    elim: rulesE)
with IH obtain T where
  S-T:  $\text{cdcl}_W\text{-merge-cp}^{**} S T$  and T-U:  $\text{conflict } T U$ 
  using full-bj unfolding full1-def by (blast dest: trancplD)
then have  $\text{cdcl}_W\text{-merge-cp } T U'$ 
  using  $\text{cdcl}_W\text{-merge-cp.conflict'}$ [of T U U'] full-bj by (simp add: full-unfold)
then have  $S\text{-}U'$ :  $\text{cdcl}_W\text{-merge-cp}^{**} S U'$  using S-T by auto
consider

```

```

    (n-s) U' = V
  | (propa) propagate++ U' V
  | (confl') conflict U' V
  | (propa-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 (blast intro: cdclW-merge-cp.intros)
    moreover have conflicting V = None
      using propa unfolding tranclp-unfold-end by (auto elim: rulesE)
    ultimately show ?thesis using S-U' by (auto elim: rulesE
      simp del: state-simp simp: state-eq-def)
  next
    case confl'
    then show ?thesis using S-U' by auto
  next
    case propa-confl' note propa = this(1) and confl = this(2)
    have cdclW-merge-cp U' U'' using propa by (blast intro: cdclW-merge-cp.intros)
    then show ?thesis using S-U' confl by (meson rtranclp.rtrancl-into-rtrancl)
  next
    case n-s
    then show ?thesis
      using S-U' apply (cases conflicting V = None)
      using full-bj apply simp
      by (metis cp full-def full-unfold full-bj)
qed
qed
qed

```

lemma no-step-cdcl_W-s'-no-ste-cdcl_W-merge-cp:

assumes

- cdcl_W-restart-all-struct-inv S
- conflicting S = None
- no-step cdcl_W-s' S

shows no-step cdcl_W-merge-cp S

using asms **apply** (auto simp: cdcl_W-s'.simps cdcl_W-merge-cp.simps)

using conflict-is-full1-cdcl_W-cp **apply** blast

using cdcl_W-cp-normalized-element-all-inv cdcl_W-cp.propagate' **by** (metis cdcl_W-cp.propagate' full-unfold tranclpD)

The no-step decide S is needed, since cdcl_W-merge-cp is cdcl_W-s' without decide.

lemma conflicting-true-no-step-cdcl_W-merge-cp-no-step-s'-without-decide:

assumes

- confl: conflicting S = None **and**
- inv: cdcl_W-restart-M-level-inv S **and**
- n-s: no-step cdcl_W-merge-cp S

shows no-step cdcl_W-s'-without-decide S

proof (rule ccontr)

assume ¬ no-step cdcl_W-s'-without-decide S

then obtain T **where**

- cdcl_W-restart: cdcl_W-s'-without-decide S T

by auto

then have inv-T: cdcl_W-restart-M-level-inv T

using rtranclp-cdcl_W-s'-without-decide-rtranclp-cdcl_W-restart[of S T]

```

  rtrancp-cdclW-restart-consistent-inv inv by blast
from cdclW-restart show False
proof cases
  case conflict'-without-decide
  have no-step propagate S
    using n-s by (blast intro: cdclW-merge-cp.intros)
  then have conflict S T
    using local.conflict' trancp-cdclW-cp-propagate-with-conflict-or-not[of S T]
    local.conflict'-without-decide unfolding full1-def rtrancp-unfold
    by (metis trancp-unfold-begin)
  moreover
    then obtain T' where full cdclW-bj T T'
    using cdclW-bj-exists-normal-form inv-T by blast
  ultimately show False using cdclW-merge-cp.conflict' n-s by meson
next
case (bj'-without-decide S')
then show ?thesis
  using confl unfolding full1-def by (fastforce simp: cdclW-bj.simps dest: trancpD
    elim: rulesE)
qed
qed

```

lemma *conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp:*

```

assumes
  inv: cdclW-restart-all-struct-inv S and
  n-s: no-step cdclW-s'-without-decide S
shows no-step cdclW-merge-cp S
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain T where cdclW-merge-cp S T
    by auto
  then show False
  proof cases
    case (conflict' S')
    then show False using n-s conflict'-without-decide conflict-is-full1-cdclW-cp by blast
  next
  case propagate'
  moreover
    have cdclW-restart-all-struct-inv T
      using inv by (meson local.propagate' rtrancp-cdclW-restart-all-struct-inv-inv
        rtrancp-propagate-is-rtrancp-cdclW-restart trancp-into-rtrancp)
    then obtain U where full cdclW-cp T U
      using cdclW-cp-normalized-element-all-inv by auto
    ultimately have full1 cdclW-cp S U
      using trancp-full-full1I[of cdclW-cp S T U] cdclW-cp.propagate'
      trancp-mono[of propagate cdclW-cp] by blast
    then show False using conflict'-without-decide n-s by blast
  qed
qed

```

lemma *no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp:*

```

no-step cdclW-merge-cp S ⇒ cdclW-restart-M-level-inv S ⇒ no-step cdclW-cp S
using cdclW-bj-exists-normal-form cdclW-restart-consistent-inv[OF cdclW-restart.conflict, of S]
by (metis cdclW-cp.cases cdclW-merge-cp.simps trancp.intros(1))

```

lemma *conflicting-not-true-rtrancp-cdcl_W-merge-cp-no-step-cdcl_W-bj:*

assumes
conflicting $S = \text{None}$ **and**
cdcl_W-merge-cp^{**} $S\ T$
shows *no-step cdcl_W-bj* T
using *assms*(2,1) **by** (*induction*)
(fastforce simp: cdcl_W-merge-cp.simps full-def tranclp-unfold-end cdcl_W-bj.simps
elim: rulesE)+

lemma *conflicting-true-full-cdcl_W-merge-cp-iff-full-cdcl_W-s'-without-decode:*

assumes
confl: *conflicting* $S = \text{None}$ **and**
inv: *cdcl_W-restart-all-struct-inv* S
shows
full cdcl_W-merge-cp $S\ V \longleftrightarrow \text{full cdcl_W-s'-without-decode } S\ V$ (**is** $?fw \longleftrightarrow ?s'$)

proof

assume $?fw$
then have *st*: *cdcl_W-merge-cp*^{**} $S\ V$ **and** *n-s*: *no-step cdcl_W-merge-cp* V
unfolding *full-def* **by** *blast*+
have *inv-V*: *cdcl_W-restart-all-struct-inv* V
using *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W-restart*[*of* $S\ V$] ($?fw$) **unfolding** *full-def*
by (*simp add: inv rtranclp-cdcl_W-restart-all-struct-inv-inv*)
consider
 (s') *cdcl_W-s'-without-decode*^{**} $S\ V$
 | (*propa*) T **where** *cdcl_W-s'-without-decode*^{**} $S\ T$ **and** *propagate*⁺⁺ $T\ V$
 | (*bj*) $T\ U$ **where** *cdcl_W-s'-without-decode*^{**} $S\ T$ **and** *full1 cdcl_W-bj* $T\ U$ **and** *propagate*^{**} $U\ V$
using *rtranclp-cdcl_W-merge-cp-is-rtranclp-cdcl_W-s'-without-decode confl st n-s* **by** *metis*
then have *cdcl_W-s'-without-decode*^{**} $S\ V$
proof cases
 case s'
 then show $?thesis$.
next
 case *propa* **note** $s' = \text{this}(1)$ **and** *propa* = *this*(2)
 have *no-step cdcl_W-cp* V
 using *no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp n-s inv-V*
 unfolding *cdcl_W-restart-all-struct-inv-def* **by** *blast*
 then have *full1 cdcl_W-cp* $T\ V$
 using *propa tranclp-mono*[*of propagate cdcl_W-cp*] *cdcl_W-cp.propagate'* **unfolding** *full1-def*
 by *blast*
 then have *cdcl_W-s'-without-decode* $T\ V$
 using *conflict'-without-decode* **by** *blast*
 then show $?thesis$ **using** s' **by** *auto*
next
 case *bj* **note** $s' = \text{this}(1)$ **and** *bj* = *this*(2) **and** *propa* = *this*(3)
 have *no-step cdcl_W-cp* V
 using *no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp n-s inv-V*
 unfolding *cdcl_W-restart-all-struct-inv-def* **by** *blast*
 then have *full cdcl_W-cp* $U\ V$
 using *propa rtranclp-mono*[*of propagate cdcl_W-cp*] *cdcl_W-cp.propagate'* **unfolding** *full-def*
 by *blast*
 moreover have *no-step cdcl_W-cp* T
 using *bj unfolding full1-def* **by** (*fastforce dest!: tranclpD simp:cdcl_W-bj.simps elim: rulesE*)
 ultimately have *cdcl_W-s'-without-decode* $T\ V$
 using *bj'-without-decode*[*of T U V*] *bj* **by** *blast*
 then show $?thesis$ **using** s' **by** *auto*
qed
moreover have *no-step cdcl_W-s'-without-decode* V

```

proof (cases conflicting V = None)
  case False
  { fix ss :: 'st
    have ff1:  $\forall s \text{ sa. } \neg \text{cdcl}_W\text{-s}' s \text{ sa} \vee \text{full1 } \text{cdcl}_W\text{-cp } s \text{ sa}$ 
       $\vee (\exists sb. \text{decide } s \text{ sb} \wedge \text{no-step } \text{cdcl}_W\text{-cp } s \wedge \text{full } \text{cdcl}_W\text{-cp } sb \text{ sa})$ 
       $\vee (\exists sb. \text{full1 } \text{cdcl}_W\text{-bj } s \text{ sb} \wedge \text{no-step } \text{cdcl}_W\text{-cp } s \wedge \text{full } \text{cdcl}_W\text{-cp } sb \text{ sa})$ 
      by (metis cdclW-s'.cases)
    have ff2:  $(\forall p \text{ s sa. } \neg \text{full1 } p (s::'st) \text{ sa} \vee p^{++} s \text{ sa} \wedge \text{no-step } p \text{ sa})$ 
       $\wedge (\forall p \text{ s sa. } (\neg p^{++} (s::'st) \text{ sa} \vee (\exists s. p \text{ sa } s)) \vee \text{full1 } p \text{ s sa})$ 
      by (meson full1-def)
    obtain ssa :: ('st  $\Rightarrow$  'st  $\Rightarrow$  bool)  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  'st where
      ff3:  $\forall p \text{ s sa. } \neg p^{++} s \text{ sa} \vee p \text{ s } (ssa \text{ p } s \text{ sa}) \wedge p^{**} (ssa \text{ p } s \text{ sa}) \text{ sa}$ 
      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  $\neg \text{cdcl}_W\text{-s}'\text{-without-decide } V \text{ ss}$ 
      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-restart-all-struct-inv-def by simp
  qed
ultimately show ?s' unfolding full-def by blast
next
assume s': ?s'
then have st: cdclW-s'-without-decide** S V and n-s: no-step cdclW-s'-without-decide V
  unfolding full-def by auto
then have cdclW-restart** S V
  using rtranclp-cdclW-s'-without-decide-rtranclp-cdclW-restart st by blast
then have inv-V: cdclW-restart-all-struct-inv V using inv rtranclp-cdclW-restart-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-restart-all-struct-inv-def by presburger
have n-s-bj: no-step cdclW-bj V
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain W where W: cdclW-bj V W by blast
  have cdclW-restart-all-struct-inv W
    using W cdclW-restart.simps cdclW-restart-all-struct-inv-inv inv-V by blast
  then obtain W' where full1 cdclW-bj V W'
    using cdclW-bj-exists-normal-form[of W] full-fullI[of cdclW-bj V W] W
    unfolding cdclW-restart-all-struct-inv-def
    by blast
  moreover
    then have cdclW-restart++ V W'
    using tranclp-mono[of cdclW-bj cdclW-restart] cdclW-restart.other cdclW-o.bj unfolding full1-def

```

by *blast*
 then have *cdcl_W-restart-all-struct-inv W'*
 by (*meson inv-V rtrancp-cdcl_W-restart-all-struct-inv-inv trancp-into-rtrancp*)
 then obtain *X* **where** *full cdcl_W-cp W' X*
 using *cdcl_W-cp-normalized-element-all-inv* **by** *blast*
 ultimately show *False*
 using *bj'-without-decide n-s-cp-V n-s* **by** *blast*
qed
from *s'* **consider**
 (*cp-true*) *cdcl_W-merge-cp** S V* **and** *conflicting V = None*
| (*cp-false*) *cdcl_W-merge-cp** S V* **and** *conflicting V ≠ None* **and** *no-step cdcl_W-cp V* **and**
 no-step cdcl_W-bj V
| (*cp-conf*) *T* **where** *cdcl_W-merge-cp** S T conflict T V*
using *rtrancp-cdcl_W-s'-without-decide-is-rtrancp-cdcl_W-merge-cp[of S V] confl*
unfolding *full-def* **by** *meson*
then have *cdcl_W-merge-cp** S V*
proof *cases*
 case *cp-conf* **note** *S-T = this(1)* **and** *conf-V = this(2)*
 have *full cdcl_W-bj V V*
 using *conf-V n-s-bj unfolding full-def* **by** *fast*
 then have *cdcl_W-merge-cp T V*
 using *cdcl_W-merge-cp.conflict' conf-V* **by** *auto*
 then show *?thesis* **using** *S-T* **by** *auto*
qed *fast+*
moreover
 then have *cdcl_W-restart** S V* **using** *rtrancp-cdcl_W-merge-cp-rtrancp-cdcl_W-restart* **by** *blast*
 then have *cdcl_W-restart-all-struct-inv V*
 using *inv rtrancp-cdcl_W-restart-all-struct-inv-inv* **by** *blast*
 then have *no-step cdcl_W-merge-cp V*
 using *conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp s'*
 unfolding *full-def* **by** *blast*
ultimately show *?fw* **unfolding** *full-def* **by** *auto*
qed

lemma *conflicting-true-full1-cdcl_W-merge-cp-iff-full1-cdcl_W-s'-without-decode:*
assumes
 confl: conflicting S = None **and**
 inv: cdcl_W-restart-all-struct-inv S
shows
 full1 cdcl_W-merge-cp S V ⟷ full1 cdcl_W-s'-without-decide S V
proof –
 have *full cdcl_W-merge-cp S V = full cdcl_W-s'-without-decide S V*
 using *confl conflicting-true-full-cdcl_W-merge-cp-iff-full-cdcl_W-s'-without-decode inv*
 by *simp*
 then show *?thesis* **unfolding** *full-unfold full1-def trancp-unfold-begin* **by** *blast*
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-restart-all-struct-inv S
shows
 full1 cdcl_W-s'-without-decide S V
proof –
 have *conflicting S = None*
 using *fw* **unfolding** *full1-def* **by** (*auto dest!: trancpD simp: cdcl_W-merge-cp.simps elim: rulesE*)

then show *?thesis*
using *conflicting-true-full1-cdcl_W-merge-cp-iff-full1-cdcl_W-s'-without-decode fw inv* **by** *simp*
qed

inductive *cdcl_W-merge-stgy* **for** *S :: 'st* **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-restart:*

*cdcl_W-merge-stgy S T \implies cdcl_W-restart^{**} S T*
apply (*induction rule: cdcl_W-merge-stgy.induct*)
using *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W-restart* **unfolding** *full1-def*
apply (*simp add: tranclp-into-rtranclp*)
using *rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W-restart cdcl_W-o.decide cdcl_W-restart.other* **unfolding**
full-def
by (*meson r-into-rtranclp rtranclp-trans*)

lemma *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-restart:*

*cdcl_W-merge-stgy^{**} S T \implies cdcl_W-restart^{**} S T*
using *rtranclp-mono[of cdcl_W-merge-stgy cdcl_W-restart^{**}] cdcl_W-merge-stgy-rtranclp-cdcl_W-restart* **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
 $\bigwedge T. \text{decide } S \text{ } T \implies \text{no-step } cdcl_W\text{-merge-cp } S \implies \text{full } cdcl_W\text{-merge-cp } T \text{ } U \implies P$
shows *P*
using *assms* **by** (*auto simp: cdcl_W-merge-stgy.simps*)

inductive *cdcl_W-s'-w :: 'st \Rightarrow 'st \Rightarrow bool* **where**

conflict': $\text{full1 } \text{cdcl}_W\text{-s'-without-decide } S \ S' \implies \text{cdcl}_W\text{-s'-w } S \ S' \mid$
decide': $\text{decide } S \ S' \implies \text{no-step } \text{cdcl}_W\text{-s'-without-decide } S \implies \text{full } \text{cdcl}_W\text{-s'-without-decide } S' \ S''$
 $\implies \text{cdcl}_W\text{-s'-w } S \ S''$

lemma *cdcl_W-s'-w-rtrancp-cdcl_W-restart*:
 $\text{cdcl}_W\text{-s'-w } S \ T \implies \text{cdcl}_W\text{-restart}^{**} \ S \ T$
apply (*induction rule*: *cdcl_W-s'-w.induct*)
using *rtrancp-cdcl_W-s'-without-decide-rtrancp-cdcl_W-restart* **unfolding** *full1-def*
apply (*simp add*: *trancp-into-rtrancp*)
using *rtrancp-cdcl_W-s'-without-decide-rtrancp-cdcl_W-restart* **unfolding** *full-def*
by (*meson decide other rtrancp-into-trancp2 trancp-into-rtrancp*)

lemma *rtrancp-cdcl_W-s'-w-rtrancp-cdcl_W-restart*:
 $\text{cdcl}_W\text{-s'-w}^{**} \ S \ T \implies \text{cdcl}_W\text{-restart}^{**} \ S \ T$
using *rtrancp-mono*[*of cdcl_W-s'-w cdcl_W-restart^{**}*] *cdcl_W-s'-w-rtrancp-cdcl_W-restart* **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-restart-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 trancpD*
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 trancpD*)

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*:
 $\text{cdcl}_W\text{-restart-all-struct-inv } S \implies \text{no-step } \text{cdcl}_W\text{-s'-without-decide } S \implies \text{no-step } \text{cdcl}_W\text{-cp } S$
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-restart-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-restart-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 trancp-unfold-end after-cdcl_W-s'-without-decide-no-step-cdcl_W-cp*)

next

case (*decide' S T U*)

moreover

then have *cdcl_W-restart^{**} S U*

using *rtrancp-cdcl_W-s'-without-decide-rtrancp-cdcl_W-restart*[*of T U*] *cdcl_W-restart.other*[*of S T*]
cdcl_W-o.decide **unfolding** *full-def* **by** *auto*

then have *cdcl_W-restart-all-struct-inv U*

using *decide'.prems rtrancp-cdcl_W-restart-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-restart-all-struct-inv S*
shows $S = T \vee \text{no-step } \text{cdcl}_W\text{-cp } T$
using *assms*
proof (*induction rule: rtrancpl-induct*)
case *base*
then show ?*case* **by** *simp*
next
case (*step T U*)
moreover have *cdcl_W-restart-all-struct-inv T*
using *rtrancpl-cdcl_W-s'-w-rtrancpl-cdcl_W-restart[of S U] assms(2) rtrancpl-cdcl_W-restart-all-struct-inv-inv*
rtrancpl-cdcl_W-s'-w-rtrancpl-cdcl_W-restart 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-restart-all-struct-inv S*
shows $S = T \vee \text{no-step } \text{cdcl}_W\text{-cp } T$
using *assms*
proof (*induction rule: rtrancpl-induct*)
case *base*
then show ?*case* **by** *simp*
next
case (*step T U*)
moreover have *cdcl_W-restart-all-struct-inv T*
using *rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W-restart[of S U] assms(2) rtrancpl-cdcl_W-restart-all-struct-inv-inv*
rtrancpl-cdcl_W-s'-w-rtrancpl-cdcl_W-restart step.hyps(1)
by (*meson rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W-restart*)
ultimately show ?*case*
using *after-cdcl_W-s'-w-no-step-cdcl_W-cp inv unfolding cdcl_W-restart-all-struct-inv-def*
by (*metis cdcl_W-restart-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-restart-all-struct-inv-inv
rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W-restart 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-restart-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-restart-all-struct-inv T*
using *S-T cdcl_W-restart-all-struct-inv-inv inv other* **by** *blast*
then obtain *T'* **where** *full1 cdcl_W-bj S T'*
using *cdcl_W-bj-exists-normal-form[of T] full-full1 S-T unfolding cdcl_W-restart-all-struct-inv-def*
by *metis*
moreover
then have *cdcl_W-restart** S T'*
using *rtrancpl-mono[of cdcl_W-bj cdcl_W-restart] cdcl_W-restart.other cdcl_W-o.bj trancpl-into-rtrancpl[of*
cdcl_W-bj]
unfolding *full1-def* **by** *blast*
then have *cdcl_W-restart-all-struct-inv T'*
using *inv rtrancpl-cdcl_W-restart-all-struct-inv-inv* **by** *blast*
then obtain *U* **where** *full cdcl_W-cp T' U*
using *cdcl_W-cp-normalized-element-all-inv* **by** *blast*

moreover have $\text{no-step } \text{cdcl}_W\text{-cp } S$
using $S\text{-}T$ **by** ($\text{auto simp: cdcl}_W\text{-bj.simps elim: rulesE}$)
ultimately show False
using $\text{assms cdcl}_W\text{-s'without-decide.intros}(2)[\text{of } S\ T'\ U]$ **by fast**
qed

lemma $\text{cdcl}_W\text{-s'-w-no-step-cdcl}_W\text{-bj}$:
assumes $\text{cdcl}_W\text{-s'-w } S\ T$ **and** $\text{cdcl}_W\text{-restart-all-struct-inv } S$
shows $\text{no-step cdcl}_W\text{-bj } T$
using $\text{assms apply induction}$
using $\text{rtrancp-cdcl}_W\text{-s'without-decide-rtrancp-cdcl}_W\text{-restart rtrancp-cdcl}_W\text{-restart-all-struct-inv-inv}$
 $\text{no-step-cdcl}_W\text{-s'without-decide-no-step-cdcl}_W\text{-bj}$ **unfolding** full1-def
apply ($\text{meson trancp-into-rtrancp}$)
using $\text{rtrancp-cdcl}_W\text{-s'without-decide-rtrancp-cdcl}_W\text{-restart rtrancp-cdcl}_W\text{-restart-all-struct-inv-inv}$
 $\text{no-step-cdcl}_W\text{-s'without-decide-no-step-cdcl}_W\text{-bj}$ **unfolding** full-def
by ($\text{meson cdcl}_W\text{-merge-restart-cdcl}_W\text{-restart fw-r-decide}$)

lemma $\text{rtrancp-cdcl}_W\text{-s'-w-no-step-cdcl}_W\text{-bj-or-eq}$:
assumes $\text{cdcl}_W\text{-s'-w}^{**} S\ T$ **and** $\text{cdcl}_W\text{-restart-all-struct-inv } S$
shows $S = T \vee \text{no-step cdcl}_W\text{-bj } T$
using $\text{assms apply induction}$
apply simp
using $\text{rtrancp-cdcl}_W\text{-s'-w-rtrancp-cdcl}_W\text{-restart rtrancp-cdcl}_W\text{-restart-all-struct-inv-inv}$
 $\text{cdcl}_W\text{-s'-w-no-step-cdcl}_W\text{-bj}$ **by** meson

lemma $\text{rtrancp-cdcl}_W\text{-s'-no-step-cdcl}_W\text{-s'without-decide-decomp-into-cdcl}_W\text{-merge}$:

assumes
 $\text{cdcl}_W\text{-s'}^{**} R\ V$ **and**
 $\text{conflicting } R = \text{None}$ **and**
 $\text{inv: cdcl}_W\text{-restart-all-struct-inv } R$
shows ($\text{cdcl}_W\text{-merge-stgy}^{**} R\ V \wedge \text{conflicting } V = \text{None}$)
 $\vee (\text{cdcl}_W\text{-merge-stgy}^{**} R\ V \wedge \text{conflicting } V \neq \text{None} \wedge \text{no-step cdcl}_W\text{-bj } V)$
 $\vee (\exists S\ T\ U. \text{cdcl}_W\text{-merge-stgy}^{**} R\ S \wedge \text{no-step cdcl}_W\text{-merge-cp } S \wedge \text{decide } S\ T$
 $\wedge \text{cdcl}_W\text{-merge-cp}^{**} T\ U \wedge \text{conflict } U\ V)$
 $\vee (\exists S\ T. \text{cdcl}_W\text{-merge-stgy}^{**} R\ S \wedge \text{no-step cdcl}_W\text{-merge-cp } S \wedge \text{decide } S\ T$
 $\wedge \text{cdcl}_W\text{-merge-cp}^{**} T\ V$
 $\wedge \text{conflicting } V = \text{None})$
 $\vee (\text{cdcl}_W\text{-merge-cp}^{**} R\ V \wedge \text{conflicting } V = \text{None})$
 $\vee (\exists U. \text{cdcl}_W\text{-merge-cp}^{**} R\ U \wedge \text{conflict } U\ V)$
using $\text{assms}(1,2)$

proof induction

case base

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

next

case ($\text{step } V\ W$) **note** $st = \text{this}(1)$ **and** $s' = \text{this}(2)$ **and** $IH = \text{this}(3)[\text{OF } \text{this}(4)]$ **and**
 $n\text{-s-}R = \text{this}(4)$

from s'

show $?case$

proof cases

case $\text{conflict}'$

consider

$(s')\ \text{cdcl}_W\text{-merge-stgy}^{**} R\ V$

| (dec-conf) $S\ T\ U$ **where** $\text{cdcl}_W\text{-merge-stgy}^{**} R\ S$ **and** $\text{no-step cdcl}_W\text{-merge-cp } S$ **and**
 $\text{decide } S\ T$ **and** $\text{cdcl}_W\text{-merge-cp}^{**} T\ U$ **and** $\text{conflict } U\ V$

| (dec) $S\ T$ **where** $\text{cdcl}_W\text{-merge-stgy}^{**} R\ S$ **and** $\text{no-step cdcl}_W\text{-merge-cp } S$ **and** $\text{decide } S\ T$
and $\text{cdcl}_W\text{-merge-cp}^{**} T\ V$ **and** $\text{conflicting } V = \text{None}$

```

| (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
  case s'
  then have R = V using inv local.conflict' unfolding full1-def
    by (metis 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 intro: cdclW-merge-cp.intros)
next
  case propa-conf
  moreover
    then have cdclW-merge-cp** V V'
    by (metis rtranclp-unfold cdclW-merge-cp.propagate' r-into-rtranclp)
    ultimately show ?thesis using s' ⟨R = V⟩ by blast
  qed
next
  case dec-conf note - = this(5)
  then have False using conflict' unfolding full1-def by (auto dest!: tranclpD elim: rulesE)
  then show ?thesis by fast
next
  case dec note T-V = this(4)
  consider
    (propa) propagate++ V W and conflicting W = None
  | (propa-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 rtranclp.rtrancl-into-rtrancl)
next
  case propa-conf
  then have cdclW-merge-cp** T V'
    using T-V by (metis rtranclp-unfold cdclW-merge-cp.propagate' rtranclp.simps)
  then show ?thesis using dec propa-conf(2) by metis
  qed
next
  case cp
  consider
    (propa) propagate++ V W and conflicting W = None
  | (propa-conf) V' where propagate** V V' and conflict V' W

```

```

    using trancp-cdclW-cp-propagate-with-conflict-or-not[of V W] conflict'
    unfolding full1-def by meson
  then show ?thesis
  proof cases
    case propa
    then show ?thesis by (meson cdclW-merge-cp.propagate' cp
      rtrancp.rtrancl-into-rtrancl)
  next
    case propa-conf
    then show ?thesis
      using propa-conf(2) cp
      by (metis (full-types) cdclW-merge-cp.propagate' rtrancp.rtrancl-into-rtrancl
        rtrancp-unfold)
  qed
next
  case cp-conf
  then show ?thesis using conflict' unfolding full1-def by (fastforce dest!: trancpD
    elim!: rulesE)
  qed
next
  case (decide' V')
  then have conf-V: conflicting V = None
    by (auto elim: rulesE)
  consider
    (s') cdclW-merge-stgy** R V
  | (dec-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
    case s'
    have conf-V': conflicting V' = None using decide'(1) by (auto elim: rulesE)
    have full: full1 cdclW-cp V' W  $\vee$  (V' = W  $\wedge$  no-step cdclW-cp W)
      using decide'(3) unfolding full-unfold by blast
    consider
      (V'-W) V' = W
    | (propa) propagate** V' W and conflicting W = None
    | (propa-conf) V'' where propagate** V' V'' and conflict V'' W
    using trancp-cdclW-cp-propagate-with-conflict-or-not[of V W] decide'
      (full1 cdclW-cp V' W  $\vee$  V' = W  $\wedge$  no-step cdclW-cp W) unfolding full1-def
      by (metis trancp-cdclW-cp-propagate-with-conflict-or-not)
    then show ?thesis
    proof cases
      case V'-W
      then show ?thesis
        using conf-V' local.decide'(1,2) s' conf-V
        no-step-cdclW-cp-no-step-cdclW-merge-restart[of V]
        by auto
    next
      case propa
      then show ?thesis using local.decide'(1,2) s' by (metis cdclW-merge-cp.simps conf-V
        no-step-cdclW-cp-no-step-cdclW-merge-restart r-into-rtrancp)
    next
      case propa-conf
      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-rtrancp)
    next
      case dec
      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-rtrancp)
    next
      case cp
      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-rtrancp)
    next
      case cp-conf
      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-rtrancp)
  qed

```

```

next
  case propa-conf
  then have cdclW-merge-cp** V' V''
    by (metis rtrancp-unfold cdclW-merge-cp.propagate' r-into-rtrancp)
  then show ?thesis
    using local.decide'(1,2) propa-conf(2) s' conf-V
    no-step-cdclW-cp-no-step-cdclW-merge-restart
    by metis
  qed
next
case (dec) note s' = this(1) and dec = this(2) and cp = this(3) and ns-cp-T = this(4)
have full cdclW-merge-cp T V
  unfolding full-def by (simp add: conf-V local.decide'(2)
    no-step-cdclW-cp-no-step-cdclW-merge-restart ns-cp-T)
moreover have no-step cdclW-merge-cp V
  by (simp add: conf-V local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart)
moreover have no-step cdclW-merge-cp S
  by (metis dec)
ultimately have cdclW-merge-stgy S V
  using cp by blast
then have cdclW-merge-stgy** R V using s' by auto
consider
  (V'-W) V' = W
  | (propa) propagate++ V' W and conflicting W = None
  | (propa-conf) V'' where propagate** V' V'' and conflict V'' W
  using trancp-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
  unfolding full-unfold full1-def by meson
then show ?thesis
proof cases
  case V'-W
  moreover have conflicting V' = None
    using decide'(1) by (auto elim: rulesE)
  ultimately show ?thesis
    using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩ by blast
next
  case propa
  moreover then have cdclW-merge-cp V' W by (blast intro: cdclW-merge-cp.intros)
  ultimately show ?thesis
    using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩
    by (meson r-into-rtrancp)
next
  case propa-conf
  moreover then have cdclW-merge-cp** V' V''
    by (metis cdclW-merge-cp.propagate' rtrancp-unfold trancp-unfold-end)
  ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
    ⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtrancp)
  qed
next
case cp
have no-step cdclW-merge-cp V
  using conf-V local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart by auto
then have full cdclW-merge-cp R V
  unfolding full-def using cp by fast
then have cdclW-merge-stgy** R V
  unfolding full-unfold by auto
have full1 cdclW-cp V' W  $\vee$  (V' = W  $\wedge$  no-step cdclW-cp W)

```

```

using decide'(3) unfolding full-unfold by blast

consider
  (V'-W) V' = W
| (propa) propagate++ V' W and conflicting W = None
| (propa-confl) V'' where propagate** V' V'' and conflict V'' W
using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] decide'
unfolding full-unfold full1-def by meson
then show ?thesis

proof cases
case V'-W
moreover have conflicting V' = None
using decide'(1) by (auto elim: rulesE)
ultimately show ?thesis
using ⟨cdclW-merge-stgy** R V⟩ decide' ⟨no-step cdclW-merge-cp V⟩ by blast
next
case propa
moreover then have cdclW-merge-cp V' W
by (blast intro: cdclW-merge-cp.intros)
ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)
next
case propa-confl
moreover then have cdclW-merge-cp** V' V''
by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
ultimately show ?thesis using ⟨cdclW-merge-stgy** R V⟩ decide'
⟨no-step cdclW-merge-cp V⟩ by (meson r-into-rtranclp)
qed
next
case (dec-confl)
show ?thesis using conf-V dec-confl(5) by (auto elim!: rulesE
simp del: state-simp simp: state-eq-def)
next
case cp-confl
then show ?thesis using decide' apply - by (intro HOL.disjI2) (fastforce elim: rulesE
simp del: state-simp simp: state-eq-def)
qed
next
case (bj' V')
then have ¬no-step cdclW-bj V
by (auto dest: tranclpD simp: full1-def)
then consider
  (s') cdclW-merge-stgy** R V and conflicting V = None
| (dec-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

```

```

    elim: rulesE)
  then show ?thesis by fast
next
  case dec note - = this(5)
  then have False
    using bj'(1) unfolding full1-def by (force dest!: tranclpD simp: cdclW-bj.simps
      elim: rulesE)
  then show ?thesis by fast
next
  case dec-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'⟩
        conflictE conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj
        dec-confl(5) r-into-rtranclp)
  qed
next
  case propa
  moreover then have cdclW-merge-cp V' W by (blast intro: cdclW-merge-cp.intros)
  ultimately show ?thesis using decide' by (meson ⟨cdclW-merge-cp** T V'⟩ dec-confl(1-3)
    rtranclp.rtrancl-into-rtrancl)
next
  case propa-confl
  moreover then have cdclW-merge-cp** V' V''
    by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)

```

```

ultimately show ?thesis by (meson ⟨cdclW-merge-cp** T V'⟩ dec-confl(1-3) rtranclp-trans)
qed
next
case cp note - = this(2)
then show ?thesis using bj'(1) ⟨¬ no-step cdclW-bj V⟩
  conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj by auto
next
case cp-confl
then have cdclW-merge-cp U V' by (simp add: cdclW-merge-cp.conflict' full-unfold
  local.bj'(1))
consider
  (V'-W) V' = W
| (propa) propagate++ V' W and conflicting W = None
| (propa-confl) V'' where propagate** V' V'' and conflict V'' W
using tranclp-cdclW-cp-propagate-with-conflict-or-not[of V' W] bj'
unfolding full-unfold full1-def by meson
then show ?thesis

proof cases
case V'-W
show ?thesis
proof cases
assume conflicting V' = None
then show ?thesis
  using V'-W ⟨cdclW-merge-cp U V'⟩ cp-confl(1) by force
next
assume confl: conflicting V' ≠ None
then have no-step cdclW-merge-stgy V'
  by (fastforce simp: cdclW-merge-stgy.simps full1-def full-def
    cdclW-merge-cp.simps dest!: tranclpD elim: rulesE)
have no-step cdclW-merge-cp V'
  using confl by (auto simp: full1-def full-def cdclW-merge-cp.simps
    dest!: tranclpD elim: rulesE)
moreover have cdclW-merge-cp U W
  using V'-W ⟨cdclW-merge-cp U V'⟩ by blast
ultimately have full1 cdclW-merge-cp R V'
  using cp-confl(1) V'-W unfolding full1-def by auto
then have cdclW-merge-stgy R V'
  by auto
moreover have no-step cdclW-merge-stgy V'
  using confl ⟨no-step cdclW-merge-cp V'⟩ by (auto simp: cdclW-merge-stgy.simps
    full1-def dest!: tranclpD elim: rulesE)
ultimately have cdclW-merge-stgy** R V' by auto
{ fix ss :: 'st
  have cdclW-merge-cp U W
    using V'-W ⟨cdclW-merge-cp U V'⟩ by blast
  then have ¬ cdclW-bj W ss
    by (meson conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj
      cp-confl(1) rtranclp.rtrancl-into-rtrancl step.prem)
  then have cdclW-merge-stgy** R W ∧ conflicting W = None ∨
    cdclW-merge-stgy** R W ∧ ¬ cdclW-bj W ss
    using V'-W ⟨cdclW-merge-stgy** R V'⟩ by presburger }
then show ?thesis
  by presburger
qed
next

```



```

    case propa
    moreover then have cdclW-merge-cp V' W
      by (blast intro: cdclW-merge-cp.intros)
    ultimately show ?thesis using ⟨cdclW-merge-cp U V'⟩ cp-confl(1) by force
  next
  case propa-confl
  moreover then have cdclW-merge-cp** V' V''
    by (metis cdclW-merge-cp.propagate' rtranclp-unfold tranclp-unfold-end)
  ultimately show ?thesis
    using ⟨cdclW-merge-cp U V'⟩ cp-confl(1) by (metis rtranclp.rtrancl-into-rtrancl
      rtranclp-trans)
  qed
qed
qed
qed

```

lemma *decide-rtranclp-cdcl_W-s'-rtranclp-cdcl_W-s'*:

```

  assumes
    dec: decide S T and
    cdclW-s'** T U and
    n-s-S: no-step cdclW-cp S and
    no-step cdclW-cp U
  shows cdclW-s'** S U
  using assms(2,4)

```

proof *induction*

```

  case (step U V) note st = this(1) and s' = this(2) and IH = this(3) and n-s = this(4)
  consider

```

```

    (TU) T = U
  | (s'-st) T' where cdclW-s' T T' and cdclW-s'** T' U
  using st[unfolded rtranclp-unfold] by (auto dest!: tranclpD)

```

then show ?case

proof *cases*

case TU

then show ?thesis

proof –

assume a1: T = U

then have f2: cdcl_W-s' T V

using s' by force

obtain ss :: 'st where

ss: cdcl_W-s^{'**} S T ∨ cdcl_W-cp T ss

using a1 step.IH by blast–

obtain ssa :: 'st ⇒ 'st where

f3: ∀ s sa sb. (¬ decide s sa ∨ cdcl_W-cp s (ssa s) ∨ ¬ full cdcl_W-cp sa sb) ∨ cdcl_W-s' s sb

using cdcl_W-s'.decide' by moura

have ∀ s sa. ¬ cdcl_W-s' s sa ∨ full1 cdcl_W-cp s sa ∨

(∃ sb. decide s sb ∧ no-step cdcl_W-cp s ∧ full cdcl_W-cp sb sa) ∨

(∃ sb. full1 cdcl_W-bj s sb ∧ no-step cdcl_W-cp s ∧ full cdcl_W-cp sb sa)

by (metis cdcl_W-s'E)

then have ∃ s. cdcl_W-s^{'**} S s ∧ cdcl_W-s' s V

using f3 ss f2 by (metis dec full1-is-full n-s-S rtranclp-unfold)

then show ?thesis

by force

qed

next

case (s'-st T') **note** s'-T' = this(1) **and** st = this(2)

```

have cdclW-sl** S T'
  using s'-T'
  proof cases
    case conflict'
      then have cdclW-s' S T'
        using dec cdclW-s'.decide' n-s-S by (simp add: full-unfold)
      then show ?thesis
        using st by auto
    next
      case (decide' T'')
      then have cdclW-s' S T
        using dec cdclW-s'.decide' n-s-S by (simp add: full-unfold)
      then show ?thesis using decide' s'-T' by auto
    next
      case bj'
      then have False
        using dec unfolding full1-def by (fastforce dest!: tranclpD simp: cdclW-bj.simps
          elim: rulesE)
      then show ?thesis by fast
  qed
then show ?thesis using s' st by auto
qed
next
case base
then have full cdclW-cp T T
  by (simp add: full-unfold)
then show ?case
  using cdclW-s'.simps dec n-s-S by auto
qed

lemma rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s':
  assumes
    cdclW-merge-stgy** R V and
    inv: cdclW-restart-all-struct-inv R
  shows cdclW-sl** R V
  using assms(1)
proof induction
  case base
  then show ?case by simp
next
  case (step S T) note st = this(1) and fw = this(2) and IH = this(3)
  have cdclW-restart-all-struct-inv S
    using inv rtranclp-cdclW-restart-all-struct-inv-inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW-restart
    st by blast
  from fw show ?case
  proof (cases rule: cdclW-merge-stgy-cases)
    case fw-s-cp
    have  $\bigwedge s. \neg \text{full cdcl}_W\text{-merge-cp } s \ S$ 
      using fw-s-cp unfolding full-def full1-def by (metis tranclp-unfold-begin)
    then have S = R
      using fw-s-cp unfolding full1-def by (metis cdclW-cp.conflict' cdclW-cp.propagate'
        cdclW-merge-cp.cases tranclp-unfold-begin inv st
        rtranclp-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq)
    then have full1 cdclW-s'-without-decide R T
      using inv local.fw-s-cp
      by (blast intro: conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode)
  qed

```

```

then show ?thesis unfolding full1-def
  by (metis (no-types) rtranclp-cdclW-s'-without-decide-rtranclp-cdclW-s' rtranclp-unfold)
next
case (fw-s-decide S') note dec = this(1) and n-S = this(2) and full = this(3)
moreover then have conflicting S' = None
  by (auto elim: rulesE)
ultimately have full cdclW-s'-without-decide S' T
  by (meson (cdclW-restart-all-struct-inv S) cdclW-merge-restart-cdclW-restart fw-r-decide
    rtranclp-cdclW-restart-all-struct-inv-inv
    conflicting-true-full-cdclW-merge-cp-iff-full-cdclW-s'-without-decide)
then have a1: cdclW-s!* S' T
  unfolding full-def by (metis (full-types) rtranclp-cdclW-s'-without-decide-rtranclp-cdclW-s')
have cdclW-merge-stgy!* S T
  using fw by blast
then have cdclW-s!* S T
  using decide-rtranclp-cdclW-s'-rtranclp-cdclW-s' a1 by (metis (cdclW-restart-all-struct-inv S) dec
    n-S no-step-cdclW-merge-cp-no-step-cdclW-cp cdclW-restart-all-struct-inv-def
    rtranclp-cdclW-merge-stgy'-no-step-cdclW-cp-or-eq)
then show ?thesis using IH by auto
qed
qed

```

lemma *rtranclp-cdcl_W-merge-stgy-distinct-mset-clauses:*

```

assumes invR: cdclW-restart-all-struct-inv R and
  st: cdclW-merge-stgy!* R S and
  dist: distinct-mset (clauses R) and
  R: trail R = []
shows distinct-mset (clauses S)
using rtranclp-cdclW-stgy-distinct-mset-clauses[OF invR - dist R]
invR st rtranclp-mono[of cdclW-s' cdclW-stgy!*] cdclW-s'-is-rtranclp-cdclW-stgy
by (auto dest!: cdclW-s'-is-rtranclp-cdclW-stgy rtranclp-cdclW-merge-stgy-rtranclp-cdclW-s')

```

lemma *no-step-cdcl_W-s'-no-step-cdcl_W-merge-stgy:*

```

assumes
  inv: cdclW-restart-all-struct-inv R and s': no-step cdclW-s' R
shows no-step cdclW-merge-stgy R

```

proof –

```

{ fix ss :: 'st
  obtain ssa :: 'st ⇒ 'st ⇒ 'st where
    ff1: ⋀ s sa. ¬ cdclW-merge-stgy s sa ∨ full1 cdclW-merge-cp s sa ∨ decide s (ssa s sa)
    using cdclW-merge-stgy.cases by moura
  obtain ssb :: ('st ⇒ 'st ⇒ bool) ⇒ 'st ⇒ 'st ⇒ 'st where
    ff2: ⋀ p s sa. ¬ p++ s sa ∨ p s (ssb p s sa)
    by (meson tranclp-unfold-begin)
  obtain ssc :: 'st ⇒ 'st where
    ff3: ⋀ s sa sb. (¬ cdclW-restart-all-struct-inv s ∨ ¬ cdclW-cp s sa ∨ cdclW-s' s (ssc s))
    ∧ (¬ cdclW-restart-all-struct-inv s ∨ ¬ cdclW-o s sb ∨ cdclW-s' s (ssc s))
    using n-step-cdclW-stgy-iff-no-step-cdclW-restart-cl-cdclW-o by moura
  then have ff4: ⋀ s. ¬ cdclW-o R s
    using s' inv by blast
  have ff5: ⋀ s. ¬ cdclW-cp++ R s
    using ff3 ff2 s' by (metis inv)
  have ⋀ s. ¬ cdclW-bj++ R s
    using ff4 ff2 by (metis bj)
  then have ⋀ s. ¬ cdclW-s'-without-decide R s
    using ff5 by (simp add: cdclW-s'-without-decide.simps full1-def)
}

```

```

then have  $\neg \text{cdcl}_W\text{-s'-without-decide}^{++} R ss$ 
  using ff2 by blast
then have  $\neg \text{full1 cdcl}_W\text{-s'-without-decide} R ss$ 
  by (simp add: full1-def)
then have  $\neg \text{cdcl}_W\text{-merge-stgy} R ss$ 
  using ff4 ff1 conflicting-true-full1-cdclW-merge-cp-imp-full1-cdclW-s'-without-decode inv
  by blast }
then show ?thesis
  by fastforce
qed
end

```

Termination and full Equivalence

We will discharge the assumption later using NOT's proof of termination.

```

locale conflict-driven-clause-learningW-termination =
  conflict-driven-clause-learningW +
  assumes wf-cdclW-merge-inv: wf {(T, S). cdclW-restart-all-struct-inv S ∧ cdclW-merge S T}
begin

```

```

lemma wf-tranclp-cdclW-merge: wf {(T, S). cdclW-restart-all-struct-inv S ∧ cdclW-merge++ S T}
  using wf-trancl[OF wf-cdclW-merge-inv]
  apply (rule wf-subset)
  by (auto simp: trancl-set-tranclp
    cdclW-restart-all-struct-inv-tranclp-cdclW-merge-tranclp-cdclW-merge-cdclW-restart-all-struct-inv)

```

```

lemma wf-cdclW-merge-cp:
  wf{(T, S). cdclW-restart-all-struct-inv S ∧ cdclW-merge-cp S T}
  using wf-tranclp-cdclW-merge by (rule wf-subset) (auto simp: cdclW-merge-cp-tranclp-cdclW-merge)

```

```

lemma wf-cdclW-merge-stgy:
  wf{(T, S). cdclW-restart-all-struct-inv S ∧ cdclW-merge-stgy S T}
  using wf-tranclp-cdclW-merge by (rule wf-subset)
  (auto simp add: cdclW-merge-stgy-tranclp-cdclW-merge)

```

```

lemma cdclW-merge-cp-obtain-normal-form:
  assumes inv: cdclW-restart-all-struct-inv R
  obtains S where full cdclW-merge-cp R S

```

proof –

```

obtain S where full (λS T. cdclW-restart-all-struct-inv S ∧ cdclW-merge-cp S T) R S
  using wf-exists-normal-form-full[OF wf-cdclW-merge-cp] by blast

```

then have

```

  st: (λS T. cdclW-restart-all-struct-inv S ∧ cdclW-merge-cp S T)** R S and
  n-s: no-step (λS T. cdclW-restart-all-struct-inv S ∧ cdclW-merge-cp S T) S
  unfolding full-def by blast+

```

```

have cdclW-merge-cp** R S
  using st by induction auto

```

moreover

```

have cdclW-restart-all-struct-inv S
  using st inv
  apply (induction rule: rtranclp-induct)
  apply simp
  by (meson r-into-rtranclp rtranclp-cdclW-restart-all-struct-inv-inv
    rtranclp-cdclW-merge-cp-rtranclp-cdclW-restart)

```

```

then have no-step cdclW-merge-cp S

```

```

    using n-s by auto
ultimately show ?thesis
    using that unfolding full-def by blast
qed

lemma no-step-cdclW-merge-stgy-no-step-cdclW-s':
  assumes
    inv: cdclW-restart-all-struct-inv R and
    confl: conflicting R = None and
    n-s: no-step cdclW-merge-stgy R
  shows no-step cdclW-s' R
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain S where cdclW-s' R S by auto
  then show False
  proof cases
    case conflict'
    then obtain S' where full1 cdclW-merge-cp R S'
    proof -
      obtain R' where
        cdclW-merge-cp R R'
      using inv unfolding cdclW-restart-all-struct-inv-def by (meson confl
        cdclW-s'-without-decide.simps conflict'
        conflicting-true-no-step-cdclW-merge-cp-no-step-s'-without-decide)
      then show ?thesis
        using that by (metis cdclW-merge-cp-obtain-normal-form full-unfold inv)
    qed
    then show False using n-s by blast
  next
    case (decide' R')
    then have cdclW-restart-all-struct-inv R'
      using inv cdclW-restart-all-struct-inv-inv cdclW-restart.other cdclW-o.decide by meson
    then obtain R'' where full cdclW-merge-cp R' R''
      using cdclW-merge-cp-obtain-normal-form by blast
    moreover have no-step cdclW-merge-cp R
      by (simp add: confl local.decide'(2) no-step-cdclW-cp-no-step-cdclW-merge-restart)
    ultimately show False using n-s cdclW-merge-stgy.intros local.decide'(1) by blast
  next
    case (bj' R')
    then show False
      using confl no-step-cdclW-cp-no-step-cdclW-s'-without-decide inv
      unfolding cdclW-restart-all-struct-inv-def by auto
  qed
qed

lemma rtrancp-cdclW-merge-cp-no-step-cdclW-bj:
  assumes conflicting R = None and cdclW-merge-cp** R S
  shows no-step cdclW-bj S
  using assms conflicting-not-true-rtrancp-cdclW-merge-cp-no-step-cdclW-bj by auto

lemma rtrancp-cdclW-merge-stgy-no-step-cdclW-bj:
  assumes confl: conflicting R = None and cdclW-merge-stgy** R S
  shows no-step cdclW-bj S
  using assms(2)
proof induction
  case base

```

```

then show ?case
  using confl by (auto simp: cdclW-bj.simps elim: rulesE)
next
case (step S T) note st = this(1) and fw = this(2) and IH = this(3)
have confl-S: conflicting S = None
  using fw apply cases
  by (auto simp: full1-def cdclW-merge-cp.simps dest!: tranclpD elim: rulesE)
from fw show ?case
  proof cases
    case fw-s-cp
    then show ?thesis
      using rtranclp-cdclW-merge-cp-no-step-cdclW-bj confl-S
      by (simp add: full1-def tranclp-into-rtranclp)
  next
    case (fw-s-decide S')
    moreover then have conflicting S' = None by (auto elim: rulesE)
    ultimately show ?thesis
      using conflicting-not-true-rtranclp-cdclW-merge-cp-no-step-cdclW-bj
      unfolding full-def by meson
  qed
qed

end

end
theory CDCL-WNOT
imports CDCL-NOT CDCL-W-Termination CDCL-W-Merge
begin

```

6.3 Link between Weidenbach's and NOT's CDCL

6.3.1 Inclusion of the states

```

declare upt.simps(2)[simp del]

fun convert-ann-lit-from-W where
  convert-ann-lit-from-W (Propagated L -) = Propagated L () |
  convert-ann-lit-from-W (Decided L) = Decided L

abbreviation convert-trail-from-W ::
  ('v, 'mark) ann-lits
  ⇒ ('v, unit) ann-lits where
  convert-trail-from-W ≡ map convert-ann-lit-from-W

lemma lits-of-l-convert-trail-from-W[simp]:
  lits-of-l (convert-trail-from-W M) = lits-of-l M
  by (induction rule: ann-lit-list-induct) simp-all

lemma lit-of-convert-trail-from-W[simp]:
  lit-of (convert-ann-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) ⟷ no-dup M
  by (auto simp: comp-def)

```

lemma *convert-trail-from-W-true-annots*[simp]:
 $\text{convert-trail-from-W } M \models_{\text{as}} C \longleftrightarrow M \models_{\text{as}} C$
by (auto simp: true-annots-true-cls image-image lits-of-def)

lemma *defined-lit-convert-trail-from-W*[simp]:
 $\text{defined-lit } (\text{convert-trail-from-W } S) L \longleftrightarrow \text{defined-lit } S L$
by (auto simp: defined-lit-map image-comp)

The values 0 and $\{\#\}$ are dummy values.

consts *dummy-cls* :: 'cls
fun *convert-ann-lit-from-NOT*
 :: ('v, 'mark) ann-lit \Rightarrow ('v, 'cls) ann-lit **where**
convert-ann-lit-from-NOT (Propagated L -) = Propagated L dummy-cls |
convert-ann-lit-from-NOT (Decided L) = Decided L

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT \equiv map *convert-ann-lit-from-NOT*

lemma *undefined-lit-convert-trail-from-NOT*[simp]:
 $\text{undefined-lit } (\text{convert-trail-from-NOT } F) L \longleftrightarrow \text{undefined-lit } F L$
by (induction F rule: ann-lit-list-induct) (auto simp: defined-lit-map)

lemma *lits-of-l-convert-trail-from-NOT*:
 $\text{lits-of-l } (\text{convert-trail-from-NOT } F) = \text{lits-of-l } F$
by (induction F rule: ann-lit-list-induct) auto

lemma *convert-trail-from-W-from-NOT*[simp]:
 $\text{convert-trail-from-W } (\text{convert-trail-from-NOT } M) = M$
by (induction rule: ann-lit-list-induct) auto

lemma *convert-trail-from-W-convert-lit-from-NOT*[simp]:
 $\text{convert-ann-lit-from-W } (\text{convert-ann-lit-from-NOT } L) = L$
by (cases L) auto

abbreviation *trail_{NOT}* **where**
trail_{NOT} $S \equiv \text{convert-trail-from-W } (\text{fst } S)$

lemma *undefined-lit-convert-trail-from-W*[iff]:
 $\text{undefined-lit } (\text{convert-trail-from-W } M) L \longleftrightarrow \text{undefined-lit } M L$
by (auto simp: defined-lit-map image-comp)

lemma *lit-of-convert-ann-lit-from-NOT*[iff]:
 $\text{lit-of } (\text{convert-ann-lit-from-NOT } L) = \text{lit-of } L$
by (cases L) auto

sublocale *state_W* \subseteq *dpll-state-ops*
 $\lambda S.$ *convert-trail-from-W* (trail S)
clauses
 λL $S.$ *cons-trail* (convert-ann-lit-from-NOT L) S
 $\lambda S.$ *tl-trail* S
 λC $S.$ *add-learned-cls* C S
 λC $S.$ *remove-cls* C S
by *unfold-locales*

sublocale *state_W* \subseteq *dpll-state*

```

λS. convert-trail-from-W (trail S)
clauses
λL S. cons-trail (convert-ann-lit-from-NOT L) S
λS. tl-trail S
λC S. add-learned-cls C S
λC 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 conflict-driven-clause-learning_W ⊆ cdcl_NOT-merge-bj-learn-ops
λS. convert-trail-from-W (trail S)
clauses
λL S. cons-trail (convert-ann-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
λC C' L' S T. backjump-l-cond C C' L' S T
  ∧ distinct-mset (C' + {#L'#}) ∧ ¬tautology (C' + {#L'#})
by unfold-locales

thm cdcl_NOT-merge-bj-learn-proxy.axioms
sublocale conflict-driven-clause-learning_W ⊆ cdcl_NOT-merge-bj-learn-proxy
λS. convert-trail-from-W (trail S)
clauses
λL S. cons-trail (convert-ann-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
inv_NOT
proof (unfold-locales, goal-cases)
case 2
then show ?case using cdcl_NOT-merged-bj-learn-no-dup-inv by (auto simp: comp-def)
next
case (1 C' S C F' K F L)
moreover
let ?C' = remdups-mset C'
have L ∉ # C'
  using ⟨F ⊨as CNot C'⟩ ⟨undefined-lit F L⟩ Decided-Propagated-in-iff-in-lits-of-l
  in-CNot-implies-uminus(2) by fast
then have distinct-mset (?C' + {#L'#})
  by (simp add: distinct-mset-single-add)
moreover
have no-dup F
  using ⟨inv_NOT S⟩ ⟨convert-trail-from-W (trail S) = F' @ Decided K # F⟩
  unfolding inv_NOT-def
  by (smt comp-apply distinct.simps(2) distinct-append list.simps(9) map-append

```



```

    no-dup-convert-from-W)
  then have consistent-interp (lits-of-l F)
    using distinct-consistent-interp by blast
  then have  $\neg$  tautology  $C'$ 
    using  $\langle F \models_{as} CNot\ C' \rangle$  consistent-CNot-not-tautology true-annots-true-cls by blast
  then have  $\neg$  tautology  $(?C' + \{\#L\# \})$ 
    using  $\langle F \models_{as} CNot\ C' \rangle$   $\langle undefined-lit\ F\ L \rangle$  by (metis CNot-remdups-mset
      Decided-Propagated-in-iff-in-lits-of-l add commute in-CNot-uminus tautology-add-single
      tautology-remdups-mset true-annot-singleton true-annots-def)
  show ?case
  proof -
    have f2: no-dup (convert-trail-from-W (trail S))
      using  $\langle inv_{NOT}\ S \rangle$  unfolding  $inv_{NOT}$ -def by (simp add: o-def)
    have f3: atm-of  $L \in$  atms-of-mm (clauses S)
       $\cup$  atm-of ' lits-of-l (convert-trail-from-W (trail S))
      using  $\langle convert-trail-from-W\ (trail\ S) = F' @ Decided\ K \# F \rangle$ 
       $\langle atm-of\ L \in atms-of-mm\ (clauses\ S) \cup atm-of\ ' lits-of-l\ (F' @ Decided\ K \# F) \rangle$  by auto
    have f4: clauses S  $\models_{pm}$  remdups-mset  $C' + \{\#L\# \}$ 
      by (metis (no-types)  $\langle L \notin \# C' \rangle$   $\langle clauses\ S \models_{pm}\ C' + \{\#L\# \} \rangle$  remdups-mset-singleton-sum(2)
      true-clss-cls-remdups-mset union-commute)
    have  $F \models_{as} CNot\ (remdups-mset\ C')$ 
      by (simp add:  $\langle F \models_{as} CNot\ C' \rangle$ )
    have Ex (backjump-l S)
      apply standard
      apply (rule backjump-l.intros[OF - f2, of - - ])
      using f4 f3 f2  $\neg$  tautology (remdups-mset  $C' + \{\#L\# \}$ )
      calculation(2-5,9)  $\langle F \models_{as} CNot\ (remdups-mset\ C') \rangle$ 
      state-eqNOT-ref unfolding backjump-l-cond-def by blast+
    then show ?thesis
      by blast
  qed
qed

```

```

sublocale conflict-driven-clause-learningW  $\subseteq$  cdclNOT-merge-bj-learn-proxy2
   $\lambda S.$  convert-trail-from-W (trail S)
  clauses
   $\lambda L\ S.$  cons-trail (convert-ann-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 backjump-l-cond  $inv_{NOT}$ 
  by unfold-locales

```

```

sublocale conflict-driven-clause-learningW  $\subseteq$  cdclNOT-merge-bj-learn
   $\lambda S.$  convert-trail-from-W (trail S)
  clauses
   $\lambda L\ S.$  cons-trail (convert-ann-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
  backjump-l-cond
   $\lambda -.$  True
   $\lambda -.$   $S.$  conflicting S = None  $inv_{NOT}$ 
  apply unfold-locales
  using dpll-bj-no-dup apply (simp add: comp-def)

```

using *cdcl_{NOT}.simps cdcl_{NOT}-no-dup no-dup-convert-from-W* **unfolding** *inv_{NOT}-def* **by** *blast*

context *conflict-driven-clause-learning_W*
begin

Notations are lost while proving locale inclusion:

notation *state-eq_{NOT}* (**infix** \sim_{NOT} 50)

6.3.2 Additional Lemmas between NOT and W states

lemma *trail_W-eq-reduce-trail-to_{NOT}-eq*:

trail S = trail T \implies trail (reduce-trail-to_{NOT} F S) = trail (reduce-trail-to_{NOT} F T)

proof (*induction F S arbitrary: T rule: reduce-trail-to_{NOT}.induct*)

case (*1 F S T*) **note** *IH = this(1)* **and** *tr = this(2)*

then have $\square = \text{convert-trail-from-W (trail S)}$

$\vee \text{length F} = \text{length (convert-trail-from-W (trail S))}$

$\vee \text{trail (reduce-trail-to}_{NOT} \text{ F (tl-trail S))} = \text{trail (reduce-trail-to}_{NOT} \text{ F (tl-trail T))}$

using *IH* **by** (*metis (no-types) trail-tl-trail*)

then show *trail (reduce-trail-to_{NOT} F S) = trail (reduce-trail-to_{NOT} F T)*

using *tr* **by** (*metis (no-types) reduce-trail-to_{NOT}.elims*)

qed

lemma *trail-reduce-trail-to_{NOT}-add-learned-cls*:

no-dup (trail S) \implies

trail (reduce-trail-to_{NOT} M (add-learned-cls D S)) = trail (reduce-trail-to_{NOT} M S)

by (*rule trail_W-eq-reduce-trail-to_{NOT}-eq simp*)

lemma *reduce-trail-to_{NOT}-reduce-trail-convert*:

reduce-trail-to_{NOT} C S = reduce-trail-to (convert-trail-from-NOT C) S

apply (*induction C S rule: reduce-trail-to_{NOT}.induct*)

apply (*subst reduce-trail-to_{NOT}.simps, subst reduce-trail-to.simps*)

by *auto*

lemma *reduce-trail-to-map[simp]*:

reduce-trail-to (map f M) S = reduce-trail-to M S

by (*rule reduce-trail-to-length simp*)

lemma *reduce-trail-to_{NOT}-map[simp]*:

reduce-trail-to_{NOT} (map f M) S = reduce-trail-to_{NOT} M S

by (*rule reduce-trail-to_{NOT}-length simp*)

lemma *skip-or-resolve-state-change*:

assumes *skip-or-resolve** S T*

shows

$\exists M. \text{trail S} = M @ \text{trail T} \wedge (\forall m \in \text{set M}. \neg \text{is-decided } m)$

clauses S = clauses T

backtrack-lvl S = backtrack-lvl T

using *assms*

proof (*induction rule: rtranclp-induct*)

case *base*

case 1 **show** *?case* **by** *simp*

case 2 **show** *?case* **by** *simp*

case 3 **show** *?case* **by** *simp*

next

case (*step T U*) **note** *st = this(1)* **and** *s-o-r = this(2)* **and** *IH = this(3)* **and** *IH' = this(3-5)*

```

case 2 show ?case using IH' s-o-r by (auto elim!: rulesE simp: skip-or-resolve.simps)
case 3 show ?case using IH' s-o-r by (auto elim!: rulesE simp: skip-or-resolve.simps)
case 1 show ?case
  using s-o-r
  proof cases
    case s-or-r-skip
    then show ?thesis using IH by (auto elim!: rulesE simp: skip-or-resolve.simps)
  next
    case s-or-r-resolve
    then show ?thesis
      using IH by (cases trail T) (auto elim!: rulesE simp: skip-or-resolve.simps)
  qed
qed

```

6.3.3 Inclusion of Weidenbach's CDCL in NOT's CDCL

This lemma shows the inclusion of Weidenbach's CDCL *cdcl_W-merge* (with merging) in NOT's *cdcl_{NOT}-merged-bj-learn*.

lemma *cdcl_W-merge-is-cdcl_{NOT}-merged-bj-learn*:

```

assumes
  inv: cdclW-restart-all-struct-inv S and
  cdclW-restart: cdclW-merge S T
shows cdclNOT-merged-bj-learn S T
  ∨ (no-step cdclW-merge T ∧ conflicting T ≠ None)
using cdclW-restart inv
proof induction
  case (fw-propagate S T) note propa = this(1)
  then obtain M N U k L C where
    H: state S = (M, N, U, k, None) and
    CL: C + {#L#} ∈ # clauses S and
    M-C: M ⊨as CNot C and
    undef: undefined-lit (trail S) L and
    T: state T = (Propagated L (C + {#L#})) # M, N, U, k, None)
    by (auto elim: propagate-high-levelE)
  have propagateNOT S T
    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-mm (init-clss S) and
    T: T ∼ cons-trail (Decided L)
    (update-backtrack-lvl (Suc (backtrack-lvl S)) S)
    by (auto elim: decideE)
  have decideNOT S T
    apply (rule decideNOT.decideNOT)
    using undef-L apply simp
    using atm-L inv unfolding cdclW-restart-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  $C$  where
   $S$ : conflicting  $S = \text{None}$  and
   $C$ -le:  $C \in \#$  learned-clss  $S$  and
   $\neg(\text{trail } S) \models_{asm} \text{clauses } S$  and
   $C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S))$  and
   $C$ -init:  $C \notin \#$  init-clss  $S$  and
   $T$ :  $T \sim \text{remove-clss } C$ 
  by (auto elim: forgetE)
have init-clss  $S \models_{pm} C$ 
using inv  $C$ -le unfolding cdclW-restart-all-struct-inv-def cdclW-restart-learned-clause-def clauses-def
  by (meson true-clss-clss-in-imp-true-clss-clss)
then have  $S$ - $C$ : removeAll-mset  $C$  (clauses  $S$ )  $\models_{pm} C$ 
using  $C$ -init  $C$ -le unfolding clauses-def by (auto simp add: Un-Diff ac-simps)
have forgetNOT  $S$   $T$ 
apply (rule forgetNOT.forgetNOT)
  using  $S$ - $C$  apply blast
  using  $S$  apply simp
  using  $C$ -init  $C$ -le 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
  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  $C_S$   $CT$  where
  confl- $T$ : conflicting  $T = \text{Some } CT$  and
   $CT$ :  $CT = C_S$  and
   $C_S$ :  $C_S \in \#$  clauses  $S$  and
  tr- $S$ - $C_S$ : trail  $S \models_{as} CNot C_S$ 
using confl by (elim conflictE) (auto simp del: state-simp simp: state-eq-def)
have cdclW-restart-all-struct-inv  $T$ 
using cdclW-restart.simps cdclW-restart-all-struct-inv-inv confl inv by blast
then have cdclW-restart-M-level-inv  $T$ 
unfolding cdclW-restart-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  $U \neq \text{None}$ 
using confl by (induction rule: rtrancpl-induct)
  (auto simp del: state-simp simp: skip-or-resolve.simps state-eq-def elim!: rulesE)
moreover then have no-step cdclW-merge  $U$ 
by (auto simp: cdclW-merge.simps elim: rulesE)
ultimately show ?thesis by blast
next
case bt note s-or-r = this(1) and bt = this(2)
have cdclW-restart**  $T$   $T'$ 
using s-or-r mono-rtrancpl[of skip-or-resolve cdclW-restart] rtrancpl-skip-or-resolve-rtrancpl-cdclW-restart
by blast
then have cdclW-restart-M-level-inv  $T'$ 
using rtrancpl-cdclW-restart-consistent-inv <cdclW-restart-M-level-inv  $T$ > by blast
then obtain  $M1$   $M2$   $i$   $D$   $L$   $K$  where

```

```

confl-T': conflicting T' = Some D and
LD: L ∈ # D and
M1-M2: (Decided K # M1, M2) ∈ set (get-all-ann-decomposition (trail T')) and
get-level (trail T') K = i+1
get-level (trail T') L = backtrack-lvl T' and
get-level (trail T') L = get-maximum-level (trail T') D and
get-maximum-level (trail T') (remove1-mset L D) = i and
U: U ~ cons-trail (Propagated L D)
  (reduce-trail-to M1
    (add-learned-cls D
      (update-backtrack-lvl i
        (update-conflicting None T')))))
using bt by (auto elim: backtrackE)
have [simp]: clauses S = clauses T
  using confl by (auto elim: rulesE)
have [simp]: clauses T = clauses T'
  using s-or-r
proof (induction)
  case base
  then show ?case by simp
next
  case (step U V) note st = this(1) and s-o-r = this(2) and IH = this(3)
  have clauses U = clauses V
    using s-o-r by (auto simp: skip-or-resolve.simps elim: rulesE)
  then show ?case using IH by auto
qed
have inv-T: cdclW-restart-all-struct-inv T
  by (meson cdclW-cp.simps confl inv r-into-rtrancplp rtrancplp-cdclW-restart-all-struct-inv-inv
    rtrancplp-cdclW-cp-rtrancplp-cdclW-restart)
have cdclW-restart** T T'
  using rtrancplp-skip-or-resolve-rtrancplp-cdclW-restart s-or-r by blast
have inv-T': cdclW-restart-all-struct-inv T'
  using ⟨cdclW-restart** T T'⟩ inv-T rtrancplp-cdclW-restart-all-struct-inv-inv by blast
have inv-U: cdclW-restart-all-struct-inv U
  using cdclW-merge-restart-cdclW-restart confl fw-r-conflict inv local.bj
    rtrancplp-cdclW-restart-all-struct-inv-inv by blast

have [simp]: init-clss S = init-clss T'
  using ⟨cdclW-restart** T T'⟩ cdclW-restart-init-clss confl cdclW-restart-all-struct-inv-def conflict
inv
  by (metis ⟨cdclW-restart-M-level-inv T⟩ rtrancplp-cdclW-restart-init-clss)
then have atm-L: atm-of L ∈ atms-of-mm (clauses S)
  using inv-T' confl-T' LD unfolding cdclW-restart-all-struct-inv-def no-strange-atm-def
    clauses-def
  by (simp add: atms-of-def image-subset-iff)
obtain M where tr-T: trail T = M @ trail T'
  using s-or-r skip-or-resolve-state-change by meson
obtain M' where
  tr-T': trail T' = M' @ Decided K # tl (trail U) and
  tr-U: trail U = Propagated L D # tl (trail U)
  using U M1-M2 inv-T' unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def
    by fastforce
def M'' ≡ M @ M'
have tr-T: trail S = M'' @ Decided K # tl (trail U)
  using tr-T tr-T' confl unfolding M''-def by (auto elim: rulesE)
have init-clss T' + learned-clss S ⊨pm D

```

```

using inv-T' confl-T' unfolding cdclW-restart-all-struct-inv-def cdclW-restart-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 @ [Decided K]])
using confl M1-M2 (trail T = M @ trail T')
apply (auto dest!: get-all-ann-decomposition-exists-prepend
  elim!: conflictE)
by (rule sym) auto
ultimately have [simp]: trail (reduce-trail-toNOT M1 S) = M1
using M1-M2 confl by (subst reduce-trail-toNOT-reduce-trail-convert)
(auto simp: comp-def elim: rulesE)
have every-mark-is-a-conflict U
using inv-U unfolding cdclW-restart-all-struct-inv-def cdclW-restart-conflicting-def by simp
then have U-D: tl (trail U)  $\models_{as}$  CNot (remove1-mset L D)
by (metis append-self-conv2 tr-U)
have undef-L: undefined-lit (tl (trail U)) L
using U M1-M2 inv-U unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def
by (auto simp: lits-of-def defined-lit-map)
have backjump-l S U
apply (rule backjump-l[of - - - - L D - remove1-mset L D])
using tr-T apply simp
using inv unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def
apply (simp add: comp-def)
using U M1-M2 confl M1-M2 inv-T' inv unfolding cdclW-restart-all-struct-inv-def
cdclW-restart-M-level-inv-def apply (auto simp: state-eqNOT-def
  trail-reduce-trail-toNOT-add-learned-cls)[]
using CS apply auto[]
using tr-S-CS apply simp

using undef-L apply auto[]
using atm-L apply (simp add: trail-reduce-trail-toNOT-add-learned-cls)
using (init-cls T' + learned-cls S  $\models_{pm}$  D) LD unfolding clauses-def
apply simp
using LD apply simp
apply (metis U-D convert-trail-from-W-true-annots)
using inv-T' inv-U U confl-T' undef-L M1-M2 LD unfolding cdclW-restart-all-struct-inv-def
distinct-cdclW-restart-state-def by (simp add: cdclW-restart-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}-restart* **where**

cdcl_{NOT}-restart \equiv *restart-ops.cdcl_{NOT}-raw-restart* *cdcl_{NOT} restart*

lemma *cdcl_W-merge-restart-is-cdcl_{NOT}-merged-bj-learn-restart-no-step*:

assumes

inv: *cdcl_W-restart-all-struct-inv* *S* **and**

cdcl_W-restart: *cdcl_W-merge-restart* *S T*

shows *cdcl_{NOT}-restart*** *S T* \vee (*no-step cdcl_W-merge* *T* \wedge *conflicting* *T* \neq *None*)

proof –

consider

(*fw*) *cdcl_W-merge* *S T*

| (*fw-r*) *restart* *S T*

```

    using cdclW-restart by (meson cdclW-merge-restart.simps cdclW-rf.cases fw-conflict fw-decide
fw-forget
    fw-propagate)
  then show ?thesis
  proof cases
    case fw
    then have IH: cdclNOT-merged-bj-learn S T  $\vee$  (no-step cdclW-merge T  $\wedge$  conflicting T  $\neq$  None)
      using inv cdclW-merge-is-cdclNOT-merged-bj-learn by blast
    have invS: invNOT S
      using inv unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def by auto
    have ff2: cdclNOT++ S T  $\longrightarrow$  cdclNOT** S T
      by (meson tranclp-into-rtranclp)
    have ff3: no-dup (convert-trail-from-W (trail S))
      using invS by (simp add: comp-def)
    have cdclNOT  $\leq$  cdclNOT-restart
      by (auto simp: restart-ops.cdclNOT-raw-restart.simps)
    then show ?thesis
      using ff3 ff2 IH cdclNOT-merged-bj-learn-is-tranclp-cdclNOT
      rtranclp-mono[of cdclNOT cdclNOT-restart] invS predicate2D by blast
  next
    case fw-r
    then show ?thesis by (blast intro: restart-ops.cdclNOT-raw-restart.intros)
  qed
qed

```

abbreviation $\mu_{FW} :: 'st \Rightarrow nat$ **where**

$\mu_{FW} S \equiv$ (if no-step cdcl_W-merge S then 0 else $1 + \mu_{CDCL}$ -merged (set-mset (init-clss S)) S)

lemma cdcl_W-merge- μ_{FW} -decreasing:

assumes

inv: cdcl_W-restart-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: atm_s-of-mm (clauses S) \subseteq atm_s-of-mm ?A

using inv unfolding cdcl_W-restart-all-struct-inv-def no-strange-atm-def clauses-def by auto

have atm-trail: atm_s-of ' lits-of-l (trail S) \subseteq atm_s-of-mm ?A

using inv unfolding cdcl_W-restart-all-struct-inv-def no-strange-atm-def clauses-def by auto

have n-d: no-dup (trail S)

using inv unfolding cdcl_W-restart-all-struct-inv-def by (auto simp: cdcl_W-restart-M-level-inv-decomp)

have [simp]: \neg no-step cdcl_W-merge S

using fw by auto

have [simp]: init-clss S = init-clss T

using cdcl_W-merge-restart-cdcl_W-restart[of S T] inv rtranclp-cdcl_W-restart-init-clss

unfolding cdcl_W-restart-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, of T] atm-trail n-d

```

    by (auto split: if-split simp: comp-def image-image lits-of-def)
next
  case  $n\text{-}s$ 
  then show ?thesis by simp
qed
qed

```

```

lemma wf-cdclW-merge: wf {(T, S). cdclW-restart-all-struct-inv S ∧ cdclW-merge S T}
  apply (rule wfP-if-measure[of - - μFW])
  using cdclW-merge-μFW-decreasing by blast

```

```

sublocale conflict-driven-clause-learningW-termination
  by unfold-locales (simp add: wf-cdclW-merge)

```

6.3.4 Correctness of $cdcl_W\text{-merge-stgy}$

```

lemma full-cdclW-s'-full-cdclW-merge-restart:

```

```

  assumes
    conflicting R = None and
    inv: cdclW-restart-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-restart-all-struct-inv V
  using ⟨cdclW-s*** R V⟩ inv rtranclp-cdclW-restart-all-struct-inv-inv rtranclp-cdclW-s'-rtranclp-cdclW-restart
  by blast
  then have  $n\text{-}s$ : 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  $n\text{-}s\text{-}bj$ : no-step cdclW-bj V
  by (metis ⟨cdclW-restart-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ bj full-def
    n-step-cdclW-stgy-iff-no-step-cdclW-restart-cl-cdclW-o)
  have  $n\text{-}s\text{-}cp$ : no-step cdclW-merge-cp V

```

```

  proof -

```

```

    { fix ss :: 'st
      obtain ssa :: 'st  $\Rightarrow$  'st where
        ff1:  $\forall s. \neg$  cdclW-restart-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-restart-all-struct-inv V⟩ ⟨full cdclW-s' R V⟩ cdclW-s'.simps
        cdclW-s'-without-decide.cases) }
    then show ?thesis
      by blast
  qed

```

```

consider

```

```

  (fw-no-conf) cdclW-merge-stgy** R V and conflicting V = None
| (fw-conf) cdclW-merge-stgy** R V and conflicting V  $\neq$  None and no-step cdclW-bj V
| (fw-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
| (fw-dec-no-conf) 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-conf) cdclW-merge-cp** R V and conflicting V = None

```



```

| (cp-conf) U where  $cdcl_W\text{-merge-cp}^{**} R U$  and  $\text{conflict } U V$ 
using  $\text{rtrancp-cdcl}_W\text{-s'-no-step-cdcl}_W\text{-s'-without-decide-decomp-into-cdcl}_W\text{-merge}[OF$ 
 $\langle \text{cdcl}_W\text{-s}^{**} R V \rangle \text{ asms}]$  by auto
then show  $?fw$ 
proof cases
  case  $fw\text{-no-conf}$ 
    then show  $?thesis$  using  $n\text{-s}$  unfolding full-def by blast
  next
    case  $fw\text{-conf}$ 
      then show  $?thesis$  using  $n\text{-s}$  unfolding full-def by blast
  next
    case  $fw\text{-dec-conf}$ 
      have  $cdcl_W\text{-merge-cp } U V$ 
        using  $n\text{-s-bj}$  by (metis  $cdcl_W\text{-merge-cp.simps full-unfold fw-dec-conf}(5)$ )
      then have  $full1\ cdcl_W\text{-merge-cp } T V$ 
        unfolding full1-def by (metis  $fw\text{-dec-conf}(4) n\text{-s-cp trancp-unfold-end}$ )
      then have  $cdcl_W\text{-merge-stgy } S V$  using  $\langle \text{decide } S T \rangle \langle \text{no-step } cdcl_W\text{-merge-cp } S \rangle$  by auto
      then show  $?thesis$  using  $n\text{-s}$   $\langle \text{cdcl}_W\text{-merge-stgy}^{**} R S \rangle$  unfolding full-def by auto
  next
    case  $fw\text{-dec-no-conf}$ 
      then have  $full\ cdcl_W\text{-merge-cp } T V$ 
        using  $n\text{-s-cp}$  unfolding full-def by blast
      then have  $cdcl_W\text{-merge-stgy } S V$  using  $\langle \text{decide } S T \rangle \langle \text{no-step } cdcl_W\text{-merge-cp } S \rangle$  by auto
      then show  $?thesis$  using  $n\text{-s}$   $\langle \text{cdcl}_W\text{-merge-stgy}^{**} R S \rangle$  unfolding full-def by auto
  next
    case  $cp\text{-no-conf}$ 
      then have  $full\ cdcl_W\text{-merge-cp } R V$ 
        by (simp add: full-def n-s-cp)
      then have  $R = V \vee \text{cdcl}_W\text{-merge-stgy}^{++} R V$ 
        using  $fw\text{-s-cp}$  unfolding full-unfold fw-s-cp
        by (metis (no-types) rtrancp-unfold trancp-unfold-end)
      then show  $?thesis$ 
        by (simp add: full-def n-s rtrancp-unfold)
  next
    case  $cp\text{-conf}$ 
      have  $full\ cdcl_W\text{-bj } V V$ 
        using  $n\text{-s-bj}$  unfolding full-def by blast
      then have  $full1\ cdcl_W\text{-merge-cp } R V$ 
        unfolding full1-def by (meson  $cdcl_W\text{-merge-cp.conflict' cp-conf}(1,2) n\text{-s-cp}$ 
 $\text{rtrancp-into-trancp1}$ )
      then show  $?thesis$  using  $n\text{-s}$  unfolding full-def by auto
  qed
next
  assume  $?fw$ 
  then have  $cdcl_W\text{-restart}^{**} R V$  using  $\text{rtrancp-mono}[of\ cdcl_W\text{-merge-stgy } cdcl_W\text{-restart}^{**}]$ 
 $\text{cdcl}_W\text{-merge-stgy-rtrancp-cdcl}_W\text{-restart}$  unfolding full-def by auto
  then have  $inv': cdcl_W\text{-restart-all-struct-inv } V$  using  $inv\ \text{rtrancp-cdcl}_W\text{-restart-all-struct-inv-inv}$  by
blast
  have  $cdcl_W\text{-s}^{**} R V$ 
    using  $\langle ?fw \rangle$  by (simp add: full-def inv rtrancp-cdcl}_W\text{-merge-stgy-rtrancp-cdcl}_W\text{-s}')
  moreover have  $\text{no-step } cdcl_W\text{-s}' V$ 
  proof cases
    assume  $\text{conflicting } V = \text{None}$ 
    then show  $?thesis$ 
      by (metis  $inv' \langle full\ cdcl_W\text{-merge-stgy } R V \rangle \text{full-def}$ 
 $\text{no-step-cdcl}_W\text{-merge-stgy-no-step-cdcl}_W\text{-s}'$ )

```

```

next
  assume confl-V: conflicting V ≠ None
  then have no-step cdclW-bj V
  using rtrancpl-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 elim: rulesE)
qed
ultimately show ?s' unfolding full-def by blast
qed

lemma full-cdclW-stgy-full-cdclW-merge:
  assumes
    conflicting R = None and
    cdclW-restart-all-struct-inv R
  shows full cdclW-stgy R V  $\longleftrightarrow$  full cdclW-merge-stgy R V
  by (simp add: assms full-cdclW-s'-full-cdclW-merge-restart full-cdclW-stgy-iff-full-cdclW-s')

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 {#} ∧ unsatisfiable (set-mset N))
    ∨ (conflicting S' = None ∧ trail S' ⊨asm N ∧ satisfiable (set-mset N))
proof -
  have cdclW-restart-all-struct-inv (init-state N)
  using no-d unfolding cdclW-restart-all-struct-inv-def by auto
  moreover have conflicting (init-state N) = None
  by auto
  ultimately show ?thesis
  using full full-cdclW-stgy-final-state-conclusive-from-init-state
    full-cdclW-stgy-full-cdclW-merge no-d by presburger
qed
end

end
theory CDCL-W-Restart
imports CDCL-W-Merge
begin

```

6.3.5 Adding Restarts

```

locale cdclW-restart-restart =
  conflict-driven-clause-learningW
  — functions for the state:
  — access functions:
  trail init-clss learned-clss backtrack-lvl conflicting
  — changing state:
  cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
  update-conflicting

  — get state:
  init-state
for
  trail :: 'st ⇒ ('v, 'v clause) ann-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, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-learned-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clss :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

init-state :: 'v clauses  $\Rightarrow$  'st +
fixes f :: nat  $\Rightarrow$  nat
assumes f: unbounded f
begin

```

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

inductive *cdcl_W-merge-with-restart* **where**

restart-step:

```

(cdclW-merge-stgy  $\sim$  (card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)))) S T
 $\implies$  card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)) > f n
 $\implies$  restart T U  $\implies$  cdclW-merge-with-restart (S, n) (U, Suc n) |

```

restart-full: full1 cdcl_W-merge-stgy S T \implies cdcl_W-merge-with-restart (S, n) (T, Suc n)

lemma *cdcl_W-merge-with-restart* S T \implies cdcl_W-merge-restart** (fst S) (fst T)

by (induction rule: *cdcl_W-merge-with-restart.induct*)

```

(auto dest!: relpowp-imp-rtranclp cdclW-merge-stgy-tranclp-cdclW-merge tranclp-into-rtranclp
  rtranclp-cdclW-merge-stgy-rtranclp-cdclW-merge rtranclp-cdclW-merge-tranclp-cdclW-merge-restart
  fw-r-rf cdclW-rf.restart
  simp: full1-def)

```

lemma *cdcl_W-merge-with-restart-rtranclp-cdcl_W-restart*:

cdcl_W-merge-with-restart S T \implies cdcl_W-restart** (fst S) (fst T)

by (induction rule: *cdcl_W-merge-with-restart.induct*)

```

(auto dest!: relpowp-imp-rtranclp rtranclp-cdclW-merge-stgy-rtranclp-cdclW-restart cdclW-restart.rf
  cdclW-rf.restart tranclp-into-rtranclp simp: full1-def)

```

lemma *cdcl_W-merge-with-restart-increasing-number*:

cdcl_W-merge-with-restart S T \implies snd T = 1 + snd S

by (induction rule: *cdcl_W-merge-with-restart.induct*) auto

lemma full1 cdcl_W-merge-stgy S T \implies cdcl_W-merge-with-restart (S, n) (T, Suc n)

using *restart-full* **by** blast

lemma *cdcl_W-restart-all-struct-inv-learned-clss-bound*:

assumes inv: cdcl_W-restart-all-struct-inv S

shows set-mset (learned-clss S) \subseteq simple-clss (atms-of-mm (init-clss S))

proof

fix C

assume C: C \in set-mset (learned-clss S)

have distinct-mset C

using C inv **unfolding** cdcl_W-restart-all-struct-inv-def distinct-cdcl_W-restart-state-def distinct-mset-set-def

```

  by auto
moreover have  $\neg$ tautology  $C$ 
  using  $C$  inv unfolding cdclW-restart-all-struct-inv-def cdclW-restart-learned-clause-def by auto
moreover
  have atms-of  $C \subseteq$  atms-of-mm (learned-clss  $S$ )
    using  $C$  by auto
  then have atms-of  $C \subseteq$  atms-of-mm (init-clss  $S$ )
    using inv unfolding cdclW-restart-all-struct-inv-def no-strange-atm-def by force
moreover have finite (atms-of-mm (init-clss  $S$ ))
  using inv unfolding cdclW-restart-all-struct-inv-def by auto
ultimately show  $C \in$  simple-clss (atms-of-mm (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:*
 $cdcl_W\text{-merge-with-restart } S \ T \implies cdcl_W\text{-restart-}M\text{-level-inv } (fst \ S) \implies$
 $init-clss \ (fst \ S) = init-clss \ (fst \ T)$
 using *cdcl_W-merge-with-restart-rtrancpl-cdcl_W-restart rtrancpl-cdcl_W-restart-init-clss* by blast

lemma

```

  wf {(T, S). cdclW-restart-all-struct-inv (fst S) ∧ cdclW-merge-with-restart S T}
proof (rule ccontr)
  assume  $\neg$  ?thesis
  then obtain  $g$  where
     $g: \bigwedge i. cdcl_W\text{-merge-with-restart } (g \ i) \ (g \ (Suc \ i))$  and
     $inv: \bigwedge i. cdcl_W\text{-restart-all-struct-inv } (fst \ (g \ i))$ 
  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-restart-all-struct-inv-def by (metis cdclW-merge-with-restart-init-clss)
    } note init-g = this
  let ?S = g 0
  have finite (atms-of-mm (init-clss (fst ?S)))
    using inv unfolding cdclW-restart-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-merge-with-restart-increasing-number g)
  then have snd-g-0:  $\bigwedge i. i > 0 \implies snd \ (g \ i) = i + snd \ (g \ 0)$ 
    by blast
  have unbounded-f-g: unbounded ( $\lambda i. f \ (snd \ (g \ i))$ )
    using f unfolding bounded-def by (metis add.commute f less-or-eq-imp-le snd-g
      not-bounded-nat-exists-larger not-le le-iff-add)

  obtain  $k$  where
     $f\text{-}g\text{-}k: f \ (snd \ (g \ k)) > card \ (simple-clss \ (atms-of-mm \ (init-clss \ (fst \ ?S))))$  and
     $k > card \ (simple-clss \ (atms-of-mm \ (init-clss \ (fst \ ?S))))$ 
  using not-bounded-nat-exists-larger[OF unbounded-f-g] by blast

```

The following does not hold anymore with the non-strict version of cardinality in the definition.

```

{ fix  $i$ 
  assume no-step cdclW-merge-stgy (fst (g i))
  with g[of i]

```

```

have False
  proof (induction rule: cdclW-merge-with-restart.induct)
    case (restart-step T S n) note H = this(1) and c = this(2) and n-s = this(4)
    obtain S' where cdclW-merge-stgy S S'
      using H c by (metis gr-implies-not0 relpoup-E2)
    then show False using n-s by auto
  next
    case (restart-full S T)
    then show False unfolding full1-def by (auto dest: tranclpD)
  qed
} note H = this
obtain m T where
  m: m = card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k)))) and
  m > f (snd (g k)) and
  restart T (fst (g (k+1))) and
  cdclW-merge-stgy: (cdclW-merge-stgy  $\sim$  m) (fst (g k)) T
  using g[of k] H[of Suc k] by (force simp: cdclW-merge-with-restart.simps full1-def)
have cdclW-merge-stgy** (fst (g k)) T
  using cdclW-merge-stgy relpoup-imp-rtranclp by metis
then have cdclW-restart-all-struct-inv T
  using inv[of k] rtranclp-cdclW-restart-all-struct-inv-inv rtranclp-cdclW-merge-stgy-rtranclp-cdclW-restart
  by blast
moreover have card (set-mset (learned-clss T)) - card (set-mset (learned-clss (fst (g k))))
  > card (simple-clss (atms-of-mm (init-clss (fst ?S))))
  unfolding m[symmetric] using m > f (snd (g k)) f-g-k by linarith
then have card (set-mset (learned-clss T))
  > card (simple-clss (atms-of-mm (init-clss (fst ?S))))
  by linarith
moreover
  have init-clss (fst (g k)) = init-clss T
    using cdclW-merge-stgy** (fst (g k)) T rtranclp-cdclW-merge-stgy-rtranclp-cdclW-restart
    rtranclp-cdclW-restart-init-clss inv unfolding cdclW-restart-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-restart-all-struct-inv-learned-clss-bound
  by (simp add: finite (atms-of-mm (init-clss (fst (g 0)))) simple-clss-finite
    card-mono leD)
qed

lemma cdclW-merge-with-restart-distinct-mset-clauses:
  assumes invR: cdclW-restart-all-struct-inv (fst R) and
  st: cdclW-merge-with-restart R S and
  dist: distinct-mset (clauses (fst R)) and
  R: trail (fst R) = []
  shows distinct-mset (clauses (fst S))
  using asms(2,1,3,4)
proof (induction)
  case (restart-full S T)
  then show ?case using rtranclp-cdclW-merge-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-merge-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: relpoup-imp-rtranclp)

```

then show *?case using (restart T U) unfolding clauses-def*
by (*metis distinct-mset-union fstI restartE subset-mset.le-iff-add union-assoc*)
qed

inductive *cdcl_W-restart-with-restart* **where**

restart-step:

(*cdcl_W-stgy* \sim (card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)))) *S T* \implies
 card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)) > *f n* \implies
restart T U \implies
cdcl_W-restart-with-restart (*S, n*) (*U, Suc n*) |

restart-full: *full1 cdcl_W-stgy S T* \implies *cdcl_W-restart-with-restart* (*S, n*) (*T, Suc n*)

lemma *cdcl_W-restart-with-restart-rtranclp-cdcl_W-restart*:

cdcl_W-restart-with-restart S T \implies *cdcl_W-restart*** (*fst S*) (*fst T*)

apply (*induction rule: cdcl_W-restart-with-restart.induct*)

by (*auto dest!: relpoup-imp-rtranclp tranclp-into-rtranclp fw-r-rf*

cdcl_W-rf.restart rtranclp-cdcl_W-stgy-rtranclp-cdcl_W-restart cdcl_W-merge-restart-cdcl_W-restart
simp: full1-def)

lemma *cdcl_W-restart-with-restart-increasing-number*:

cdcl_W-restart-with-restart S T \implies *snd T* = 1 + *snd S*

by (*induction rule: cdcl_W-restart-with-restart.induct*) *auto*

lemma *full1 cdcl_W-stgy S T* \implies *cdcl_W-restart-with-restart* (*S, n*) (*T, Suc n*)

using *restart-full* **by** *blast*

lemma *cdcl_W-restart-with-restart-init-clss*:

cdcl_W-restart-with-restart S T \implies *cdcl_W-restart-M-level-inv* (*fst S*) \implies *init-clss* (*fst S*) = *init-clss* (*fst T*)

using *cdcl_W-restart-with-restart-rtranclp-cdcl_W-restart rtranclp-cdcl_W-restart-init-clss* **by** *blast*

lemma

wf {(*T, S*). *cdcl_W-restart-all-struct-inv* (*fst S*) \wedge *cdcl_W-restart-with-restart S T*}

proof (*rule ccontr*)

assume \neg *?thesis*

then obtain *g* **where**

g: $\bigwedge i. \text{cdcl}_W\text{-restart-with-restart } (g\ i) (g\ (\text{Suc } i))$ **and**

inv: $\bigwedge i. \text{cdcl}_W\text{-restart-all-struct-inv } (\text{fst } (g\ i))$

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-restart-all-struct-inv-def* **by** (*metis cdcl_W-restart-with-restart-init-clss*)

} **note** *init-g* = *this*

let *?S* = *g 0*

have *finite* (*atms-of-mm* (*init-clss* (*fst ?S*)))

using *inv unfolding cdcl_W-restart-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-restart-with-restart-increasing-number g*)

then have *snd-g-0*: $\bigwedge i. i > 0 \implies \text{snd } (g\ i) = i + \text{snd } (g\ 0)$

by *blast*

have *unbounded-f-g*: *unbounded* ($\lambda i. f\ (\text{snd } (g\ i))$)

using *f unfolding bounded-def* **by** (*metis add commute f less-or-eq-imp-le snd-g*)

not-bounded-nat-exists-larger not-le le-iff-add)

obtain k **where**

$f\text{-}g\text{-}k$: $f \text{ (snd } (g \ k)) > \text{card } (\text{simple-clss } (\text{atms-of-mm } (\text{init-clss } (\text{fst } ?S))))$ **and**
 $k > \text{card } (\text{simple-clss } (\text{atms-of-mm } (\text{init-clss } (\text{fst } ?S))))$
using *not-bounded-nat-exists-larger*[*OF unbounded-f-g*] **by** *blast*

The following does not hold anymore with the non-strict version of cardinality in the definition.

```
{ fix i
  assume no-step cdclW-stgy (fst (g i))
  with g[of i]
  have False
    proof (induction rule: cdclW-restart-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
   $m$ :  $m = \text{card } (\text{set-mset } (\text{learned-clss } T)) - \text{card } (\text{set-mset } (\text{learned-clss } (\text{fst } (g \ k))))$  and
   $m > f \text{ (snd } (g \ k))$  and
  restart  $T$  (fst (g (k+1))) and
  cdclW-merge-stgy: (cdclW-stgy  $\sim m$ ) (fst (g k))  $T$ 
  using  $g$ [of  $k$ ]  $H$ [of  $\text{Suc } k$ ] by (force simp: cdclW-restart-with-restart.simps full1-def)
have cdclW-stgy** (fst (g k))  $T$ 
  using cdclW-merge-stgy relpowp-imp-rtranclp by metis
then have cdclW-restart-all-struct-inv  $T$ 
  using inv[of  $k$ ] rtranclp-cdclW-restart-all-struct-inv-inv rtranclp-cdclW-stgy-rtranclp-cdclW-restart
by blast
moreover have card (set-mset (learned-clss  $T$ )) - card (set-mset (learned-clss (fst (g k))))
  > card (simple-clss (atms-of-mm (init-clss (fst ?S))))
  unfolding  $m$ [symmetric] using  $\langle m > f \text{ (snd } (g \ k)) \rangle$   $f\text{-}g\text{-}k$  by linarith
then have card (set-mset (learned-clss  $T$ ))
  > card (simple-clss (atms-of-mm (init-clss (fst ?S))))
  by linarith
moreover
  have init-clss (fst (g k)) = init-clss  $T$ 
  using  $\langle \text{cdcl}_W\text{-stgy}^{**} \text{ (fst } (g \ k)) \ T \rangle$  rtranclp-cdclW-stgy-rtranclp-cdclW-restart rtranclp-cdclW-restart-init-clss
  inv unfolding cdclW-restart-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-restart-all-struct-inv-learned-clss-bound
  by (simp add:  $\langle \text{finite } (\text{atms-of-mm } (\text{init-clss } (\text{fst } (g \ 0)))) \rangle$  simple-clss-finite
    card-mono leD)
qed
```

lemma *cdcl_W-restart-with-restart-distinct-mset-clauses*:

assumes *invR*: *cdcl_W-restart-all-struct-inv* (fst R) **and**
st: *cdcl_W-restart-with-restart* R S **and**
dist: *distinct-mset* (clauses (fst R)) **and**

```

R: trail (fst R) = []
shows distinct-mset (clauses (fst S))
using assms(2,1,3,4)
proof (induction)
  case (restart-full S T)
  then show ?case using rtrancpl-cdclW-stgy-distinct-mset-clauses[of S T] unfolding full1-def
    by (auto dest: trancpl-into-rtrancpl)
next
  case (restart-step T S n U)
  then have distinct-mset (clauses T) using rtrancpl-cdclW-stgy-distinct-mset-clauses[of S T]
    unfolding full1-def by (auto dest: relpowp-imp-rtrancpl)
  then show ?case using (restart T U) unfolding clauses-def
    by (metis distinct-mset-union fstI restartE subset-mset.le-iff-add union-assoc)
qed
end

locale luby-sequence =
  fixes ur :: nat
  assumes ur > 0
begin

lemma exists-luby-decomp:
  fixes i :: nat
  shows  $\exists k::nat. (2^{k-1} \leq i \wedge i < 2^k - 1) \vee i = 2^k - 1$ 
proof (induction i)
  case 0
  then show ?case
    by (rule exI[of - 0], simp)
next
  case (Suc n)
  then obtain k where  $2^{k-1} \leq n \wedge n < 2^k - 1 \vee n = 2^k - 1$ 
    by blast
  then consider
    (st-interv)  $2^{k-1} \leq n$  and  $n \leq 2^k - 2$ 
  | (end-interv)  $2^{k-1} \leq n$  and  $n = 2^k - 2$ 
  | (pow2)  $n = 2^k - 1$ 
  by linarith
  then show ?case
  proof cases
    case st-interv
    then show ?thesis apply - apply (rule exI[of - k])
      by (metis (no-types, lifting) One-nat-def Suc-diff-Suc Suc-lessI
         $2^{k-1} \leq n \wedge n < 2^k - 1 \vee n = 2^k - 1$  diff-self-eq-0
        dual-order.trans le-SucI le-imp-less-Suc numeral-2-eq-2 one-le-numeral
        one-le-power zero-less-numeral zero-less-power)
    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 $2^{\wedge}((\text{SOME } k. i = 2^k - 1) - 1)$

else *luby-sequence-core* ($i - 2^{\wedge}((\text{SOME } k. 2^{\wedge}(k-1) \leq i \wedge i < 2^k - 1) + 1)$))

by *auto*

termination

proof (*relation less-than*, *goal-cases*)

case 1

then show ?*case* **by** *auto*

next

case (2 *i*)

let ?*k* = *SOME* *k*. $2^{\wedge}(k - 1) \leq i \wedge i < 2^{\wedge}k - 1$

have $2^{\wedge}(\text{?k} - 1) \leq i \wedge i < 2^{\wedge}\text{?k} - 1$

apply (*rule someI-ex*)

using 2 *exists-luby-decomp* **by** *blast*

then show ?*case*

proof –

have $\forall n \text{ na. } \neg (1::\text{nat}) \leq n \vee 1 \leq n^{\wedge} \text{na}$

by (*meson one-le-power*)

then have *f1*: $(1::\text{nat}) \leq 2^{\wedge}(\text{?k} - 1)$

using *one-le-numeral* **by** *blast*

have *f2*: $i - 2^{\wedge}(\text{?k} - 1) + 2^{\wedge}(\text{?k} - 1) = i$

using $2^{\wedge}(\text{?k} - 1) \leq i \wedge i < 2^{\wedge}\text{?k} - 1$ *le-add-diff-inverse2* **by** *blast*

have *f3*: $2^{\wedge}\text{?k} - 1 \neq \text{Suc } 0$

using *f1* $2^{\wedge}(\text{?k} - 1) \leq i \wedge i < 2^{\wedge}\text{?k} - 1$ **by** *linarith*

have $2^{\wedge}\text{?k} - (1::\text{nat}) \neq 0$

using $2^{\wedge}(\text{?k} - 1) \leq i \wedge i < 2^{\wedge}\text{?k} - 1$ *gr-implies-not0* **by** *blast*

then have *f4*: $2^{\wedge}\text{?k} \neq (1::\text{nat})$

by *linarith*

have *f5*: $\forall n \text{ na. if } \text{na} = 0 \text{ then } (n::\text{nat})^{\wedge} \text{na} = 1 \text{ else } n^{\wedge} \text{na} = n * n^{\wedge} (\text{na} - 1)$

by (*simp add: power-eq-if*)

then have ?*k* $\neq 0$

using *f4* **by** *meson*

then have $2^{\wedge}(\text{?k} - 1) \neq \text{Suc } 0$

using *f5 f3* **by** *presburger*

then have $\text{Suc } 0 < 2^{\wedge}(\text{?k} - 1)$

using *f1* **by** *linarith*

then show ?*thesis*

using *f2 less-than-iff* **by** *presburger*

qed

qed

function *natlog2* :: *nat* \Rightarrow *nat* **where**

natlog2 *n* = (if *n* = 0 then 0 else 1 + *natlog2* (*n* div 2))

using *not0-implies-Suc* **by** *auto*

termination **by** (*relation measure* ($\lambda n. n$)) *auto*

declare *natlog2.simps*[*simp del*]

declare *luby-sequence-core.simps*[*simp del*]

lemma *two-pover-n-eq-two-power-n'-eq*:

assumes $H: (2::nat) \wedge (k::nat) - 1 = 2 \wedge k' - 1$
shows $k' = k$

proof –

have $(2::nat) \wedge (k::nat) = 2 \wedge k'$
using H **by** (*metis One-nat-def Suc-pred zero-less-numeral zero-less-power*)
then show *?thesis* **by** *simp*

qed

lemma *luby-sequence-core-two-power-minus-one*:

luby-sequence-core $(2^k - 1) = 2^{(k-1)}$ (**is** *?L* = *?K*)

proof –

have *decomp*: $\exists ka. 2^k - 1 = 2^{ka} - 1$
by *auto*
have $?L = 2^{((SOME\ k'. (2::nat) \wedge k - 1 = 2^{k'} - 1) - 1)}$
apply (*subst luby-sequence-core.simps, subst decomp*)
by *simp*
moreover have $(SOME\ k'. (2::nat) \wedge k - 1 = 2^{k'} - 1) = k$
apply (*rule some-equality*)
apply *simp*
using *two-pover-n-eq-two-power-n'-eq* **by** *blast*
ultimately show *?thesis* **by** *presburger*

qed

lemma *different-luby-decomposition-false*:

assumes
 $H: 2 \wedge (k - Suc\ 0) \leq i$ **and**
 $k': i < 2 \wedge k' - Suc\ 0$ **and**
 $k-k': k > k'$

shows *False*

proof –

have $2 \wedge k' - Suc\ 0 < 2 \wedge (k - Suc\ 0)$
using $k-k'$ *less-eq-Suc-le* **by** *auto*
then show *?thesis*
using $H\ k'$ **by** *linarith*

qed

lemma *luby-sequence-core-not-two-power-minus-one*:

assumes
 $k-i: 2 \wedge (k - 1) \leq i$ **and**
 $i-k: i < 2^k - 1$
shows *luby-sequence-core* $i = luby-sequence-core\ (i - 2 \wedge (k - 1) + 1)$

proof –

have $H: \neg (\exists ka. i = 2^{ka} - 1)$
proof (*rule ccontr*)
assume $\neg ?thesis$
then obtain $k': nat$ **where** $k': i = 2^{k'} - 1$ **by** *blast*
have $(2::nat) \wedge k' - 1 < 2^k - 1$
using $i-k$ **unfolding** k' .
then have $(2::nat) \wedge k' < 2^k$
by *linarith*
then have $k' < k$
by *simp*

```

have  $2^{\wedge} (k - 1) \leq 2^{\wedge} k' - (1::nat)$ 
  using k-i unfolding k' .
then have  $(2::nat)^{\wedge} (k-1) < 2^{\wedge} k'$ 
  by (metis Suc-diff-1 not-le not-less-eq zero-less-numeral zero-less-power)
then have  $k-1 < k'$ 
  by simp

show False using  $\langle k' < k \rangle \langle k-1 < k' \rangle$  by linarith
qed
have  $\bigwedge k k'. 2^{\wedge} (k - Suc\ 0) \leq i \implies i < 2^{\wedge} k - Suc\ 0 \implies 2^{\wedge} (k' - Suc\ 0) \leq i \implies$ 
 $i < 2^{\wedge} k' - Suc\ 0 \implies k = k'$ 
  by (meson different-luby-decomposition-false linorder-neqE-nat)
then have k:  $(SOME\ k. 2^{\wedge} (k - Suc\ 0) \leq i \wedge i < 2^{\wedge} k - Suc\ 0) = k$ 
  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  $(2^{\wedge}(b+1) - 1) = 2^{\wedge}b$ 
    using luby-sequence-core-two-power-minus-one[of b+1] by simp
  moreover have  $(2::nat)^{\wedge}b > b$ 
    by (induction b) auto
  ultimately show False using b[of  $2^{\wedge}(b+1) - 1$ ] by linarith
qed

abbreviation luby-sequence :: nat  $\Rightarrow$  nat where
luby-sequence n  $\equiv$  ur * luby-sequence-core n

lemma bounded-luby-sequence: unbounded luby-sequence
  using bounded-const-product[of ur] luby-sequence-axioms
  luby-sequence-def unbounded-luby-sequence-core by blast

lemma luby-sequence-core-0: luby-sequence-core 0 = 1
proof -
  have 0:  $(0::nat) = 2^{\wedge}0 - 1$ 
    by auto
  show ?thesis
    by (subst 0, subst luby-sequence-core-two-power-minus-one) simp
qed

lemma luby-sequence-core n  $\geq$  1
proof (induction n rule: nat-less-induct-case)
  case 0
  then show ?case by (simp add: luby-sequence-core-0)
next
  case (Suc n) note IH = this

  consider
    (interv) k where  $2^{\wedge} (k - 1) \leq Suc\ n$  and  $Suc\ n < 2^{\wedge} k - 1$ 

```

```

| (pow2) k where Suc n = 2 ^ k - Suc 0
using exists-luby-decomp[of Suc n] by auto

then show ?case
proof cases
  case pow2
  show ?thesis
    using luby-sequence-core-two-power-minus-one pow2 by auto
  next
  case interv
  have n: Suc n - 2 ^ (k - 1) + 1 < Suc n
  by (metis Suc-1 Suc-eq-plus1 add.commute add-diff-cancel-left' add-less-mono1 gr0I
    interv(1) interv(2) le-add-diff-inverse2 less-Suc-eq not-le power-0 power-one-right
    power-strict-increasing-iff)
  show ?thesis
  apply (subst luby-sequence-core-not-two-power-minus-one[OF interv])
  using IH n by auto
qed
qed
end

locale luby-sequence-restart =
  luby-sequence ur +
  conflict-driven-clause-learningW
  — functions for the state:
  — access functions:
  trail init-clss learned-clss backtrack-lvl conflicting
  — changing state:
  cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
  update-conflicting

  — get state:
  init-state
for
  ur :: nat and
  trail :: 'st ⇒ ('v, 'v clause) ann-lits and
  hd-trail :: 'st ⇒ ('v, 'v clause) ann-lit and
  init-clss :: 'st ⇒ 'v clauses and
  learned-clss :: 'st ⇒ 'v clauses and
  backtrack-lvl :: 'st ⇒ nat and
  conflicting :: 'st ⇒ 'v clause option and

  cons-trail :: ('v, 'v clause) ann-lit ⇒ 'st ⇒ 'st and
  tl-trail :: 'st ⇒ 'st and
  add-learned-cls :: 'v clause ⇒ 'st ⇒ 'st and
  remove-cls :: 'v clause ⇒ 'st ⇒ 'st and
  update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
  update-conflicting :: 'v clause option ⇒ 'st ⇒ 'st and

  init-state :: 'v clauses ⇒ 'st
begin

sublocale cdclW-restart-restart - - - - - luby-sequence
apply unfold-locales
using bounded-luby-sequence by blast

```

```

end
end
theory CDCL-W-Incremental
imports CDCL-W-Termination
begin

```

6.4 Incremental SAT solving

```

locale stateW-adding-init-clause =
  stateW
  — functions about the state:
  — getter:
  trail init-clss learned-clss backtrack-lvl conflicting
  — setter:
  cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
  update-conflicting

  — Some specific states:
  init-state
for
  trail :: 'st  $\Rightarrow$  ('v, 'v clause) ann-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, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-learned-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v clauses  $\Rightarrow$  'st +
fixes
  add-init-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st
assumes
  add-init-cls:
  state st = (M, N, U, S')  $\implies$ 
  state (add-init-cls C st) = (M, {#C#} + N, U, S')
begin

lemma
  trail-add-init-cl[simp]:
  trail (add-init-cls C st) = trail st and
  init-clss-add-init-cl[simp]:
  init-clss (add-init-cls C st) = {#C#} + init-clss st
  and
  learned-clss-add-init-cl[simp]:
  learned-clss (add-init-cls C st) = learned-clss st and
  backtrack-lvl-add-init-cl[simp]:
  backtrack-lvl (add-init-cls C st) = backtrack-lvl st and
  conflicting-add-init-cl[simp]:
  conflicting (add-init-cls C st) = conflicting st
using add-init-cls[of st - - - C] by (cases state st; auto)+

```

```

lemma clauses-add-init-cls[simp]:
  clauses (add-init-cls N S) = {#N#} + init-clss S + learned-clss S
  unfolding clauses-def by auto

lemma reduce-trail-to-add-init-cls[simp]:
  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 conflicting-add-init-cls-iff-conflicting[simp]:
  conflicting (add-init-cls C S) = None  $\longleftrightarrow$  conflicting S = None
  by fastforce+
end

locale conflict-driven-clause-learning-with-adding-init-clauseW =
  stateW-adding-init-clause

  — functions for the state:
  — access functions:
  trail init-clss learned-clss backtrack-lvl conflicting
  — changing state:
  cons-trail tl-trail add-learned-cls remove-cls update-backtrack-lvl
  update-conflicting

  — get state:
  init-state
  — Adding a clause:
  add-init-cls
for
  trail :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits and
  hd-trail :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lit 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, 'v clause) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-learned-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conflicting :: 'v clause option  $\Rightarrow$  'st  $\Rightarrow$  'st and

  init-state :: 'v clauses  $\Rightarrow$  'st and
  add-init-cls :: 'v clause  $\Rightarrow$  'st  $\Rightarrow$  'st
begin

sublocale conflict-driven-clause-learningW
  by unfold-locales

```

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl_W-restart-all-struct-inv*

definition *cdcl_W-stgy-invariant* **where**
cdcl_W-stgy-invariant S \longleftrightarrow
 conflict-is-false-with-level S
 \wedge no-clause-is-false S

\wedge *no-smaller-conf* S
 \wedge *no-clause-is-false* S

lemma *cdcl_W-stgy-cdcl_W-stgy-invariant*:

assumes

cdcl_W-restart: *cdcl_W-stgy* S T **and**

inv-s: *cdcl_W-stgy-invariant* S **and**

inv: *cdcl_W-restart-all-struct-inv* S

shows

cdcl_W-stgy-invariant T

unfolding *cdcl_W-stgy-invariant-def* *cdcl_W-restart-all-struct-inv-def* **apply** (*intro conjI*)

apply (*rule cdcl_W-stgy-ex-lit-of-max-level*[*of S*])

using *assms unfolding cdcl_W-stgy-invariant-def cdcl_W-restart-all-struct-inv-def* **apply** *auto*[7]

using *cdcl_W-restart cdcl_W-stgy-not-non-negated-init-clss* **apply** *simp*

apply (*rule cdcl_W-stgy-no-smaller-conf*-*inv*)

using *assms unfolding cdcl_W-stgy-invariant-def cdcl_W-restart-all-struct-inv-def* **apply** *auto*[4]

using *cdcl_W-restart cdcl_W-stgy-not-non-negated-init-clss* **by** *auto*

lemma *rtrancp-cdcl_W-stgy-cdcl_W-stgy-invariant*:

assumes

cdcl_W-restart: *cdcl_W-stgy*** S T **and**

inv-s: *cdcl_W-stgy-invariant* S **and**

inv: *cdcl_W-restart-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-restart-all-struct-inv-inv*

rtrancp-cdcl_W-stgy-rtrancp-cdcl_W-restart **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 (*Decided* $L \ \# \ M$) $S =$

(*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 *clause* \Rightarrow ' $st \Rightarrow$ ' st **where**

add-new-clause-and-update $C \ S =$

(*if* *trail* $S \models_{as} CNot \ C$

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*:
 $\exists M. \text{trail } S = M @ \text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } S) S)$
proof (*induction trail S arbitrary: S rule: ann-lit-list-induct*)
case *Nil*
then show ?*case* **by** *simp*
next
case (*Decided 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 (*Propagated 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**
 $M: \text{trail } T = M @ \text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)$
using *trail-cut-trail-wrt-clause[of T C]* **by** *auto*
show ?*thesis*
using *n-d unfolding arg-cong[OF M, of no-dup]* **by** *auto*
qed

lemma *cut-trail-wrt-clause-backtrack-lvl-length-decided*:
assumes
 $\text{backtrack-lvl } T = \text{count-decided } (\text{trail } T)$
shows
 $\text{backtrack-lvl } (\text{cut-trail-wrt-clause } C (\text{trail } T) T) =$
 $\text{count-decided } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))$
using *assms*
proof (*induction trail T arbitrary: T rule: ann-lit-list-induct*)
case *Nil*
then show ?*case* **by** *simp*
next
case (*Decided 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 (*Propagated 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-CNot-trail*:
assumes $\text{trail } T \models_{\text{as}} C \text{Not } C$
shows
 $(\text{trail } ((\text{cut-trail-wrt-clause } C (\text{trail } T) T))) \models_{\text{as}} C \text{Not } C$


```

using assms
proof (induction trail T arbitrary:T rule: ann-lit-list-induct)
  case Nil
  then show ?case by simp
next
case (Decided L M) note IH = this(1)[of decr-bt-lvl (tl-trail T)] and M = this(2)[symmetric]
  and bt = this(3)
show ?case
proof (cases count C (-L) = 0)
  case False
  then show ?thesis
    using IH M bt by (auto simp: true-annots-true-cls)
next
case True
  obtain mma :: 'v clause where
    f6: (mma ∈ {{#- l#} | l. l ∈# C} → M ⊨a mma) → M ⊨as {{#- l#} | l. l ∈# C}
    using true-annots-def by blast
  have mma ∈ {{#- l#} | l. l ∈# C} → trail T ⊨a mma
    using CNot-def M bt by (metis (no-types) true-annots-def)
  then have M ⊨as {{#- l#} | l. l ∈# C}
    using f6 True M bt by (force simp: count-eq-zero-iff)
  then show ?thesis
    using IH true-annots-true-cls M by (auto simp: CNot-def)
qed
next
case (Propagated L l M) note IH = this(1)[of tl-trail T] and M = this(2)[symmetric] and bt =
this(3)
show ?case
proof (cases count C (-L) = 0)
  case False
  then show ?thesis
    using IH M bt by (auto simp: true-annots-true-cls)
next
case True
  obtain mma :: 'v clause where
    f6: (mma ∈ {{#- l#} | l. l ∈# C} → M ⊨a mma) → M ⊨as {{#- l#} | l. l ∈# C}
    using true-annots-def by blast
  have mma ∈ {{#- l#} | l. l ∈# C} → trail T ⊨a mma
    using CNot-def M bt by (metis (no-types) true-annots-def)
  then have M ⊨as {{#- l#} | l. l ∈# C}
    using f6 True M bt by (force simp: count-eq-zero-iff)
  then show ?thesis
    using IH true-annots-true-cls M by (auto simp: CNot-def)
qed
qed

lemma cut-trail-wrt-clause-hd-trail-in-or-empty-trail:
  ((∀ L ∈# C. -L ∉ lits-of-l (trail T)) ∧ trail (cut-trail-wrt-clause C (trail T) T) = [])
  ∨ (-lit-of (hd (trail (cut-trail-wrt-clause C (trail T) T))) ∈# C
    ∧ length (trail (cut-trail-wrt-clause C (trail T) T)) ≥ 1)
proof (induction trail T arbitrary:T rule: ann-lit-list-induct)
  case Nil
  then show ?case by simp
next
case (Decided 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 (*Propagated L l M*) **note** $IH = \text{this}(1)[\text{of } \text{tl-trail } T]$ **and** $M = \text{this}(2)[\text{symmetric}]$
then show $?case$ **by** *simp force*

qed

We can fully run *cdcl_W-restart-s* or add a clause. Remark that we use *cdcl_W-restart-s* to avoid an explicit *skip*, *resolve*, and *backtrack* normalisation to get rid of the conflict *C* if possible.

inductive *incremental-cdcl_W-restart* :: '*st* ⇒ '*st* ⇒ *bool* **for** *S* **where**

add-conf:

$\text{trail } S \models_{asm} \text{init-clss } S \implies \text{distinct-mset } C \implies \text{conflicting } S = \text{None} \implies$
 $\text{trail } S \models_{as} CNot \ C \implies$
 $\text{full } cdcl_W\text{-stgy}$
 $(\text{update-conflicting } (\text{Some } C)$
 $(\text{add-init-clss } C (\text{cut-trail-wrt-clause } C (\text{trail } S) S))) \ T \implies$
 $\text{incremental-cdcl}_W\text{-restart } S \ T \mid$

add-no-conf:

$\text{trail } S \models_{asm} \text{init-clss } S \implies \text{distinct-mset } C \implies \text{conflicting } S = \text{None} \implies$
 $\neg \text{trail } S \models_{as} CNot \ C \implies$
 $\text{full } cdcl_W\text{-stgy } (\text{add-init-clss } C \ S) \ T \implies$
 $\text{incremental-cdcl}_W\text{-restart } S \ T$

lemma *cdcl_W-restart-all-struct-inv-add-new-clause-and-update-cdcl_W-restart-all-struct-inv*:

assumes

inv-T: *cdcl_W-restart-all-struct-inv T* **and**

tr-T-N[simp]: $\text{trail } T \models_{asm} N$ **and**

tr-C[simp]: $\text{trail } T \models_{as} CNot \ C$ **and**

[*simp*]: *distinct-mset C*

shows *cdcl_W-restart-all-struct-inv (add-new-clause-and-update C T) (is cdcl_W-restart-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: $\text{trail } T = M @ \text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)$

using *trail-cut-trail-wrt-clause[of T C]* **by** *blast*

have *H[dest]*: $\bigwedge x. x \in \text{lits-of-l } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)) \implies$
 $x \in \text{lits-of-l } (\text{trail } T)$

using *inv-T arg-cong[OF M, of lits-of-l]* **by** *auto*

have *H'[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-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 cdcl_W-restart-all-struct-inv-def no-strange-atm-def add-new-clause-and-update-def*
cdcl_W-restart-M-level-inv-def **by** (*auto 20 1*)

have *M-lev*: *cdcl_W-restart-M-level-inv T*

using *inv-T unfolding cdcl_W-restart-all-struct-inv-def* **by** *blast*

then have *no-dup* ($M @ \text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)$)

unfolding *cdcl_W-restart-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-l (M @ trail (cut-trail-wrt-clause C (trail T) T)))
  using M-lev unfolding cdclW-restart-M-level-inv-def unfolding M[symmetric] by auto
then have [simp]: consistent-interp (lits-of-l (trail (cut-trail-wrt-clause C
  (trail T) T)))
  unfolding consistent-interp-def by auto

have [simp]: cdclW-restart-M-level-inv ?T
  using M-lev unfolding cdclW-restart-M-level-inv-def by (auto dest: H H'
  simp: M-lev cdclW-restart-M-level-inv-def cut-trail-wrt-clause-backtrack-lvl-length-decided)

have [simp]:  $\bigwedge s. s \in \# \text{ learned-clss } T \implies \neg \text{tautology } s$ 
  using inv-T unfolding cdclW-restart-all-struct-inv-def by auto

have distinct-cdclW-restart-state T
  using inv-T unfolding cdclW-restart-all-struct-inv-def by auto
then have [simp]: distinct-cdclW-restart-state ?T
  unfolding distinct-cdclW-restart-state-def by auto

have cdclW-restart-conflicting T
  using inv-T unfolding cdclW-restart-all-struct-inv-def by auto
have trail ?T  $\models_{as} C \text{Not } C$ 
  by (simp add: cut-trail-wrt-clause-CNot-trail)
then have [simp]: cdclW-restart-conflicting ?T
  unfolding cdclW-restart-conflicting-def apply simp
  by (metis M  $\langle \text{cdcl}_W\text{-restart-conflicting } T \rangle$  append-assoc cdclW-restart-conflicting-decomp(2))

have
  decomp-T: all-decomposition-implies-m (init-clss T) (get-all-ann-decomposition (trail T))
  using inv-T unfolding cdclW-restart-all-struct-inv-def by auto
have all-decomposition-implies-m (init-clss ?T)
  (get-all-ann-decomposition (trail ?T))
  unfolding all-decomposition-implies-def
  proof clarify
    fix a b
    assume (a, b)  $\in \text{set } (get\text{-all-ann-decomposition } (trail ?T))$ 
    from in-get-all-ann-decomposition-in-get-all-ann-decomposition-prepend[OF this, of M]
    obtain b' where
      (a, b' @ b)  $\in \text{set } (get\text{-all-ann-decomposition } (trail T))$ 
      using M by auto
    then have unmark-l a  $\cup \text{set-mset } (init-clss T) \models_{ps} \text{unmark-l } (b' @ b)$ 
      using decomp-T unfolding all-decomposition-implies-def by fastforce
    then have unmark-l a  $\cup \text{set-mset } (init-clss ?T) \models_{ps} \text{unmark-l } (b @ b')$ 
      by (simp add: Un-commute)
    then show unmark-l a  $\cup \text{set-mset } (init-clss ?T) \models_{ps} \text{unmark-l } b$ 
      by (auto simp: image-Un)
  qed

have [simp]: cdclW-restart-learned-clause ?T
  using inv-T unfolding cdclW-restart-all-struct-inv-def cdclW-restart-learned-clause-def
  by (auto dest!: H-proped simp: clauses-def)
show ?thesis
  using  $\langle \text{all-decomposition-implies-m } (init-clss ?T) \rangle$ 
  (get-all-ann-decomposition (trail ?T))
  unfolding cdclW-restart-all-struct-inv-def by (auto simp: add-new-clause-and-update-def)
qed

```

lemma *cdcl_W-restart-all-struct-inv-add-new-clause-and-update-cdcl_W-stgy-inv*:

assumes

inv-s: *cdcl_W-stgy-invariant* *T* **and**
inv: *cdcl_W-restart-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 *cdcl_W-stgy-invariant* (*add-new-clause-and-update* *C T*)
(is *cdcl_W-stgy-invariant* ?*T'*)

proof –

have *cdcl_W-restart-all-struct-inv* ?*T'*

using *cdcl_W-restart-all-struct-inv-add-new-clause-and-update-cdcl_W-restart-all-struct-inv* *assms* **by**

blast

then have

no-dup-cut-T[simp]: *no-dup* (*trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)) **and**
n-d[simp]: *no-dup* (*trail* *T*)
using *cdcl_W-restart-M-level-inv-decomp(2)* *cdcl_W-restart-all-struct-inv-def* *inv*
n-dup-no-dup-trail-cut-trail-wrt-clause **by** *blast+*

then have *trail* (*add-new-clause-and-update* *C T*) \models_{as} *CNot C*

by (*simp add: add-new-clause-and-update-def cut-trail-wrt-clause-CNot-trail*
cdcl_W-restart-M-level-inv-def cdcl_W-restart-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*) $\forall L \in \#C. - L \notin \text{ lits-of-l } (\text{trail } T)$ **and**
trail (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*) = []
| (*not-false*)
– *lit-of* (*hd* (*trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*))) $\in \# C$ **and**
 $1 \leq \text{length } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))$

using *cut-trail-wrt-clause-hd-trail-in-or-empty-trail[of C T]* **by** *auto*

then show ?*thesis*

proof *cases*

case *false* **note** *C* = *this(1)* **and** *empty-tr* = *this(2)*

then have *[simp]*: *C* = {#}

by (*simp add: in-CNot-implies-uminus(2) multiset-eqI*)

show ?*thesis*

using *empty-tr unfolding cdcl_W-stgy-invariant-def no-smaller-conflict-def*
cdcl_W-restart-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 *L*: *get-level* (*trail* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)) (–?*L*)
= *count-decided* (*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: if-split-asm*
simp: rev-swap[symmetric] add-new-clause-and-update-def)

have *L'*: *count-decided*(*trail* (*cut-trail-wrt-clause* *C*
(*trail* *T*) *T*))

= *backtrack-lvl* (*cut-trail-wrt-clause* *C* (*trail* *T*) *T*)

using *cdcl_W-restart-all-struct-inv ?T'* **unfolding** *cdcl_W-restart-all-struct-inv-def cdcl_W-restart-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 M' D)
  then consider
    (DC) D = C
  | (D-T) D ∈# clauses T
  by (auto simp: clauses-def split: if-split-asm)
then show False
proof cases
  case D-T
  have no-smaller-confl T
    using inv-s unfolding cdclW-stgy-invariant-def by auto
  have (MT @ M') @ Decided K # M = trail T
    using MT 1(1) by auto
  then show False
    using D-T ⟨no-smaller-confl T⟩ 1(3) unfolding no-smaller-confl-def by blast
next
  case DC note -[simp] = this
  then have atm-of (−?L) ∈ atm-of ‘ (lits-of-l M)
    using 1(3) C in-CNot-implies-uminus(2) by blast
  moreover
    have lit-of (hd (M' @ Decided K # [])) = −?L
      using l 1(1)[symmetric] inv
    by (cases M', cases trail (add-init-cls C
      (cut-trail-wrt-clause C (trail T) T)))
      (auto dest!: arg-cong[of - # - hd] simp: hd-append cdclW-restart-all-struct-inv-def
        cdclW-restart-M-level-inv-def)
    from arg-cong[OF this, of atm-of]
    have atm-of (−?L) ∈ atm-of ‘ (lits-of-l (M' @ Decided K # []))
      by (cases (M' @ Decided K # [])) auto
    moreover have no-dup (trail (cut-trail-wrt-clause C (trail T) T))
      using ⟨cdclW-restart-all-struct-inv ?T'⟩ unfolding cdclW-restart-all-struct-inv-def
        cdclW-restart-M-level-inv-def by (auto simp: add-new-clause-and-update-def)
    ultimately show False
      unfolding 1(1)[symmetric, simplified] by (auto simp: lits-of-def)
  qed
qed
show ?thesis using L L' C
  unfolding cdclW-stgy-invariant-def cdclW-restart-all-struct-inv-def
  by (auto simp: add-new-clause-and-update-def intro: rev-bexI)
qed
qed

```

lemma full-cdcl_W-stgy-inv-normal-form:

assumes

full: full cdcl_W-stgy S T **and**

inv-s: cdcl_W-stgy-invariant S **and**

inv: cdcl_W-restart-all-struct-inv S

shows conflicting T = Some {#} ∧ unsatisfiable (set-mset (init-cls S))

∨ conflicting T = None ∧ trail T ⊨_{asm} init-cls S ∧ satisfiable (set-mset (init-cls S))

proof –

have no-step cdcl_W-stgy T

using full **unfolding** full-def **by** blast

moreover have cdcl_W-restart-all-struct-inv T **and** inv-s: cdcl_W-stgy-invariant T

```

apply (metis rtrancpl-cdclW-stgy-rtrancpl-cdclW-restart full full-def inv
  rtrancpl-cdclW-restart-all-struct-inv-inv)
by (metis full full-def inv inv-s rtrancpl-cdclW-stgy-cdclW-stgy-invariant)
ultimately have conflicting  $T = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } T))$ 
 $\vee \text{ conflicting } T = \text{None} \wedge \text{trail } T \models_{\text{asm}} \text{init-clss } T$ 
using cdclW-stgy-final-state-conclusive[of  $T$ ] full
unfolding cdclW-restart-all-struct-inv-def cdclW-stgy-invariant-def full-def by fast
moreover have consistent-interp (lits-of-l (trail  $T$ ))
using (cdclW-restart-all-struct-inv  $T$ ) unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def
by auto
moreover have init-clss  $S = \text{init-clss } T$ 
using inv unfolding cdclW-restart-all-struct-inv-def
by (metis rtrancpl-cdclW-stgy-no-more-init-clss full full-def)
ultimately show ?thesis
by (metis satisfiable-carac' true-annot-def true-annots-def true-clss-def)
qed

```

lemma incremental-cdcl_W-restart-inv:

assumes

inc: incremental-cdcl_W-restart S T **and**

inv: cdcl_W-restart-all-struct-inv S **and**

s-inv: cdcl_W-stgy-invariant S

shows

cdcl_W-restart-all-struct-inv T **and**

cdcl_W-stgy-invariant T

using *inc*

proof (induction)

case (add-confl C T)

let ? $T = (\text{update-conflicting } (\text{Some } C) (\text{add-init-clss } C$
 $(\text{cut-trail-wrt-clause } C (\text{trail } S) S)))$

have cdcl_W-restart-all-struct-inv ? T **and** inv-s- T : cdcl_W-stgy-invariant ? T

using add-confl.hyps(1,2,4) add-new-clause-and-update-def

cdcl_W-restart-all-struct-inv-add-new-clause-and-update-cdcl_W-restart-all-struct-inv inv **apply** auto[1]

using add-confl.hyps(1,2,4) add-new-clause-and-update-def

cdcl_W-restart-all-struct-inv-add-new-clause-and-update-cdcl_W-stgy-inv inv s-inv **by** auto

case 1 **show** ?case

by (metis add-confl.hyps(1,2,4,5) add-new-clause-and-update-def

cdcl_W-restart-all-struct-inv-add-new-clause-and-update-cdcl_W-restart-all-struct-inv

rtrancpl-cdcl_W-restart-all-struct-inv-inv rtrancpl-cdcl_W-stgy-rtrancpl-cdcl_W-restart full-def inv)

case 2 **show** ?case

by (metis inv-s- T add-confl.hyps(1,2,4,5) add-new-clause-and-update-def

cdcl_W-restart-all-struct-inv-add-new-clause-and-update-cdcl_W-restart-all-struct-inv full-def inv

rtrancpl-cdcl_W-stgy-cdcl_W-stgy-invariant)

next

case (add-no-confl C T)

case 1

have cdcl_W-restart-all-struct-inv (add-init-clss C S)

using inv (distinct-mset C) **unfolding** cdcl_W-restart-all-struct-inv-def no-strange-atm-def

cdcl_W-restart-M-level-inv-def distinct-cdcl_W-restart-state-def cdcl_W-restart-conflicting-def cdcl_W-restart-learned-clause

by (auto 9 1 simp: all-decomposition-implies-insert-single clauses-def)

then show ?case

using add-no-confl(5) **unfolding** full-def **by** (auto intro: rtrancpl-cdcl_W-stgy-cdcl_W-restart-all-struct-inv)

case 2

have nc: $\forall M. (\exists K \ i \ M'. \text{trail } S = M' @ \text{Decided } K \ \# \ M) \longrightarrow \neg M \models_{\text{as}} C \text{Not } C$

```

using  $\langle \neg \text{ trail } S \models_{as} C \text{Not } C \rangle$ 
by (auto simp: true-annots-true-cls-def-iff-negation-in-model)

have cdclW-stgy-invariant (add-init-cls C S)
  using s-inv  $\langle \neg \text{ trail } S \models_{as} C \text{Not } C \rangle$  inv unfolding cdclW-stgy-invariant-def
no-smaller-confl-def eq-commute[of - trail -] cdclW-restart-M-level-inv-def cdclW-restart-all-struct-inv-def
  by (auto simp: clauses-def nc)
then show ?case
  by (metis  $\langle \text{cdcl}_W\text{-restart-all-struct-inv } (\text{add-init-cls } C \ S) \rangle$  add-no-confl.hyps(5) full-def
    rtranclp-cdclW-stgy-cdclW-stgy-invariant)
qed

lemma rtranclp-incremental-cdclW-restart-inv:
assumes
  inc: incremental-cdclW-restart** S T and
  inv: cdclW-restart-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows
  cdclW-restart-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-restart-inv by blast+

lemma incremental-conclusive-state:
assumes
  inc: incremental-cdclW-restart S T and
  inv: cdclW-restart-all-struct-inv S and
  s-inv: cdclW-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
proof induction
print-cases
case (add-confl C T) note tr = this(1) and dist = this(2) and conf = this(3) and C = this(4) and
full = this(5)

  have full cdclW-stgy T T
    using full unfolding full-def by auto
  then show ?case
    using full C conf dist tr
    by (metis full-cdclW-stgy-inv-normal-form incremental-cdclW-restart.simps incremental-cdclW-restart-inv(1)
      incremental-cdclW-restart-inv(2) inv s-inv)
next
case (add-no-confl C T) note tr = this(1) and dist = this(2) and conf = this(3) and C = this(4)
  and full = this(5)

  have full cdclW-stgy T T
    using full unfolding full-def by auto
  then show ?case
    by (meson C conf dist full full-cdclW-stgy-inv-normal-form incremental-cdclW-restart.add-no-confl
      incremental-cdclW-restart-inv(1) incremental-cdclW-restart-inv(2) inv s-inv tr)
qed

lemma tranclp-incremental-correct:

```

```

assumes
  inc: incremental-cdclW-restart++ S T and
  inv: cdclW-restart-all-struct-inv S and
  s-inv: cdclW-stgy-invariant S
shows conflicting T = Some {#} ∧ unsatisfiable (set-mset (init-clss T))
   $\vee$  conflicting T = None ∧ trail T ⊨asm init-clss T ∧ satisfiable (set-mset (init-clss T))
using inc apply induction
using assms incremental-conclusive-state apply blast
by (meson incremental-conclusive-state inv rtranclp-incremental-cdclW-restart-inv s-inv
  tranclp-into-rtranclp)

```

end

end

theory *DPLL-CDCL-W-Implementation*

imports *Partial-Annotated-Clausal-Logic CDCL-W-Level*

begin

Chapter 7

Implementation of DPLL and CDCL

We then reuse all the theorems to go towards an implementation using 2-watched literals:

- `CDCL_W_Abstract_State.thy` defines a better-suited state: the operation operating on it are more constrained, allowing simpler proofs and less edge cases later.

7.1 Simple List-Based Implementation of the DPLL and CDCL

The idea of the list-based implementation is to test the stack: the theories about the calculi, adapting the theorems to a simple implementation and the code exportation. The implementation are very simple and simply iterate over-and-over on lists.

7.1.1 Common Rules

Propagation

The following theorem holds:

lemma *lits-of-l-unfold*[*iff*]:

$(\forall c \in \text{set } C. -c \in \text{lits-of-l } Ms) \longleftrightarrow Ms \models_{\text{as}} C\text{Not } (mset\ C)$

unfolding *true-annots-def Ball-def true-annot-def CNot-def* **by** *auto*

The right-hand version is written at a high-level, but only the left-hand side is executable.

definition *is-unit-clause* :: *'a literal list* \Rightarrow (*'a, 'b*) *ann-lits* \Rightarrow *'a literal option*

where

is-unit-clause *l M* =

(*case List.filter* ($\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of-l } M$) *l of*
 a # [] \Rightarrow *if* *M* $\models_{\text{as}} C\text{Not } (mset\ l - \{\#a\# \})$ *then Some a else None*
 | - \Rightarrow *None*)

definition *is-unit-clause-code* :: *'a literal list* \Rightarrow (*'a, 'b*) *ann-lits*

\Rightarrow *'a literal option* **where**

is-unit-clause-code *l M* =

(*case List.filter* ($\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of-l } M$) *l of*
 a # [] \Rightarrow *if* ($\forall c \in \text{set } (\text{remove1 } a\ l). -c \in \text{lits-of-l } M$) *then Some a else None*
 | - \Rightarrow *None*)

lemma *is-unit-clause-is-unit-clause-code*[*code*]:

is-unit-clause *l M* = *is-unit-clause-code* *l M*

```

proof –
  have 1:  $\bigwedge a. (\forall c \in \text{set } (\text{remove1 } a \ l). - c \in \text{lits-of-}l \ M) \longleftrightarrow M \models_{as} CNot \ (mset \ l - \{\#a\# \})$ 
    using lits-of-l-unfold[of remove1 - l, of - M] by simp
  then show ?thesis
    unfolding is-unit-clause-code-def is-unit-clause-def 1 by blast
qed

lemma is-unit-clause-some-undef:
  assumes is-unit-clause l M = Some a
  shows undefined-lit M a
proof –
  have (case [a ← l . atm-of a ∉ atm-of ‘lits-of-l M] of [] ⇒ None
    | [a] ⇒ if M  $\models_{as}$  CNot (mset l - {#a#}) then Some a else None
    | a # ab # xa ⇒ Map.empty xa) = Some a
    using assms unfolding is-unit-clause-def .
  then have a ∈ set [a ← l . atm-of a ∉ atm-of ‘lits-of-l M]
    apply (cases [a ← l . atm-of a ∉ atm-of ‘lits-of-l M])
      apply simp
      apply (rename-tac aa list; case-tac list) by (auto split: if-split-asm)
  then have atm-of a ∉ atm-of ‘lits-of-l M by auto
  then show ?thesis
    by (simp add: Decided-Propagated-in-iff-in-lits-of-l
      atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set )
qed

lemma is-unit-clause-some-CNot: is-unit-clause l M = Some a ⇒ M  $\models_{as}$  CNot (mset l - {#a#})
  unfolding is-unit-clause-def
proof –
  assume (case [a ← l . atm-of a ∉ atm-of ‘lits-of-l M] of [] ⇒ None
    | [a] ⇒ if M  $\models_{as}$  CNot (mset l - {#a#}) then Some a else None
    | a # ab # xa ⇒ Map.empty xa) = Some a
  then show ?thesis
    apply (cases [a ← l . atm-of a ∉ atm-of ‘lits-of-l M], simp)
      apply simp
      apply (rename-tac aa list; case-tac list) by (auto split: if-split-asm)
qed

lemma is-unit-clause-some-in: is-unit-clause l M = Some a ⇒ a ∈ set l
  unfolding is-unit-clause-def
proof –
  assume (case [a ← l . atm-of a ∉ atm-of ‘lits-of-l M] of [] ⇒ None
    | [a] ⇒ if M  $\models_{as}$  CNot (mset l - {#a#}) then Some a else None
    | a # ab # xa ⇒ Map.empty xa) = Some a
  then show a ∈ set l
    by (cases [a ← l . atm-of a ∉ atm-of ‘lits-of-l M])
      (fastforce dest: filter-eq-ConsD split: if-split-asm split: list.splits) +
qed

lemma is-unit-clause-Nil[simp]: is-unit-clause [] M = None
  unfolding is-unit-clause-def by auto

```

Unit propagation for all clauses

Finding the first clause to propagate

fun *find-first-unit-clause* :: '*a* literal list list ⇒ ('*a*, '*b*) ann-lits

$\Rightarrow ('a \text{ literal} \times 'a \text{ literal list}) \text{ option}$ **where**
find-first-unit-clause ($a \# l$) $M =$
 (case *is-unit-clause* $a \ M$ of
 None \Rightarrow *find-first-unit-clause* $l \ M$
 | Some $L \Rightarrow$ Some (L, a) |
find-first-unit-clause [] - = None

lemma *find-first-unit-clause-some*:
find-first-unit-clause $l \ M =$ Some (a, c)
 $\implies c \in \text{set } l \wedge M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\# \}) \wedge \text{undefined-lit } M \ a \wedge a \in \text{set } c$
apply (*induction* l)
apply *simp*
by (*auto split: option.splits dest: is-unit-clause-some-in is-unit-clause-some-CNot is-unit-clause-some-undef*)

lemma *propagate-is-unit-clause-not-None*:
assumes *dist: distinct* c **and**
 $M: M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\# \})$ **and**
undef: undefined-lit $M \ a$ **and**
ac: a $\in \text{set } c$
shows *is-unit-clause* $c \ M \neq$ None

proof -
have $[a \leftarrow c . \text{atm-of } a \notin \text{atm-of ' lits-of-} l \ M] = [a]$
using *assms*
proof (*induction* c)
 case Nil **then show** ?case **by** *simp*
next
 case (Cons $ac \ c$)
 show ?case
 proof (*cases* $a = ac$)
 case True
 then show ?thesis **using** Cons
 by (*auto simp del: lits-of-l-unfold*
 simp add: lits-of-l-unfold[symmetric] Decided-Propagated-in-iff-in-lits-of-l
 atm-of-eq-atm-of atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)
 next
 case False
 then have $T: \text{mset } c + \{\#ac\# \} - \{\#a\# \} = \text{mset } c - \{\#a\# \} + \{\#ac\# \}$
 by (*auto simp add: multiset-eq-iff*)
 show ?thesis **using** False Cons
 by (*auto simp add: T atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*)
 qed
qed
then show ?thesis
using M **unfolding** *is-unit-clause-def* **by** *auto*
qed

lemma *find-first-unit-clause-none*:
 $\text{distinct } c \implies c \in \text{set } l \implies M \models_{\text{as}} \text{CNot } (\text{mset } c - \{\#a\# \}) \implies \text{undefined-lit } M \ a \implies a \in \text{set } c$
 $\implies \text{find-first-unit-clause } l \ M \neq$ None
by (*induction* l)
 (*auto split: option.split simp add: propagate-is-unit-clause-not-None*)

Decide

fun *find-first-unused-var* :: $'a \text{ literal list list} \Rightarrow 'a \text{ literal set} \Rightarrow 'a \text{ literal option}$ **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 ($\lambda lit. lit \notin M \wedge \neg lit \notin M$) $a = \text{None} \longleftrightarrow \text{atm-of ' set } a \subseteq \text{atm-of ' } M$
apply (*induct* a)
using *atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*
by (*force simp add: atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set*) +

lemma *find-some*: *List.find* ($\lambda lit. lit \notin M \wedge \neg lit \notin M$) $a = \text{Some } b \implies b \in \text{set } a \wedge b \notin M \wedge \neg b \notin M$
unfolding *find-Some-iff* **by** (*metis nth-mem*)

lemma *find-first-unused-var-None*[*iff*]:
find-first-unused-var l $M = \text{None} \longleftrightarrow (\forall a \in \text{set } l. \text{atm-of ' set } a \subseteq \text{atm-of ' } M)$
by (*induct* l)
 (auto *split: option.splits dest!: find-some*
simp add: image-subset-iff atm-of-in-atm-of-set-iff-in-set-or-uminus-in-set)

lemma *find-first-unused-var-Some-not-all-incl*:
assumes *find-first-unused-var* l $M = \text{Some } c$
shows $\neg(\forall a \in \text{set } l. \text{atm-of ' set } a \subseteq \text{atm-of ' } M)$
proof –
have *find-first-unused-var* l $M \neq \text{None}$
using *assms* **by** (*cases find-first-unused-var* l M) *auto*
then show $\neg(\forall a \in \text{set } l. \text{atm-of ' set } a \subseteq \text{atm-of ' } M)$ **by** *auto*
qed

lemma *find-first-unused-var-Some*:
find-first-unused-var l $M = \text{Some } a \implies (\exists m \in \text{set } l. a \in \text{set } m \wedge a \notin M \wedge \neg a \notin M)$
by (*induct* l) (*auto split: option.splits dest: find-some*)

lemma *find-first-unused-var-undefined*:
find-first-unused-var l (*lits-of-l* Ms) = Some $a \implies \text{undefined-lit } Ms$ a
using *find-first-unused-var-Some*[*of* l *lits-of-l* Ms a] *Decided-Propagated-in-iff-in-lits-of-l*
by *blast*

7.1.2 CDCL specific functions

Level

fun *maximum-level-code*:: 'a *literal list* \Rightarrow ('a, 'b) *ann-lits* \Rightarrow nat
where
maximum-level-code [] = 0 |
maximum-level-code ($L \# Ls$) $M = \max (\text{get-level } M$ $L) (\text{maximum-level-code } Ls$ $M)$

lemma *maximum-level-code-eq-get-maximum-level*[*simp*]:
maximum-level-code D $M = \text{get-maximum-level } M$ (*mset* D)
by (*induction* D) (*auto simp add: get-maximum-level-plus*)

lemma [*code*]:
fixes $M :: ('a, 'b) \text{ann-lits}$
shows *get-maximum-level* M (*mset* D) = *maximum-level-code* D M
by *simp*

Backjumping

fun *find-level-decomp* **where**

find-level-decomp $M \ [] \ D \ k = \text{None} \mid$

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* $M \ Ls \ D \ k = \text{Some} \ (L, j)$

shows $L \in \text{set } Ls \wedge \text{get-maximum-level } M \ (\text{mset } (\text{remove1 } L \ (Ls \ @ \ D))) = j \wedge \text{get-level } M \ L = k$

using *assms*

proof (*induction* Ls *arbitrary*: D)

case *Nil*

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

next

case (*Cons* $L' \ Ls$) **note** $IH = \text{this}(1)$ **and** $H = \text{this}(2)$

def *find* \equiv (*if* *get-level* $M \ L' \neq k \vee \neg \text{get-maximum-level } M \ (\text{mset } D + \text{mset } Ls) < \text{get-level } M \ L'$

then *find-level-decomp* $M \ Ls \ (L' \ \# \ D) \ k$

else *Some* ($L', \text{get-maximum-level } M \ (\text{mset } D + \text{mset } Ls)$))

have $a1: \bigwedge D. \text{find-level-decomp } M \ Ls \ D \ k = \text{Some} \ (L, j) \Rightarrow$

$L \in \text{set } Ls \wedge \text{get-maximum-level } M \ (\text{mset } Ls + \text{mset } D - \{\#L\ \#\}) = j \wedge \text{get-level } M \ L = k$

using IH **by** *simp*

have $a2: \text{find} = \text{Some} \ (L, j)$

using H **unfolding** *find-def* **by** (*auto split: if-split-asm*)

{ **assume** *Some* ($L', \text{get-maximum-level } M \ (\text{mset } D + \text{mset } Ls)$) $\neq \text{find}$

then have $f3: L \in \text{set } Ls$ **and** *get-maximum-level* $M \ (\text{mset } Ls + \text{mset } (L' \ \# \ D) - \{\#L\ \#\}) = j$

using $a1 \ IH \ a2$ **unfolding** *find-def* **by** *meson+*

moreover then have $\text{mset } Ls + \text{mset } D - \{\#L\ \#\} + \{\#L'\ \#\} = \{\#L'\ \#\} + \text{mset } D + (\text{mset } Ls - \{\#L\ \#\})$

by (*auto simp: ac-simps multiset-eq-iff Suc-leI*)

ultimately have $f4: \text{get-maximum-level } M \ (\text{mset } Ls + \text{mset } D - \{\#L\ \#\} + \{\#L'\ \#\}) = j$

by (*metis add.commute diff-union-single-conv in-multiset-in-set mset.simps(2)*)

} **note** $f4 = \text{this}$

have $\{\#L'\ \#\} + (\text{mset } Ls + \text{mset } D) = \text{mset } Ls + (\text{mset } D + \{\#L'\ \#\})$

by (*auto simp: ac-simps*)

then have

$L = L' \longrightarrow \text{get-maximum-level } M \ (\text{mset } Ls + \text{mset } D) = j \wedge \text{get-level } M \ L' = k$ **and**

$L \neq L' \longrightarrow L \in \text{set } Ls \wedge \text{get-maximum-level } M \ (\text{mset } Ls + \text{mset } D - \{\#L\ \#\} + \{\#L'\ \#\}) = j \wedge \text{get-level } M \ L = k$

using $a2 \ a1[\text{of } L' \ \# \ D]$ **unfolding** *find-def* **apply** (*metis add-diff-cancel-left' mset.simps(2) option.inject prod.inject union-commute*)

using $f4 \ a2 \ a1[\text{of } L' \ \# \ D]$ **unfolding** *find-def* **by** (*metis option.inject prod.inject*)

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

qed

lemma *find-level-decomp-none*:

assumes *find-level-decomp* $M \ Ls \ E \ k = \text{None}$ **and** $\text{mset } (L \ \# \ D) = \text{mset } (Ls \ @ \ E)$

shows $\neg(L \in \text{set } Ls \wedge \text{get-maximum-level } M \ (\text{mset } D) < k \wedge k = \text{get-level } M \ L)$

using *assms*

proof (*induction* Ls *arbitrary*: $E \ L \ D$)

case *Nil*

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

next

```

case (Cons L' Ls) note IH = this(1) and find-none = this(2) and LD = this(3)
have mset D + {#L'#} = mset E + (mset Ls + {#L'#})  $\implies$  mset D = mset E + mset Ls
  by (metis add-right-imp-eq union-assoc)
then show ?case
  using find-none IH[of L' # E L D] LD by (auto simp add: ac-simps split: if-split-asm)
qed

```

fun bt-cut **where**

```

bt-cut i (Propagated - - # Ls) = bt-cut i Ls |
bt-cut i (Decided K # Ls) = (if count-decided Ls = i then Some (Decided K # Ls) else bt-cut i Ls) |
bt-cut i [] = None

```

lemma bt-cut-some-decomp:

```

assumes no-dup M and bt-cut i M = Some M'
shows  $\exists K M2 M1. M = M2 @ M' \wedge M' = \text{Decided } K \# M1 \wedge \text{get-level } M K = (i+1)$ 
using assms by (induction i M rule: bt-cut.induct) (auto split: if-split-asm)

```

lemma bt-cut-not-none:

```

assumes no-dup M and M = M2 @ Decided K # M' and get-level M K = (i+1)
shows bt-cut i M  $\neq$  None
using assms by (induction M2 arbitrary: M rule: ann-lit-list-induct)
(auto simp: atm-lit-of-set-lits-of-l)

```

lemma get-all-ann-decomposition-ex:

```

 $\exists N. (\text{Decided } K \# M', N) \in \text{set } (\text{get-all-ann-decomposition } (M2 @ \text{Decided } K \# M'))$ 
apply (induction M2 rule: ann-lit-list-induct)
apply auto[2]
by (rename-tac L m xs, case-tac get-all-ann-decomposition (xs @ Decided K # M'))
auto

```

lemma bt-cut-in-get-all-ann-decomposition:

```

assumes no-dup M and bt-cut i M = Some M'
shows  $\exists M2. (M', M2) \in \text{set } (\text{get-all-ann-decomposition } M)$ 
using bt-cut-some-decomp[OF assms] by (auto simp add: get-all-ann-decomposition-ex)

```

fun do-backtrack-step **where**

```

do-backtrack-step (M, N, U, k, Some D) =
  (case find-level-decomp M D [] k of
    None  $\Rightarrow$  (M, N, U, k, Some D)
  | Some (L, j)  $\Rightarrow$ 
    (case bt-cut j M of
      Some (Decided - # Ls)  $\Rightarrow$  (Propagated L D # Ls, N, D # U, j, None)
    | -  $\Rightarrow$  (M, N, U, k, Some D))
  ) |
do-backtrack-step S = S

```

end

theory DPLL-W-Implementation

imports DPLL-CDCL-W-Implementation DPLL-W $\sim\sim$ /src/HOL/Library/Code-Target-Numeral

begin

7.1.3 Simple Implementation of DPLL

Combining the propagate and decide: a DPLL step

definition DPLL-step :: int dpll_W-ann-lits \times int literal list list

```

⇒ int dpllW-ann-lits × int literal list list where
DPLL-step = (λ(Ms, N).
  (case find-first-unit-clause N Ms of
    Some (L, -) ⇒ (Propagated L () # Ms, N)
  | - ⇒
    if ∃ C ∈ set N. (∀ c ∈ set C. -c ∈ lits-of-l Ms)
    then
      (case backtrack-split Ms of
        (-, L # M) ⇒ (Propagated (- (lit-of L)) () # M, N)
      | (-, -) ⇒ (Ms, N)
      )
    else
      (case find-first-unused-var N (lits-of-l Ms) of
        Some a ⇒ (Decided a # Ms, N)
      | None ⇒ (Ms, N))))

```

Example of propagation:

```

value DPLL-step ([Decided (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 ≡ λ(Ms::(int, unit) ann-lits)
  (N:: int literal list list). (Ms, mset (map mset N))
abbreviation toS' ≡ λ(Ms::(int, unit) ann-lits,
  N:: int literal list list). (Ms, mset (map mset N))

```

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-dpll_W-step*:

```

assumes step: (Ms', N') = DPLL-step (Ms, N)
and neq: (Ms, N) ≠ (Ms', N')
shows dpllW (toS Ms N) (toS Ms' N')

```

proof –

```

let ?S = (Ms, mset (map mset N))
{ fix L E
  assume unit: find-first-unit-clause N Ms = Some (L, E)
  then have Ms'N: (Ms', N') = (Propagated L () # Ms, N)
    using step unfolding DPLL-step-def by auto
  obtain C where
    C: C ∈ set N and
    Ms: Ms ⊨as CNot (mset C - {#L#}) and
    undef: undefined-lit Ms L and
    L ∈ set C using find-first-unit-clause-some[OF unit] by metis
  have dpllW (Ms, mset (map mset N))
    (Propagated L () # fst (Ms, mset (map mset N)), snd (Ms, mset (map mset N)))
    apply (rule dpllW.propagate)
    using Ms undef C ⟨L ∈ set C⟩ by (auto simp add: C)
  then have ?thesis using Ms'N by auto
}
moreover
{ assume unit: find-first-unit-clause N Ms = None
  assume exC: ∃ C ∈ set N. Ms ⊨as CNot (mset C)
  then obtain C where C: C ∈ set N and Ms: Ms ⊨as CNot (mset C) by auto
  then obtain L M M' where bt: backtrack-split Ms = (M', L # M)
    using step exC neq unfolding DPLL-step-def prod.case unit
    by (cases backtrack-split Ms, rename-tac b, case-tac b) auto

```

```

then have is-decided L using backtrack-split-snd-hd-decided[of Ms] by auto
have 1: dpllW (Ms, mset (map mset N))
  (Propagated (¬ lit-of L) () # M, snd (Ms, mset (map mset N)))
  apply (rule dpllW.backtrack[OF - ⟨is-decided L⟩, of ])
  using C Ms bt by auto
moreover have (Ms', N') = (Propagated (¬ (lit-of L)) () # M, N)
  using step exC unfolding DPLL-step-def bt prod.case unit by auto
ultimately have ?thesis by auto
}
moreover
{ assume unit: find-first-unit-clause N Ms = None
  assume exC: ¬ (∃ C ∈ set N. Ms ⊨as CNot (mset C))
  obtain L where unused: find-first-unused-var N (lits-of-l Ms) = Some L
    using step exC neq unfolding DPLL-step-def prod.case unit
    by (cases find-first-unused-var N (lits-of-l Ms)) auto
  have dpllW (Ms, mset (map mset N))
    (Decided L # fst (Ms, mset (map mset N)), snd (Ms, mset (map mset N)))
    apply (rule dpllW.decided[of ?S L])
    using find-first-unused-var-Some[OF unused]
    by (auto simp add: Decided-Propagated-in-iff-in-lits-of-l atms-of-ms-def)
  moreover have (Ms', N') = (Decided L # Ms, N)
    using step exC unfolding DPLL-step-def unused prod.case unit by auto
  ultimately have ?thesis by auto
}
ultimately show ?thesis by (cases find-first-unit-clause N Ms) auto
qed

```

lemma DPLL-step-stuck-final-state:

```

assumes step: (Ms, N) = DPLL-step (Ms, N)
shows conclusive-dpllW-state (toS Ms N)

```

proof –

```

have unit: find-first-unit-clause N Ms = None
  using step unfolding DPLL-step-def by (auto split:option.splits)

```

```

{ assume n: ∃ C ∈ set N. Ms ⊨as CNot (mset C)
  then have Ms: (Ms, N) = (case backtrack-split Ms of (x, []) ⇒ (Ms, N)
    | (x, L # M) ⇒ (Propagated (¬ lit-of L) () # M, N))
    using step unfolding DPLL-step-def by (simp add:unit)

```

```

have snd (backtrack-split Ms) = []

```

```

proof (cases backtrack-split Ms, cases snd (backtrack-split Ms))

```

```

  fix a b

```

```

  assume backtrack-split Ms = (a, b) and snd (backtrack-split Ms) = []

```

```

  then show 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

```

```

  then have Ms: Ms = Propagated (¬ lit-of aa) () # list using Ms by auto

```

```

  have is-decided aa using backtrack-split-snd-hd-decided[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

```

```

  then show snd (backtrack-split Ms) = [] by blast

```

```

qed

```



```

then have ?thesis
  using n backtrack-snd-empty-not-decided[of Ms] unfolding conclusive-dpllW-state-def
  by (cases backtrack-split Ms) auto
}
moreover {
  assume n:  $\neg (\exists C \in \text{set } N. Ms \models_{as} CNot (mset C))$ 
  then have find-first-unused-var N (lits-of-l Ms) = None
    using step unfolding DPLL-step-def by (simp add: unit split: option.splits)
  then have a:  $\forall a \in \text{set } N. atm\text{-of } 'set\ a \subseteq atm\text{-of } '(lits\text{-of-l } Ms)$  by auto
  have fst (toS Ms N)  $\models_{asm}$  snd (toS Ms N) unfolding true-annots-def CNot-def Ball-def
  proof clarify
    fix x
    assume x:  $x \in \text{set-mset } (clauses (toS Ms N))$ 
    then have  $\neg Ms \models_{as} CNot\ x$  using n unfolding true-annots-def CNot-def Ball-def by auto
    moreover have total-over-m (lits-of-l Ms) {x}
      using a x image-iff in-mono atms-of-s-def
      unfolding total-over-m-def total-over-set-def lits-of-def by fastforce
    ultimately show fst (toS Ms N)  $\models_a$  x
      using total-not-CNot[of lits-of-l Ms x] by (simp add: true-annot-def true-annots-true-cls)
    qed
  then have ?thesis unfolding conclusive-dpllW-state-def by blast
}
ultimately show ?thesis by blast
qed

```

Adding invariants

Invariant tested in the function `function DPLL-ci :: int dpllW-ann-lits \Rightarrow int literal list list \Rightarrow int dpllW-ann-lits \times int literal list list where`

```

DPLL-ci Ms N =
  (if  $\neg dpll_W\text{-all-inv } (Ms, mset (map mset N))$ 
   then (Ms, N)
   else
    let (Ms', N') = DPLL-step (Ms, N) in
    if (Ms', N') = (Ms, N) then (Ms, N) else DPLL-ci Ms' N)
  by fast+

```

termination

```

proof (relation {(S', S). (toS' S', toS' S)  $\in$  {(S', S). dpllW-all-inv S  $\wedge$  dpllW S S'}})
  show wf {(S', S). (toS' S', toS' S)  $\in$  {(S', S). dpllW-all-inv S  $\wedge$  dpllW S S'}}
    using wf-if-measure-f[OF dpllW-wf, of toS'] by auto

```

next

```

fix Ms :: int dpllW-ann-lits and N x xa y
assume  $\neg \neg dpll_W\text{-all-inv } (toS Ms N)$ 
and step: x = DPLL-step (Ms, N)
and x: (xa, y) = x
and (xa, y)  $\neq$  (Ms, N)
then show ((xa, N), Ms, N)  $\in$  {(S', S). (toS' S', toS' S)  $\in$  {(S', S). dpllW-all-inv S  $\wedge$  dpllW S S'}}
  using DPLL-step-is-a-dpllW-step dpllW-same-clauses split-conv by fastforce
qed

```

No invariant tested `function (domintros) DPLL-part :: int dpllW-ann-lits \Rightarrow int literal list list \Rightarrow int dpllW-ann-lits \times int literal list list where`

```

DPLL-part Ms N =
  (let (Ms', N') = DPLL-step (Ms, N) in
  if (Ms', N') = (Ms, N) then (Ms, N) else DPLL-part Ms' N)

```

by *fast+*

lemma *snd-DPLL-step[simp]*:
 $\text{snd } (DPLL\text{-step } (Ms, N)) = N$
unfolding *DPLL-step-def* **by** (*auto split: if-split option.splits prod.splits list.splits*)

lemma *dpll_W-all-inv-implieS-2-eq3-and-dom*:
assumes *dpll_W-all-inv* (*Ms, mset (map mset N)*)
shows *DPLL-ci Ms N = DPLL-part Ms N \wedge DPLL-part-dom (Ms, N)*
using *assms*

proof (*induct rule: DPLL-ci.induct*)
case (*1 Ms N*)
have $\text{snd } (DPLL\text{-step } (Ms, N)) = N$ **by** *auto*
then obtain *Ms'* **where** *Ms'*: $DPLL\text{-step } (Ms, N) = (Ms', N)$ **by** (*cases DPLL-step (Ms, N)*) *auto*
have *inv'*: *dpll_W-all-inv* (*toS Ms' N*) **by** (*metis (mono-tags) 1.prem DPLL-step-is-a-dpll_W-step Ms' dpll_W-all-inv old.prod.inject*)
{ assume (*Ms', N*) \neq (*Ms, N*)
then have *DPLL-ci Ms' N = DPLL-part Ms' N \wedge DPLL-part-dom (Ms', N)* **using** *1(1)[of - Ms' N] Ms'*
1(2) inv' by auto
then have *DPLL-part-dom (Ms, N)* **using** *DPLL-part.domintros Ms' by fastforce*
moreover have *DPLL-ci Ms N = DPLL-part Ms N* **using** *1.prem DPLL-part.psimps Ms'*
 $\langle DPLL\text{-ci } Ms' N = DPLL\text{-part } Ms' N \wedge DPLL\text{-part-dom } (Ms', N) \rangle \langle DPLL\text{-part-dom } (Ms, N) \rangle$ **by**
auto
ultimately have *?case* **by** *blast*
}
moreover {
assume (*Ms', N*) = (*Ms, N*)
then have *?case* **using** *DPLL-part.domintros DPLL-part.psimps Ms' by fastforce*
}
ultimately show *?case* **by** *blast*
qed

lemma *DPLL-ci-dpll_W-rtrancp*:
assumes *DPLL-ci Ms N = (Ms', N')*
shows *dpll_W** (toS Ms N) (toS Ms' N)*
using *assms*

proof (*induct Ms N arbitrary: Ms' N' rule: DPLL-ci.induct*)
case (*1 Ms N Ms' N'*) **note** *IH = this(1)* **and** *step = this(2)*
obtain *S₁ S₂* **where** *S*: (*S₁, S₂*) = *DPLL-step (Ms, N)* **by** (*cases DPLL-step (Ms, N)*) *auto*

{ assume $\neg dpll_W\text{-all-inv } (toS Ms N)$
then have (*Ms, N*) = (*Ms', N*) **using** *step* **by** *auto*
then have *?case* **by** *auto*
}
moreover
{ assume *dpll_W-all-inv* (*toS Ms N*)
and (*S₁, S₂*) = (*Ms, N*)
then have *?case* **using** *S step* **by** *auto*
}
moreover
{ assume *dpll_W-all-inv* (*toS Ms N*)
and (*S₁, S₂*) \neq (*Ms, N*)
moreover obtain *S₁' S₂'* **where** *DPLL-ci S₁ N = (S₁', S₂') by (cases DPLL-ci S₁ N) auto*
moreover have *DPLL-ci Ms N = DPLL-ci S₁ N* **using** *DPLL-ci.simps[of Ms N] calculation*
proof –

```

have (case (S1, S2) of (ms, lss) ⇒
  if (ms, lss) = (Ms, N) then (Ms, N) else DPLL-ci ms N) = DPLL-ci Ms N
  using S DPLL-ci.simps[of Ms N] calculation by presburger
then have (if (S1, S2) = (Ms, N) then (Ms, N) else DPLL-ci S1 N) = DPLL-ci Ms N
  by fastforce
then show ?thesis
  using calculation(2) by presburger
qed
ultimately have dpllW** (toS S1' N) (toS Ms' N) using IH[of (S1, S2) S1 S2] S step by simp

moreover have dpllW (toS Ms N) (toS S1 N)
  by (metis DPLL-step-is-a-dpllW-step S ⟨(S1, S2) ≠ (Ms, N)⟩ prod.sel(2) snd-DPLL-step)
ultimately have ?case by (metis (mono-tags, hide-lams) IH S ⟨(S1, S2) ≠ (Ms, N)⟩
  ⟨DPLL-ci Ms N = DPLL-ci S1 N⟩ ⟨dpllW-all-inv (toS Ms N)⟩ converse-rtranclp-into-rtranclp
  local.step)
}
ultimately show ?case by blast
qed

lemma dpllW-all-inv-dpllW-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
  then show 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
    then have DPLL-ci Ms' N = (Ms, N) using step inv st Ms'N'[symmetric] by fastforce
    then have 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)
    then show False using dpllW-all-inv-dpllW-tranclp-irrefl inv by auto
  qed
  then show ?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** $SN: (S, N) = DPLL\text{-}step\ (Ms, N)$ **using** $DPLL\text{-}step\text{-}obtains$ **by** $metis$
{ **assume** $\neg dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
then have $?case$ **using** $that$ **by** $auto$
}
moreover {
assume $n: (S, N) \neq (Ms, N)$
and $inv: dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
have $\exists ms. DPLL\text{-}step\ (Ms, N) = (ms, N)$
by ($metis\ \langle \bigwedge thesisa. (\bigwedge S. (S, N) = DPLL\text{-}step\ (Ms, N) \implies thesisa) \implies thesisa \rangle$)
then have $?thesis$
using $IH\ that$ **by** $fastforce$
}
moreover {
assume $n: (S, N) = (Ms, N)$
then have $?case$ **using** $SN\ that$ **by** $fastforce$
}
ultimately show $?case$ **by** $blast$
qed

lemma $DPLL\text{-}ci\text{-}no\text{-}more\text{-}step$:

assumes $step: DPLL\text{-}ci\ Ms\ N = (Ms', N')$
shows $DPLL\text{-}ci\ Ms'\ N' = (Ms', N')$
using $assms$

proof (*induct arbitrary: $Ms'\ N'$ rule: $DPLL\text{-}ci.induct$*)
case (1 $Ms\ N\ Ms'\ N'$) **note** $IH = this(1)$ **and** $step = this(2)$
obtain S_1 **where** $S: (S_1, N) = DPLL\text{-}step\ (Ms, N)$ **using** $DPLL\text{-}step\text{-}obtains$ **by** $auto$
{ **assume** $\neg dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
then have $?case$ **using** $step$ **by** $auto$
}
moreover {
assume $dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
and $(S_1, N) = (Ms, N)$
then have $?case$ **using** $S\ step$ **by** $auto$
}
moreover
{ **assume** $inv: dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
assume $n: (S_1, N) \neq (Ms, N)$
obtain S_1' **where** $SS: (S_1', N) = DPLL\text{-}ci\ S_1\ N$ **using** $DPLL\text{-}ci\text{-}obtains$ **by** $blast$
moreover have $DPLL\text{-}ci\ Ms\ N = DPLL\text{-}ci\ S_1\ N$
proof –
have ($case\ (S_1, N)\ of\ (ms, lss) \Rightarrow if\ (ms, lss) = (Ms, N)\ then\ (Ms, N)\ else\ DPLL\text{-}ci\ ms\ N$)
 $= DPLL\text{-}ci\ Ms\ N$
using $S\ DPLL\text{-}ci.simps[of\ Ms\ N]\ calculation\ inv$ **by** $presburger$
then have ($if\ (S_1, N) = (Ms, N)\ then\ (Ms, N)\ else\ DPLL\text{-}ci\ S_1\ N = DPLL\text{-}ci\ Ms\ N$)
by $fastforce$
then show $?thesis$
using $calculation\ n$ **by** $presburger$
qed
moreover
have $DPLL\text{-}ci\ S_1'\ N = (S_1', N)$ **using** $step\ IH[OF\ -\ -\ S\ n\ SS[symmetric]]\ inv$ **by** $blast$
ultimately have $?case$ **using** $step$ **by** $fastforce$
}
ultimately show $?case$ **by** $blast$

qed

lemma *DPLL-part-dpll_W-all-inv-final*:

fixes *M Ms':: (int, unit) ann-lits and*

N :: int literal list list

assumes *inv: dpll_W-all-inv (Ms, mset (map mset N))*

and *MsN: DPLL-part Ms N = (Ms', N)*

shows *conclusive-dpll_W-state (toS Ms' N) ∧ dpll_W** (toS Ms N) (toS Ms' N)*

proof –

have *2: DPLL-ci Ms N = DPLL-part Ms N using inv dpll_W-all-inv-implieS-2-eq3-and-dom by blast*

then have *star: dpll_W** (toS Ms N) (toS Ms' N) unfolding MsN using DPLL-ci-dpll_W-rtrancpl*

by blast

then have *inv': dpll_W-all-inv (toS Ms' N) using inv rtrancpl-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) ann-lits, N::int literal list list).

dpll_W-all-inv (toS M N))}

morphisms *rough-state-of state-of*

proof

show *([],[]) ∈ {(M, N). dpll_W-all-inv (toS M N)} by (auto simp add: dpll_W-all-inv-def)*

qed

lemma

DPLL-part-dom ([], N)

using *dpll_W-all-inv-implieS-2-eq3-and-dom[of [] N] by (simp add: dpll_W-all-inv-def)*

Some type classes **instantiation** *dpll_W-state :: equal*

begin

definition *equal-dpll_W-state :: dpll_W-state ⇒ dpll_W-state ⇒ 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 ⇒ 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) ∈ {(M, N). dpll_W-all-inv (toS M N)}

by *(smt DPLL-ci.simps DPLL-ci-dpll_W-rtrancpl case-prodE case-prodI2 rough-state-of*

mem-Collect-eq old.prod.case prod.sel(2) rtrancpl-dpll_W-all-inv snd-DPLL-step)

lemma *rough-state-of-DPLL-step'-DPLL-step[simp]:*

rough-state-of (DPLL-step' S) = DPLL-step (rough-state-of S)

using *DPLL-step-dpll_W-conc-inv DPLL-step'-def state-of-inverse by auto*

function *DPLL-tot:: dpll_W-state ⇒ dpll_W-state where*

$DPLL\text{-}tot\ S =$
 (let $S' = DPLL\text{-}step'\ S$ in
 if $S' = S$ then S else $DPLL\text{-}tot\ S'$)
 by *fast+*
termination
proof (relation $\{(T', T)\}$.
 (rough-state-of T' , rough-state-of T)
 $\in \{(S', S). (toS'\ S', toS'\ S)$
 $\in \{(S', S). dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}\})$
show wf $\{(b, a)\}$.
 (rough-state-of b , rough-state-of a)
 $\in \{(b, a). (toS'\ b, toS'\ a)$
 $\in \{(b, a). dpll_W\text{-}all\text{-}inv\ a \wedge dpll_W\ a\ b\}\})$
 using wf-if-measure-f[OF wf-if-measure-f[OF $dpll_W\text{-}wf$, of toS'], of rough-state-of] .
next
fix $S\ x$
assume $x: x = DPLL\text{-}step'\ S$
and $x \neq S$
have $dpll_W\text{-}all\text{-}inv$ (case rough-state-of S of $(Ms, N) \Rightarrow (Ms, mset\ (map\ mset\ N))$)
 by (metis (no-types, lifting) case-prodE mem-Collect-eq old.prod.case rough-state-of)
moreover have $dpll_W$ (case rough-state-of S of $(Ms, N) \Rightarrow (Ms, mset\ (map\ mset\ N))$)
 (case rough-state-of $(DPLL\text{-}step'\ S)$ of $(Ms, N) \Rightarrow (Ms, mset\ (map\ mset\ N))$)
proof –
obtain $Ms\ N$ **where** $Ms: (Ms, N) = \text{rough-state-of}\ S$ **by** (cases rough-state-of S) *auto*
have $dpll_W\text{-}all\text{-}inv$ ($toS'\ (Ms, N)$) **using** *calculation* **unfolding** Ms **by** *blast*
moreover obtain $Ms'\ N'$ **where** $Ms': (Ms', N') = \text{rough-state-of}\ (DPLL\text{-}step'\ S)$
 by (cases rough-state-of $(DPLL\text{-}step'\ S)$) *auto*
ultimately have $dpll_W\text{-}all\text{-}inv$ ($toS'\ (Ms', N')$) **unfolding** Ms'
 by (metis (no-types, lifting) case-prod-unfold mem-Collect-eq rough-state-of)

have $dpll_W$ ($toS\ Ms\ N$) ($toS\ Ms'\ N'$)
 apply (rule $DPLL\text{-}step\text{-}is\text{-}a\text{-}dpll_W\text{-}step$ [of $Ms'\ N'\ Ms\ N$])
 unfolding $Ms\ Ms'$ **using** $\langle x \neq S \rangle$ rough-state-of-inject x **by** *fastforce+*
then show ?thesis **unfolding** $Ms[symmetric]\ Ms'[symmetric]$ **by** *auto*
qed
ultimately show $(x, S) \in \{(T', T). (\text{rough-state-of}\ T', \text{rough-state-of}\ T)$
 $\in \{(S', S). (toS'\ S', toS'\ S) \in \{(S', S). dpll_W\text{-}all\text{-}inv\ S \wedge dpll_W\ S\ S'\}\})$
by (*auto simp add: x*)
qed

lemma [code]:

$DPLL\text{-}tot\ S =$
 (let $S' = DPLL\text{-}step'\ S$ in
 if $S' = S$ then S else $DPLL\text{-}tot\ S'$) **by** *auto*

lemma $DPLL\text{-}tot\text{-}DPLL\text{-}step\text{-}DPLL\text{-}tot$ [simp]: $DPLL\text{-}tot\ (DPLL\text{-}step'\ S) = DPLL\text{-}tot\ S$
apply (cases $DPLL\text{-}step'\ S = S$)
apply *simp*
unfolding $DPLL\text{-}tot.simps$ [of S] **by** (*simp del: DPLL-tot.simps*)

lemma $DOPLL\text{-}step'\text{-}DPLL\text{-}tot$ [simp]:

$DPLL\text{-}step'\ (DPLL\text{-}tot\ S) = DPLL\text{-}tot\ S$
by (rule $DPLL\text{-}tot.induct$ [of $\lambda S. DPLL\text{-}step'\ (DPLL\text{-}tot\ S) = DPLL\text{-}tot\ S$])
 (metis (full-types) $DPLL\text{-}tot.simps$)

lemma *DPLL-tot-final-state*:
assumes *DPLL-tot* $S = S$
shows *conclusive-dpll_W-state* (*toS'* (*rough-state-of* S))
proof –
have *DPLL-step'* $S = S$ **using** *assms*[*symmetric*] *DOPLL-step'-DPLL-tot* **by** *metis*
then have *DPLL-step* (*rough-state-of* S) = (*rough-state-of* S)
unfolding *DPLL-step'-def* **using** *DPLL-step-dpll_W-conc-inv* *rough-state-of-inverse*
by (*metis* *rough-state-of-DPLL-step'-DPLL-step*)
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *DPLL-step-stuck-final-state* *old.prod.exhaust* *split-conv*)
qed

lemma *DPLL-tot-star*:
assumes *rough-state-of* (*DPLL-tot* S) = S'
shows *dpll_W*** (*toS'* (*rough-state-of* S)) (*toS'* S')
using *assms*
proof (*induction arbitrary: S' rule: DPLL-tot.induct*)
case (1 $S S'$)
let $?x = \text{DPLL-step}' S$
{ assume $?x = S$
then have *?case* **using** 1(2) **by** *simp*
}
moreover {
assume $S: ?x \neq S$
have *?case*
apply (*cases* *DPLL-step'* $S = S$)
using S **apply** *blast*
by (*smt* 1.IH 1.prem *DPLL-step-is-a-dpll_W-step* *DPLL-tot.simps* *case-prodE2*
rough-state-of-DPLL-step'-DPLL-step *rtranclp.rtrancl-into-rtrancl* *rtranclp.rtrancl-refl*
rtranclp-idemp *split-conv*)
}
ultimately show *?case* **by** *auto*
qed

lemma *rough-state-of-rough-state-of-Nil*[*simp*]:
rough-state-of (*state-of* (\square , N)) = (\square , N)
apply (*rule* *DPLL-W-Implementation.dpll_W-state.state-of-inverse*)
unfolding *dpll_W-all-inv-def* **by** *auto*

Theorem of correctness

lemma *DPLL-tot-correct*:
assumes *rough-state-of* (*DPLL-tot* (*state-of* (\square , N)))) = (M , N')
and (M' , N'') = *toS'* (M , N')
shows $M' \models_{asm} N'' \longleftrightarrow \text{satisfiable} (\text{set-mset } N'')$
proof –
have *dpll_W*** (*toS'* (\square , N)) (*toS'* (M , N'))) **using** *DPLL-tot-star*[*OF* *assms*(1)] **by** *auto*
moreover have *conclusive-dpll_W-state* (*toS'* (M , N')))
using *DPLL-tot-final-state* **by** (*metis* (*mono-tags*, *lifting*) *DOPLL-step'-DPLL-tot* *DPLL-tot.simps*
assms(1))
ultimately show *?thesis* **using** *dpll_W-conclusive-state-correct* **by** (*smt* *DPLL-ci.simps*
DPLL-ci-dpll_W-rtranclp *assms*(2) *dpll_W-all-inv-def* *prod.case* *prod.sel*(1) *prod.sel*(2)
rtranclp-dpll_W-inv(3) *rtranclp-dpll_W-inv-starting-from-0*)
qed

Code export

A conversion to $DPLL$ - W -Implementation. $dpll_W$ -state **definition** $Con :: (int, unit) \text{ ann-lits} \times \text{int literal list list}$

```

     $\Rightarrow dpll_W$ -state where
     $Con\ xs = \text{state-of } (if\ dpll_W\text{-all-inv } (toS\ (fst\ xs)\ (snd\ xs))\ \text{then } xs\ \text{else } ([], []))$ 
lemma [code abstype]:
     $Con\ (\text{rough-state-of } S) = S$ 
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]:
     $Con\ (DPLL\text{-step } (\text{rough-state-of } s)) = \text{state-of } (DPLL\text{-step } (\text{rough-state-of } s))$ 
unfolding  $Con$ -def by (metis (mono-tags, lifting)  $DPLL$ -step- $dpll_W$ -conc-inv mem-Collect-eq
    prod.case-eq-if)
```

A slightly different version of $DPLL$ -tot where the returned boolean indicates the result.

definition $DPLL$ -tot-rep **where**

```

 $DPLL$ -tot-rep  $S =$ 
    (let  $(M, N) = (\text{rough-state-of } (DPLL\text{-tot } S))$  in  $(\forall A \in \text{set } N. (\exists a \in \text{set } A. a \in \text{lits-of-l } (M)), M)$ )
```

One version of the generated SML code is here, but not included in the generated document. The only differences are:

- export 'a literal from the SML Module *Clausal-Logic*;
- export the constructor Con from $DPLL$ - W -Implementation;
- export the int constructor from *Arith*.

All these allows to test on the code on some examples.

end

theory $CDCL$ - W -Implementation

imports $DPLL$ - $CDCL$ - W -Implementation $CDCL$ - W -Termination

begin

7.1.4 List-based CDCL Implementation

We here have a very simple implementation of Weidenbach's CDCL, based on the same principle as the implementation of DPLL: iterating over-and-over on lists. We do not use any fancy data-structure (see the two-watched literals for a better suited data-structure).

The goal was (as for DPLL) to test the infrastructure and see if an important lemma was missing to prove the correctness and the termination of a simple implementation.

Types and Instantiation

notation $image$ -mset (**infixr** '# 90)

type-synonym 'a $cdcl_W$ -restart-mark = 'a clause

type-synonym 'v $cdcl_W$ -restart-ann-lit = ('v, 'v $cdcl_W$ -restart-mark) ann-lit

type-synonym 'v $cdcl_W$ -restart-ann-lits = ('v, 'v $cdcl_W$ -restart-mark) ann-lits

type-synonym 'v $cdcl_W$ -restart-state =

$'v \text{ cdcl}_W\text{-restart-ann-lits} \times 'v \text{ clauses} \times 'v \text{ clauses} \times \text{nat} \times 'v \text{ clause option}$

abbreviation $\text{raw-trail} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \text{ where}$

$\text{raw-trail} \equiv (\lambda(M, -). M)$

abbreviation $\text{raw-cons-trail} :: 'a \Rightarrow 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \text{ list} \times 'b \times 'c \times 'd \times 'e$
where

$\text{raw-cons-trail} \equiv (\lambda L (M, S). (L \# M, S))$

abbreviation $\text{raw-tl-trail} :: 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \text{ where}$

$\text{raw-tl-trail} \equiv (\lambda(M, S). (\text{tl } M, S))$

abbreviation $\text{raw-init-clss} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b \text{ where}$

$\text{raw-init-clss} \equiv \lambda(M, N, -). N$

abbreviation $\text{raw-learned-clss} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c \text{ where}$

$\text{raw-learned-clss} \equiv \lambda(M, N, U, -). U$

abbreviation $\text{raw-backtrack-lvl} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'd \text{ where}$

$\text{raw-backtrack-lvl} \equiv \lambda(M, N, U, k, -). k$

abbreviation $\text{raw-update-backtrack-lvl} :: 'd \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where

$\text{raw-update-backtrack-lvl} \equiv \lambda k (M, N, U, -, S). (M, N, U, k, S)$

abbreviation $\text{raw-conflicting} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e \text{ where}$

$\text{raw-conflicting} \equiv \lambda(M, N, U, k, D). D$

abbreviation $\text{raw-update-conflicting} :: 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where

$\text{raw-update-conflicting} \equiv \lambda S (M, N, U, k, -). (M, N, U, k, S)$

abbreviation $S0\text{-cdcl}_W\text{-restart } N \equiv (([], N, \{\#\}, 0, \text{None})) :: 'v \text{ cdcl}_W\text{-restart-state}$

abbreviation $\text{raw-add-learned-clss} \text{ where}$

$\text{raw-add-learned-clss} \equiv \lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$

abbreviation $\text{raw-remove-clss} \text{ where}$

$\text{raw-remove-clss} \equiv \lambda C (M, N, U, S). (M, \text{removeAll-mset } C \ N, \text{removeAll-mset } C \ U, S)$

lemma $\text{raw-trail-conv: raw-trail } (M, N, U, k, D) = M \text{ and}$

$\text{clauses-conv: raw-init-clss } (M, N, U, k, D) = N \text{ and}$

$\text{raw-learned-clss-conv: raw-learned-clss } (M, N, U, k, D) = U \text{ and}$

$\text{raw-conflicting-conv: raw-conflicting } (M, N, U, k, D) = D \text{ and}$

$\text{raw-backtrack-lvl-conv: raw-backtrack-lvl } (M, N, U, k, D) = k$

by auto

lemma state-conv:

$S = (\text{raw-trail } S, \text{raw-init-clss } S, \text{raw-learned-clss } S, \text{raw-backtrack-lvl } S, \text{raw-conflicting } S)$

by (cases S) auto

interpretation state_W

$\text{raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting}$

$\lambda L (M, S). (L \# M, S)$

$\lambda(M, S). (\text{tl } M, S)$

```

λC (M, N, U, S). (M, N, {#C#} + U, S)
λC (M, N, U, S). (M, removeAll-mset C N, removeAll-mset C U, S)
λ(k::nat) (M, N, U, -, D). (M, N, U, k, D)
λD (M, N, U, k, -). (M, N, U, k, D)
λN. ([], N, {#}, 0, None)
by unfold-locales auto

```

interpretation *conflict-driven-clause-learning_W raw-trail raw-init-clss raw-learned-clss raw-backtrack-lvl raw-conflicting*

```

λL (M, S). (L # M, S)
λ(M, S). (tl M, S)
λC (M, N, U, S). (M, N, {#C#} + U, S)
λC (M, N, U, S). (M, removeAll-mset C N, removeAll-mset C U, S)
λ(k::nat) (M, N, U, -, D). (M, N, U, k, D)
λD (M, N, U, k, -). (M, N, U, k, D)
λN. ([], N, {#}, 0, None)
by unfold-locales auto

```

declare *clauses-def[simp]*

lemma *cdcl_W-restart-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]*

lemma *reduce-trail-to-empty-trail[simp]*:
reduce-trail-to F ([], *aa*, *ab*, *ac*, *b*) = ([], *aa*, *ab*, *ac*, *b*)
using *reduce-trail-to.simps* **by** *auto*

lemma *raw-trail-reduce-trail-to-length-le*:
assumes *length F > length (raw-trail S)*
shows *raw-trail (reduce-trail-to F S) = []*
using *assms trail-reduce-trail-to-length-le[of S F]*
by (*cases S*, *cases reduce-trail-to F S*) *auto*

lemma *reduce-trail-to*:
reduce-trail-to F S =
 ((if *length (raw-trail S) ≥ length F*
 then *drop (length (raw-trail S) - length F) (raw-trail S)*
 else []) , *raw-init-clss S*, *raw-learned-clss S*, *raw-backtrack-lvl S*, *raw-conflicting S*)
 (is ?S = -)

proof (*induction F S rule: reduce-trail-to.induct*)
case (1 *F S*) **note** *IH = this*
show ?*case*
proof (*cases raw-trail S*)
case *Nil*
then show ?*thesis* **using** *IH* **by** (*cases S*) *auto*
next
case (*Cons L M*)
then show ?*thesis*
apply (*cases Suc (length M) > length F*)
prefer 2 **using** *IH reduce-trail-to-length-ne[of S F]* **apply** (*cases S*) **apply** *auto*
apply (*subgoal-tac Suc (length M) - length F = Suc (length M - length F)*)
using *reduce-trail-to-length-ne[of S F] IH* **by** (*cases S*) *auto*
qed
qed

7.1.5 CDCL Implementation

Definition of the rules

Types lemma *true-raw-init-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*

type-synonym *'v cdcl_W-restart-state-inv-st* = (*'v*, *'v literal list*) *ann-lit list* ×
'v literal list list × *'v literal list list* × *nat* × *'v literal list option*

We need some functions to convert between our abstract state *'v cdcl_W-restart-state* and the concrete state *'v cdcl_W-restart-state-inv-st*.

fun *convert* :: (*'a*, *'c list*) *ann-lit* ⇒ (*'a*, *'c multiset*) *ann-lit* **where**
convert (*Propagated L C*) = *Propagated L (mset C)* |
convert (*Decided K*) = *Decided K*

abbreviation *convertC* :: *'a list option* ⇒ *'a multiset option* **where**
convertC ≡ *map-option mset*

lemma *convert-Propagated*[elim!]:
 $convert\ z = Propagated\ L\ C \implies (\exists\ C'.\ z = Propagated\ L\ C' \wedge C = mset\ C')$
by (*cases z*) *auto*

lemma *is-decided-convert*[simp]: *is-decided (convert x)* = *is-decided x*
by (*cases x*) *auto*

lemma *get-level-map-convert*[simp]:
 $get_level\ (map\ convert\ M)\ x = get_level\ M\ x$
by (*induction M rule: ann-lit-list-induct*) (*auto simp: comp-def*)

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)

Conversion function

fun *toS* :: *'v cdcl_W-restart-state-inv-st* ⇒ *'v cdcl_W-restart-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 *'v cdcl_W-restart-state-inv* = {*S*::*'v cdcl_W-restart-state-inv-st. cdcl_W-restart-all-struct-inv (toS S)*}

morphisms *rough-state-of state-of*

proof

show (*[]*, *[]*, *[]*, *0*, *None*) ∈ {*S. cdcl_W-restart-all-struct-inv (toS S)*}

by (*auto simp add: cdcl_W-restart-all-struct-inv-def*)

qed

instantiation *cdcl_W-restart-state-inv* :: (*type*) *equal*
begin

definition *equal-cdcl_W-restart-state-inv* :: 'v cdcl_W-restart-state-inv \Rightarrow 'v cdcl_W-restart-state-inv \Rightarrow bool
where

equal-cdcl_W-restart-state-inv *S S'* = (*rough-state-of* *S* = *rough-state-of* *S'*)

instance

by *standard* (*simp add: rough-state-of-inject equal-cdcl_W-restart-state-inv-def*)
end

lemma *lits-of-map-convert[*simp*]*: *lits-of-l* (*map convert* *M*) = *lits-of-l* *M*

by (*induction* *M* *rule: ann-lit-list-induct*) *simp-all*

lemma *atm-lit-of-convert[*simp*]*:

lit-of (*convert* *x*) = *lit-of* *x*

by (*cases* *x*) *auto*

lemma *undefined-lit-map-convert[*iff*]*:

undefined-lit (*map convert* *M*) *L* \longleftrightarrow *undefined-lit* *M* *L*

by (*auto simp add: defined-lit-map image-image*)

lemma *true-annot-map-convert[*simp*]*: *map convert* *M* \models_a *N* \longleftrightarrow *M* \models_a *N*

by (*simp-all add: true-annot-def image-image lits-of-def*)

lemma *true-annots-map-convert[*simp*]*: *map convert* *M* \models_{as} *N* \longleftrightarrow *M* \models_{as} *N*

unfolding *true-annots-def* **by** *auto*

lemmas *propagateE*

lemma *find-first-unit-clause-some-is-propagate*:

assumes *H*: *find-first-unit-clause* (*N* @ *U*) *M* = *Some* (*L*, *C*)

shows *propagate* (*toS* (*M*, *N*, *U*, *k*, *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#}])

The Transitions

Propagate **definition** *do-propagate-step* **where**

do-propagate-step *S* =

(*case* *S* *of*

(*M*, *N*, *U*, *k*, *None*) \Rightarrow

(*case* *find-first-unit-clause* (*N* @ *U*) *M* *of*

Some (*L*, *C*) \Rightarrow (*Propagated* *L* *C* # *M*, *N*, *U*, *k*, *None*)

| *None* \Rightarrow (*M*, *N*, *U*, *k*, *None*))

| *S* \Rightarrow *S*)

lemma *do-propagate-step*:

do-propagate-step *S* \neq *S* \implies *propagate* (*toS* *S*) (*toS* (*do-propagate-step* *S*))

apply (*cases* *S*, *cases raw-conflicting* *S*)

using *find-first-unit-clause-some-is-propagate*[*of raw-init-clss* *S* *raw-learned-clss* *S* *raw-trail* *S* - -
raw-backtrack-lvl *S*]

by (*auto simp add: do-propagate-step-def split: option.splits*)

lemma *do-propagate-step-option[*simp*]*:

raw-conflicting *S* \neq *None* \implies *do-propagate-step* *S* = *S*

unfolding *do-propagate-step-def* **by** (*cases* *S*, *cases raw-conflicting* *S*) *auto*

lemma *do-propagate-step-no-step*:

assumes *dist*: $\forall c \in \text{set } (\text{raw-init-clss } S @ \text{raw-learned-clss } S). \text{distinct } c$ **and**

```

prop-step: do-propagate-step  $S = S$ 
shows no-step propagate (toS  $S$ )
proof (standard, standard)
  fix  $T$ 
  assume propagate (toS  $S$ )  $T$ 
  then obtain  $M\ N\ U\ k\ C\ L\ E$  where
    toSS:  $\text{toS } S = (M, N, U, k, \text{None})$  and
    LE:  $L \in \# E$  and
    T:  $T = (\text{Propagated } L\ E\ \# M, N, U, k, \text{None})$  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 elim!: propagateE)
  let  $?M = \text{raw-trail } S$ 
  let  $?N = \text{raw-init-clss } S$ 
  let  $?U = \text{raw-learned-clss } S$ 
  let  $?k = \text{raw-backtrack-lvl } S$ 
  let  $?D = \text{None}$ 
  have  $S: S = (?M, ?N, ?U, ?k, ?D)$ 
    using toSS by (cases S, cases raw-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  $\text{toSS}[\text{unfolded } S]$  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  $\text{setD } L$  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 $M\ (N \# Ns) = (\text{if } (\forall c \in \text{set } N. \neg c \in \text{lits-of-l } M) \text{ then } \text{Some } N \text{ else } \text{find-conflict } M\ Ns)$

lemma *find-conflict-Some*:

find-conflict $M\ Ns = \text{Some } N \implies N \in \text{set } Ns \wedge M \models_{\text{as}} C \text{Not } (\text{mset } N)$

by (*induction Ns rule: find-conflict.induct*)

(*auto split: if-split-asm*)

lemma *find-conflict-None*:

find-conflict M $Ns = \text{None} \longleftrightarrow (\forall N \in \text{set } Ns. \neg M \models_{as} CNot (mset\ N))$
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*
 $(M, N, U, k, \text{None}) \Rightarrow$
 $(\text{case find-conflict } M\ (N @ U) \text{ of}$
 $\text{Some } a \Rightarrow (M, N, U, k, \text{Some } a)$
 $| \text{None} \Rightarrow (M, N, U, k, \text{None}))$
 $| S \Rightarrow S)$

lemma *do-conflict-step*:

do-conflict-step $S \neq S \implies \text{conflict } (toS\ S) (toS\ (do-conflict-step\ S))$
apply (*cases* S , *cases raw-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 raw-conflicting* S)
unfolding *do-conflict-step-def*
using *find-conflict-None-no-conf*[*of raw-trail* S *raw-init-clss* S *raw-learned-clss* S
raw-backtrack-lvl S]
by (*auto split: option.splits elim!: conflictE*)

lemma *do-conflict-step-option[simp]*:

raw-conflicting $S \neq \text{None} \implies do-conflict-step\ S = S$
unfolding *do-conflict-step-def* **by** (*cases* S , *cases raw-conflicting* S) *auto*

lemma *do-conflict-step-raw-conflicting[dest]*:

do-conflict-step $S \neq S \implies \text{raw-conflicting } (do-conflict-step\ S) \neq \text{None}$
unfolding *do-conflict-step-def* **by** (*cases* S , *cases raw-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-raw-conflicting do-cp-step-def*)

next

case *False*

then have *cnfl[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 ? $S = \text{toS } S$ 
    let ? $T = \text{toS } (\text{do-propagate-step } S)$ 
    let ? $U = \text{toS } (\text{do-conflict-step } (\text{do-propagate-step } S))$ 
    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-confl:
 $\text{do-cp-step } S = S \implies \text{do-conflict-step } S = S \wedge \text{do-propagate-step } S = S$
by (cases S , cases raw-conflicting S)
(auto simp add: do-conflict-step-def do-propagate-step-def do-cp-step-def split: option.splits)

lemma no-cdcl_W-cp-iff-no-propagate-no-conflict:
 $\text{no-step cdcl}_W\text{-cp } S \longleftrightarrow \text{no-step propagate } S \wedge \text{no-step conflict } S$
by (auto simp: cdcl_W-cp.simps)

lemma do-cp-step-eq-no-step:
assumes H : do-cp-step $S = S$ **and** $\forall c \in \text{set } (\text{raw-init-clss } S @ \text{raw-learned-clss } S)$. distinct c
shows no-step cdcl_W-cp (toS S)
unfolding no-cdcl_W-cp-iff-no-propagate-no-conflict
using assms **apply** (cases S , cases raw-conflicting S)
using do-propagate-step-no-step[of S]
by (auto dest!: do-cp-step-eq-no-prop-no-confl[simplified] do-conflict-step-no-step
split: option.splits)

lemma cdcl_W-cp-cdcl_W-restart-st: cdcl_W-cp $S S' \implies \text{cdcl}_W\text{-restart}^{**} S S'$
by (simp add: cdcl_W-cp-tranclp-cdcl_W-restart tranclp-into-rtranclp)

lemma cdcl_W-restart-all-struct-inv-rough-state[simp]: cdcl_W-restart-all-struct-inv (toS (rough-state-of S))
using rough-state-of **by** auto

lemma [simp]: cdcl_W-restart-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]:
 $\text{rough-state-of } (\text{state-of } (\text{do-cp-step } (\text{rough-state-of } S))) = \text{do-cp-step } (\text{rough-state-of } S)$
proof –
have cdcl_W-restart-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-restart-all-struct-inv-rough-state[of S]
cdcl_W-cp-cdcl_W-restart-st rtranclp-cdcl_W-restart-all-struct-inv-inv **by** blast
then show ?thesis **by** auto
qed

Skip fun *do-skip-step* :: 'v *cdcl_W-restart-state-inv-st* \Rightarrow 'v *cdcl_W-restart-state-inv-st* **where**
do-skip-step (*Propagated* L C # Ls, N, U, k, *Some* D) =
 (if $\neg L \in \text{set } D \wedge D \neq []$
 then (Ls, N, U, k, *Some* D)
 else (*Propagated* L C # Ls, N, U, k, *Some* D)) |
do-skip-step S = S

lemma *do-skip-step*:
do-skip-step S \neq S \implies *skip* (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: if-split-asm elim: skipE*)

lemma *do-skip-step-raw-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, 'a *literal list*) *ann-lit list* \Rightarrow nat
where
maximum-level-code [] = 0 |
maximum-level-code (L # Ls) M = max (get-level M L) (*maximum-level-code* Ls M)

lemma *maximum-level-code-eq-get-maximum-level*[*code, simp*]:
maximum-level-code D M = *get-maximum-level* M (*mset* D)
by (*induction* D) (*auto simp add: get-maximum-level-plus*)

fun *do-resolve-step* :: 'v *cdcl_W-restart-state-inv-st* \Rightarrow 'v *cdcl_W-restart-state-inv-st* **where**
do-resolve-step (*Propagated* L C # Ls, N, U, k, *Some* D) =
 (if $\neg L \in \text{set } D \wedge \text{maximum-level-code } (\text{remove1 } (\neg L) D) (\text{Propagated } L C \# Ls) = k$
 then (Ls, N, U, k, *Some* (*remdups* (*remove1* L C @ *remove1* ($\neg L$) D)))
 else (*Propagated* L C # Ls, N, U, k, *Some* D)) |
do-resolve-step S = S

lemma *do-resolve-step*:
cdcl_W-restart-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* ($\neg L$) D) (*Propagated* L C # M) = k

by (*cases mset* D – {#– L#} = {#},

auto dest!: *get-maximum-level-exists-lit-of-max-level*[of - *Propagated* L C # M]

split: if-split-asm) +

have *every-mark-is-a-conflict* (toS (*Propagated* L C # M, N, U, k, *Some* D))

using 1(1) **unfolding** *cdcl_W-restart-all-struct-inv-def* *cdcl_W-restart-conflicting-def* **by** *fast*

then have L \in *set* C **by** *fastforce*

then obtain C' **where** C: *mset* C = C' + {#L#}

by (*metis add.commute in-multiset-in-set insert-DiffM*)

obtain D' **where** D: *mset* D = D' + {#– L#}

using $\langle \neg L \in \text{set } D \rangle$ **by** (*metis add.commute in-multiset-in-set insert-DiffM*)

have D'L: D' + {#– L#} – {#– L#} = D' **by** (*auto simp add: multiset-eq-iff*)


```

have CL: mset C - {#L#} + {#L#} = mset C using ⟨L ∈ 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#}) # ∪ (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 ⟨cdclW-restart-all-struct-inv (toS (Propagated L C # M, N, U, k, Some D))⟩
    unfolding cdclW-restart-all-struct-inv-def distinct-cdclW-restart-state-def
    by auto
  then have (mset C - {#L#}) # ∪ (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#}) # ∪ (mset C - {#L#})))
  = toS (do-resolve-step (Propagated L C # M, N, U, k, Some D))
    using ⟨- L ∈ 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 ⟹ no-step resolve (toS S)
apply (cases S; cases hd (raw-trail S); cases raw-trail S; cases raw-conflicting S)
by (auto
  elim!: resolveE split: if-split-asm
  dest!: union-single-eq-member
  simp del: in-multiset-in-set get-maximum-level-map-convert
  simp: get-maximum-level-map-convert[symmetric])

```

lemma *rough-state-of-state-of-resolve[simp]:*

```

cdclW-restart-all-struct-inv (toS S) ⟹ rough-state-of (state-of (do-resolve-step S)) = do-resolve-step S
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-restart-all-struct-inv-inv)

```

lemma *do-resolve-step-raw-trail-is-None[iff]:*

```

do-resolve-step S = (a, b, c, d, None) ⟷ S = (a, b, c, d, None)
by (cases S rule: do-resolve-step.cases) auto

```

Backjumping lemma *get-all-ann-decomposition-map-convert:*

```

(get-all-ann-decomposition (map convert M)) =
  map (λ(a, b). (map convert a, map convert b)) (get-all-ann-decomposition M)

```

```

apply (induction M rule: ann-lit-list-induct)
  apply simp
by (rename-tac L xs, case-tac get-all-ann-decomposition xs; auto)+

lemma do-backtrack-step:
assumes
  db: do-backtrack-step S  $\neq$  S and
  inv: cdclW-restart-all-struct-inv (toS S)
shows backtrack (toS S) (toS (do-backtrack-step S))
proof (cases S, cases raw-conflicting S, goal-cases)
  case (1 M N U k E)
  then show ?case using db by auto
next
  case (2 M N U k E C) note S = this(1) and confl = this(2)
  have E: E = Some C using S confl by auto

  obtain L j where fd: find-level-decomp M C [] k = Some (L, j)
  using db unfolding S E by (cases C) (auto split: if-split-asm option.splits list.splits
    ann-lit.splits)
  have
    L  $\in$  set C and
    j: get-maximum-level M (mset (remove1 L C)) = j and
    levL: get-level M L = k
  using find-level-decomp-some[OF fd] by auto
  obtain C' where C: mset C = mset C' + {#L#}
  using ⟨L  $\in$  set C⟩ by (metis add.commute ex-mset in-multiset-in-set insert-DiffM)
  obtain M2 where M2: bt-cut j M = Some M2
  using db fd unfolding S E by (auto split: option.splits)
  have no-dup M and k: k = count-decided (filter is-decided M)
  using inv unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def S by (auto simp:
    comp-def)
  then obtain M1 K c where
    M1: M2 = Decided K # M1 and lev-K: get-level M K = j + 1 and
    c: M = c @ M2
  using bt-cut-some-decomp[OF - M2] by (cases M2) auto
  have j  $\leq$  k unfolding c j[symmetric] k
  by (metis (mono-tags, lifting) count-decided-ge-get-maximum-level filter-cong filter-filter)
  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 ann-lit.splits
    dest!: find-level-decomp-some)[1]
  have get-maximum-level M (mset C)  $\geq$  k
  using ⟨L  $\in$  set C⟩ levL get-maximum-level-ge-get-level by (metis set-mset-mset)
  moreover have get-maximum-level M (mset C)  $\leq$  k
  using get-maximum-level-exists-lit-of-max-level[of mset C M] inv
    cdclW-restart-M-level-inv-get-level-le-backtrack-lvl[of toS S]
  unfolding C cdclW-restart-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')  $\in$  set (get-all-ann-decomposition M)
  using bt-cut-in-get-all-ann-decomposition[OF ⟨no-dup M⟩ M2] by metis
  have decomp:
    (Decided K # (map convert M1),
    (map convert M2'))  $\in$ 
    set (get-all-ann-decomposition (map convert M))
  using imageI[of - -  $\lambda(a, b).$  (map convert a, map convert b), OF M2] j

```

```

    unfolding S E M1 by (simp add: get-all-ann-decomposition-map-convert)
  show ?case
    apply (rule backtrack-rule)
      using M2 fd confl (L ∈ set C) j decomp levL (get-maximum-level M (mset C) = k)
      unfolding S E M1 apply (auto simp: mset-map)[6]
      using M2' M2 fd j lev-K unfolding S E M1 CDCL-W-Implementation.state-eq-def
      by (auto simp: comp-def ac-simps)[2]
qed

```

lemma *map-eq-list-length*:
 $\text{map } f \ L = L' \implies \text{length } L = \text{length } L'$
 by auto

lemma *map-mmset-of-mlit-eq-cons*:
 assumes $\text{map convert } M = a \ @ \ c$
 obtains $a' \ c'$ where
 $M = a' \ @ \ c'$ and
 $a = \text{map convert } a'$ and
 $c = \text{map convert } c'$
 using that[of take (length a) M drop (length a) M]
 assms by (metis append-eq-conv-conj append-take-drop-id drop-map take-map)

lemma *Decided-convert-iff*:
 $\text{Decided } K = \text{convert } za \longleftrightarrow za = \text{Decided } K$
 by (cases za) auto

lemma *do-backtrack-step-no*:
 assumes
 $db: \text{do-backtrack-step } S = S$ and
 $inv: \text{cdcl}_W\text{-restart-all-struct-inv } (toS \ S)$
 shows *no-step backtrack* (toS S)
proof (rule ccontr, cases S, cases raw-conflicting S, goal-cases)
 case 1
 then show ?case using db by (auto split: option.splits elim: backtrackE)
next
 case (2 M N U k E C) note bt = this(1) and S = this(2) and confl = this(3)
 obtain K j M1 M2 L D where
 $CE: \text{raw-conflicting } S = \text{Some } D$ and
 $LD: L \in \# \text{ mset } D$ and
 $\text{decomp}: (\text{Decided } K \ \# \ M1, M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{raw-trail } S))$ and
 $\text{levL}: \text{get-level } (\text{raw-trail } S) \ L = \text{raw-backtrack-lvl } S$ and
 $k: \text{get-level } (\text{raw-trail } S) \ L = \text{get-maximum-level } (\text{raw-trail } S) \ (\text{mset } D)$ and
 $j: \text{get-maximum-level } (\text{raw-trail } S) \ (\text{remove1-mset } L \ (\text{mset } D)) \equiv j$ and
 $\text{lev-K}: \text{get-level } (\text{raw-trail } S) \ K = \text{Suc } j$
 using bt apply clarsimp
 apply (elim backtrackE)
 apply (cases S)
 by (auto simp add: get-all-ann-decomposition-map-convert reduce-trail-to
 Decided-convert-iff)
 obtain c where $c: \text{raw-trail } S = c \ @ \ M2 \ @ \ \text{Decided } K \ \# \ M1$
 using decomp by blast
 have $k = \text{count-decided } (\text{raw-trail } S)$ and $n\text{-d}: \text{no-dup } M$
 using inv S unfolding $\text{cdcl}_W\text{-restart-all-struct-inv-def}$ $\text{cdcl}_W\text{-restart-M-level-inv-def}$
 by (auto simp: comp-def)
 then have $k > j$
 using j count-decided-ge-get-maximum-level[of raw-trail S remove1-mset L (mset D)]

```

    count-decided-ge-get-level[of K raw-trail S]
  unfolding k lev-K
  unfolding c by (auto simp: get-all-ann-decomposition-map-convert simp del: count-decided-ge-get-level)
  have [simp]: L ∈ set D
    using LD by auto
  have CD: C = D
    using CE confl by auto
  obtain D' where
    E: E = Some D and
    DD': mset D = {#L#} + mset D'
    using that[of remove1 L D]
    using S CE confl LD by (auto simp add: insert-DiffM)
  have find-level-decomp M D [] k ≠ None
    apply rule
    apply (drule find-level-decomp-none[of - - - L D])
    using DD' ⟨k > j⟩ mset-eq-setD S levL unfolding k[symmetric] j[symmetric]
    by (auto simp: ac-simps)
  then obtain L' j' where fd-some: find-level-decomp M D [] k = Some (L', j')
    by (cases find-level-decomp M D [] k) auto
  have L': L' = L
    proof (rule ccontr)
      assume ¬ ?thesis
      then have L' ∈# mset (remove1 L D)
        by (metis fd-some find-level-decomp-some in-set-remove1 set-mset-mset)
      then have get-level M L' ≤ get-maximum-level M (mset (remove1 L D))
        using get-maximum-level-ge-get-level by blast
      then show False using ⟨k > j⟩ j find-level-decomp-some[OF fd-some] S DD' by auto
    qed
  then have j': j' = j using find-level-decomp-some[OF fd-some] j S DD' by auto

  obtain c' M1' where cM: M = c' @ Decided K # M1'
  apply (rule map-mmset-of-mlit-eq-cons[of M map convert (c @ M2)
    map convert (Decided K # M1)])
  using c S apply simp
  apply (rule map-mmset-of-mlit-eq-cons[of - map convert [Decided K] map convert M1])
  apply auto[]
  apply (rename-tac a b' aa b, case-tac aa)
  apply auto[]
  apply (rename-tac a b' aa b, case-tac aa)
  by auto
  have btc-none: bt-cut j M ≠ None
  apply (rule bt-cut-not-none[of M ])
  using n-d cM S lev-K S apply blast+
  using lev-K S by auto
  show ?case using db n-d unfolding S E
  by (auto split: option.splits list.splits ann-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: cdcW-restart-all-struct-inv (toS S)
  shows rough-state-of (state-of (do-backtrack-step S)) = do-backtrack-step S
proof (rule state-of-inverse)
  consider
    (step) backtrack (toS S) (toS (do-backtrack-step S)) |

```

```

(0) do-backtrack-step S = S
using do-backtrack-step inv by blast
then show do-backtrack-step S ∈ {S. cdclW-restart-all-struct-inv (toS S)}
proof cases
  case 0
  thus ?thesis using inv by simp
next
  case step
  then show ?thesis
  using inv
  by (auto dest!: cdclW-restart.other cdclW-o.bj cdclW-bj.backtrack intro: cdclW-restart-all-struct-inv-inv)
qed
qed

```

Decide fun do-decide-step where
do-decide-step (M, N, U, k, None) =
(case find-first-unused-var N (lits-of-l M) of
 None ⇒ (M, N, U, k, None)
 | Some L ⇒ (Decided L # M, N, U, k+1, None)) |
do-decide-step S = S

lemma do-decide-step:
do-decide-step S ≠ S ⇒ decide (toS S) (toS (do-decide-step S))
apply (cases S, cases raw-conflicting S)
defer
apply (auto split: option.splits simp add: decide.simps
dest: find-first-unused-var-undefined find-first-unused-var-Some
intro: atms-of-atms-of-ms-mono)[1]

proof –
fix a :: ('a, 'a literal list) ann-lit list **and**
b :: 'a literal list list **and** c :: 'a literal list list **and**
d :: nat **and** e :: 'a literal list option
{
fix a :: ('a, 'a literal list) ann-lit list **and**
b :: 'a literal list list **and** c :: 'a literal list list **and**
d :: nat **and** x2 :: 'a literal **and** m :: 'a literal list
assume a1: m ∈ set b
assume x2 ∈ set m
then have f2: atm-of x2 ∈ atms-of (mset m)
by simp
have ∧f. (f m::'a literal multiset) ∈ f ' set b
using a1 by blast
then have ∧f. (atms-of (f m)::'a set) ⊆ atms-of-ms (f ' set b)
using atms-of-atms-of-ms-mono by blast
then have ∧n f. (n::'a) ∈ atms-of-ms (f ' set b) ∨ n ∉ atms-of (f m)
by (meson contra-subsetD)
then have atm-of x2 ∈ atms-of-ms (mset ' set b)
using f2 by blast
} **note** H = this
{
fix m :: 'a literal list **and** x2
have m ∈ set b ⇒ x2 ∈ set m ⇒ x2 ∉ lits-of-l a ⇒ – x2 ∉ lits-of-l a ⇒
∃ aa∈set b. ¬ atm-of ' set aa ⊆ atm-of ' lits-of-l a
by (meson atm-of-in-atm-of-set-in-uminus contra-subsetD rev-image-eqI)
} **note** H' = this

```

assume do-decide-step  $S \neq S$  and
   $S = (a, b, c, d, e)$  and
  raw-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 defined-lit-map lits-of-def
    image-image atm-of-eq-atm-of dest!: find-first-unused-var-Some)
qed

```

```

lemma do-decide-step-no:
  do-decide-step  $S = S \implies \text{no-step decide (toS } S)$ 
apply (cases  $S$ , cases raw-conflicting  $S$ )
apply (auto simp: atms-of-ms-mset-unfold Decided-Propagated-in-iff-in-lits-of-l lits-of-def
  dest!: atm-of-in-atm-of-set-in-uminus
  elim!: decideE
  split: option.splits) +
using atm-of-eq-atm-of by blast +

```

```

lemma rough-state-of-state-of-do-decide-step[simp]:
  cdclW-restart-all-struct-inv (toS  $S$ )  $\implies \text{rough-state-of (state-of (do-decide-step } S)) = do-decide-step } S$ 
proof (subst state-of-inverse, goal-cases)
  case 1
  then show ?case
    by (cases do-decide-step  $S = S$ )
    (auto dest: do-decide-step decide other intro: cdclW-restart-all-struct-inv-inv)
qed simp

```

```

lemma rough-state-of-state-of-do-skip-step[simp]:
  cdclW-restart-all-struct-inv (toS  $S$ )  $\implies \text{rough-state-of (state-of (do-skip-step } S)) = do-skip-step } S$ 
apply (subst state-of-inverse, cases do-skip-step  $S = S$ )
apply simp
by (blast dest: other skip bj do-skip-step cdclW-restart-all-struct-inv-inv) +

```

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  $xs = \text{state-of (if cdcl}_W\text{-restart-all-struct-inv (toS (fst xs, snd xs)) then xs}$ 
  else ( $[], [], [], 0, \text{None}$ ))

```

```

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'  $S = \text{state-of (do-cp-step (rough-state-of } S))$ 

```

```

typedef 'v cdclW-restart-state-inv-from-init-state =
  { $S:: 'v \text{ cdcl}_W\text{-restart-state-inv-st. cdcl}_W\text{-restart-all-struct-inv (toS } S)$ 
     $\wedge \text{cdcl}_W\text{-stgy}^{**} (S0\text{-cdcl}_W\text{-restart (raw-init-clss (toS } S))) (toS } S)$ }
  morphisms rough-state-from-init-state-of state-from-init-state-of
proof

```

show ($\square, \square, \square, 0, \text{None}$) $\in \{S. \text{cdcl}_W\text{-restart-all-struct-inv } (toS\ S)$
 $\wedge \text{cdcl}_W\text{-stgy}^{**} (S0\text{-cdcl}_W\text{-restart } (raw\text{-init-clss } (toS\ S))) (toS\ S)\}$
by (*auto simp add: cdcl_W-restart-all-struct-inv-def*)
qed

instantiation *cdcl_W-restart-state-inv-from-init-state* :: (type) equal
begin

definition *equal-cdcl_W-restart-state-inv-from-init-state* :: 'v *cdcl_W-restart-state-inv-from-init-state* \Rightarrow
'v *cdcl_W-restart-state-inv-from-init-state* \Rightarrow bool **where**
equal-cdcl_W-restart-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-restart-state-inv-from-init-state-def)
end

definition *ConI* **where**

ConI S = state-from-init-state-of (if *cdcl_W-restart-all-struct-inv* (*toS* (*fst S*, *snd S*))
 $\wedge \text{cdcl}_W\text{-stgy}^{**} (S0\text{-cdcl}_W\text{-restart } (raw\text{-init-clss } (toS\ S))) (toS\ S)$ then *S* else ($\square, \square, \square, 0, \text{None}$))

lemma [*code abstype*]:

ConI (rough-state-from-init-state-of *S*) = *S*
using *rough-state-from-init-state-of[of S]* **unfolding** *ConI-def*
by (*simp add: rough-state-from-init-state-of-inverse*)

definition *id-of-I-to* :: 'v *cdcl_W-restart-state-inv-from-init-state* \Rightarrow 'v *cdcl_W-restart-state-inv* **where**
id-of-I-to S = state-of (rough-state-from-init-state-of *S*)

lemma [*code abstract*]:

rough-state-of (*id-of-I-to S*) = rough-state-from-init-state-of *S*
unfolding *id-of-I-to-def* **using** *rough-state-from-init-state-of[of S]* **by** *auto*

Conflict and Propagate **function** *do-full1-cp-step* :: 'v *cdcl_W-restart-state-inv* \Rightarrow 'v *cdcl_W-restart-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 $\{(T', T). (rough\text{-state-of } T', rough\text{-state-of } T) \in \{(S', S). (toS\ S', toS\ S) \in \{(S', S). \text{cdcl}_W\text{-restart-all-struct-inv } S \wedge \text{cdcl}_W\text{-cp } S\ S')\}\}, goal\text{-cases})$

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:

$c \in \text{set } (\text{raw-init-clss } (\text{rough-state-of } S) \text{ @ raw-learned-clss } (\text{rough-state-of } S)) \implies \text{distinct } c$
apply (cases rough-state-of S)
using rough-state-of[of S] **by** (auto simp add: distinct-mset-set-distinct cdcl_W-restart-all-struct-inv-def
distinct-cdcl_W-restart-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 rough-state-of-state-of-do-cp-step **unfolding** do-cp-step'-def **by** fastforce

have f2: $\bigwedge c. (\text{if } c = \text{state-of } (\text{do-cp-step } (\text{rough-state-of } c))$
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: $\neg \text{cdcl}_W\text{-cp } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{do-cp-step } (\text{rough-state-of } S)))$

\vee state-of (do-cp-step (rough-state-of S)) = S

$\vee \text{cdcl}_W\text{-cp}^{++} (\text{toS } (\text{rough-state-of } S))$

(toS (rough-state-of (do-full1-cp-step (state-of (do-cp-step (rough-state-of S))))))

using f1 **by** (meson rtranclp-into-tranclp2)

{ **assume** do-full1-cp-step S \neq S

then have do-cp-step (rough-state-of S) = rough-state-of S

$\longrightarrow \text{cdcl}_W\text{-cp}^{**} (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } S)))$

\vee do-cp-step (rough-state-of S) \neq rough-state-of S

\wedge state-of (do-cp-step (rough-state-of S)) \neq S

using f2 f1 **by** (metis (no-types))

then have do-cp-step (rough-state-of S) \neq rough-state-of S

\wedge state-of (do-cp-step (rough-state-of S)) \neq S

$\vee \text{cdcl}_W\text{-cp}^{**} (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } S)))$

by (metis rough-state-of-state-of-do-cp-step)

then have cdcl_W-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-cdcl_W-cp tranclp-into-rtranclp) }

then show ?case

by fastforce

next

show no-step cdcl_W-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]:

rough-state-of (do-cp-step' S) = do-cp-step (rough-state-of S)

unfolding do-cp-step'-def **by** auto

The other rules **fun** do-other-step **where**

do-other-step S =

(let T = do-skip-step S in

if T \neq S


```

then T
else
  (let U = do-resolve-step T in
   if U ≠ T
   then U else
   (let V = do-backtrack-step U in
    if V ≠ U then V else do-decide-step V)))

```

lemma *do-other-step*:

assumes *inv*: *cdcl_W-restart-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: if-split-asm*
simp add: Let-def
dest!: *do-skip-step do-resolve-step do-backtrack-step do-decide-step*
dest!: *cdcl_W-o.intros cdcl_W-bj.intros*)

lemma *do-other-step-no*:

assumes *inv*: *cdcl_W-restart-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: if-split-asm elim: cdcl_W-bjE*
simp add: Let-def cdcl_W-bj.simps elim!: cdcl_W-o.cases
dest!: *do-skip-step-no do-resolve-step-no do-backtrack-step-no do-decide-step-no*)

lemma *rough-state-of-state-of-do-other-step[simp]*:

rough-state-of (*state-of* (*do-other-step* (*rough-state-of S*))) = *do-other-step* (*rough-state-of S*)

proof (*cases do-other-step* (*rough-state-of S*) = *rough-state-of S*)

case *True*

then show *?thesis* **by** *simp*

next

case *False*

have *cdcl_W-o* (*toS* (*rough-state-of S*)) (*toS* (*do-other-step* (*rough-state-of S*)))

by (*metis False cdcl_W-restart-all-struct-inv-rough-state do-other-step[of rough-state-of S]*)

then have *cdcl_W-restart-all-struct-inv* (*toS* (*do-other-step* (*rough-state-of S*)))

using *cdcl_W-restart-all-struct-inv-inv cdcl_W-restart-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-restart-all-struct-inv-inv*
cdcl_W-restart-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 $U = (\text{do-other-step}' T)$ in
 ($\text{do-full1-cp-step } U$)))

definition $\text{do-cdcl}_W\text{-stgy-step}'$ **where**

$\text{do-cdcl}_W\text{-stgy-step}' S = \text{state-from-init-state-of } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{id-of-I-to } S)))$

lemma $\text{toS-do-full1-cp-step-not-eq}$: $\text{do-full1-cp-step } S \neq S \implies$
 $\text{toS } (\text{rough-state-of } S) \neq \text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } S))$

proof –

assume $a1$: $\text{do-full1-cp-step } S \neq S$
then have $S \neq \text{do-cp-step}' S$
by *fastforce*
then show *?thesis*
by (*metis* (*no-types*) *cp-step-is-cdcl_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 $\text{do-full1-cp-step.simps}$ [*simp del*]

Correction of the transformation **lemma** $\text{do-cdcl}_W\text{-stgy-step}$:

assumes $\text{do-cdcl}_W\text{-stgy-step } S \neq S$
shows $\text{cdcl}_W\text{-stgy } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } S)))$

proof (*cases* $\text{do-full1-cp-step } S = S$)

case *False*
then show *?thesis*
using *assms do-full1-cp-step-full*[*of S*] **unfolding** *full-unfold do-cdcl_W-stgy-step-def*
by (*auto intro!*: *cdcl_W-stgy.intros dest: toS-do-full1-cp-step-not-eq*)

next

case *True*
have $\text{cdcl}_W\text{-o } (\text{toS } (\text{rough-state-of } S)) (\text{toS } (\text{rough-state-of } (\text{do-other-step}' S)))$
by (*smt True assms cdcl_W-restart-all-struct-inv-rough-state do-cdcl_W-stgy-step-def do-other-step*
rough-state-of-do-other-step' rough-state-of-inverse)

moreover

have
 np : *no-step propagate* ($\text{toS } (\text{rough-state-of } S)$) **and**
 nc : *no-step conflict* ($\text{toS } (\text{rough-state-of } S)$)
apply (*metis True do-cp-step-eq-no-prop-no-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 cdcl_W-cp* ($\text{toS } (\text{rough-state-of } S)$)
by (*simp add: cdcl_W-cp.simps*)

moreover have *full cdcl_W-cp* ($\text{toS } (\text{rough-state-of } (\text{do-other-step}' S)))$
 $(\text{toS } (\text{rough-state-of } (\text{do-full1-cp-step } (\text{do-other-step}' S))))$

using *do-full1-cp-step-full* **by** *auto*

ultimately show *?thesis*

using *assms True unfolding do-cdcl_W-stgy-step-def*
by (*auto intro!*: *cdcl_W-stgy.other' dest: toS-do-full1-cp-step-not-eq*)

qed

lemma $\text{length-raw-trail-toS}$ [*simp*]:

$\text{length } (\text{raw-trail } (\text{toS } S)) = \text{length } (\text{raw-trail } S)$

by (cases S) auto

lemma *raw-conflicting-noTrue-iff-toS[simp]*:

raw-conflicting (toS S) \neq None \longleftrightarrow raw-conflicting S \neq None

by (cases S) auto

lemma *raw-trail-toS-neq-imp-raw-trail-neq*:

raw-trail (toS S) \neq raw-trail (toS S') \implies raw-trail S \neq raw-trail S'

by (cases S, cases S') auto

lemma *do-skip-step-raw-trail-changed-or-conflict*:

assumes *d*: *do-other-step S \neq S*

and *inv*: *cdcl_W-restart-all-struct-inv (toS S)*

shows *raw-trail S \neq raw-trail (do-other-step S)*

proof –

have *M*: $\bigwedge M K M1 c. M = c @ K \# M1 \implies \text{Suc} (\text{length } M1) \leq \text{length } M$

by auto

have *cdcl_W-restart-M-level-inv (toS S)*

using *inv* **unfolding** *cdcl_W-restart-all-struct-inv-def* **by** auto

have *cdcl_W-o (toS S) (toS (do-other-step S))* **using** *do-other-step[OF inv d]* .

then show *?thesis*

using *cdcl_W-restart-M-level-inv (toS S)*

proof (*induction toS (do-other-step S) rule: cdcl_W-o-induct*)

case *decide*

then show *?thesis*

by (*auto simp add: raw-trail-toS-neq-imp-raw-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 L D K i M1 M2*) **note** *LD = this(2)* **and** *decomp = this(3)* **and** *confl-S = this(1)*
and *i = this(6)* **and** *U = this(8)*

have

bt: *raw-backtrack-lvl (toS S) = count-decided (raw-trail (toS S))* **and**

raw-trail (toS S) \models_{as} CNot D **and**

cons: *consistent-interp (lits-of-l (raw-trail (toS S)))*

using *inv confl-S* **unfolding** *cdcl_W-restart-all-struct-inv-def cdcl_W-restart-M-level-inv-def cdcl_W-restart-conflicting-def* **by** *simp-all*

then have $-L \in \text{lits-of-l (raw-trail (toS S))}$

using *LD true-annots-true-cls-def-iff-negation-in-model* **by** *blast*

then have $-L \in \text{lits-of-l (raw-trail S)}$

by (*cases S*) (*auto simp: lits-of-def*)

moreover have *consistent-interp (lits-of-l (raw-trail S))*

using *cons* **by** (*cases S*) (*auto simp: lits-of-def image-image*)

ultimately have $L \notin \text{lits-of-l (raw-trail S)}$

using *consistent-interp-def* **by** *blast*

moreover

have $L \in \text{lits-of-l (raw-trail (toS (do-other-step S)))}$

using *U* **by** *auto*

then have $L \in \text{ lits-of-l } (\text{ raw-trail } (\text{ do-other-step } S))$
by $(\text{ cases do-other-step } S) (\text{ auto simp: lits-of-def })$
ultimately show $?thesis$
by metis
qed
qed

lemma $\text{do-full1-cp-step-induct}$:
 $(\bigwedge S. (S \neq \text{ do-cp-step' } S \implies P (\text{ do-cp-step' } S)) \implies P S) \implies P a0$
using $\text{do-full1-cp-step.induct}$ **by** metis

lemma $\text{do-cp-step-neq-raw-trail-increase}$:
 $\exists c. \text{ raw-trail } (\text{ do-cp-step } S) = c @ \text{ raw-trail } S \wedge (\forall m \in \text{ set } c. \neg \text{ is-decided } m)$
by $(\text{ cases } S, \text{ cases raw-conflicting } S)$
 $(\text{ auto simp add: do-cp-step-def do-conflict-step-def do-propagate-step-def split: option.splits })$

lemma $\text{do-full1-cp-step-neq-raw-trail-increase}$:
 $\exists c. \text{ raw-trail } (\text{ rough-state-of } (\text{ do-full1-cp-step } S)) = c @ \text{ raw-trail } (\text{ rough-state-of } S)$
 $\wedge (\forall m \in \text{ set } c. \neg \text{ is-decided } m)$
apply $(\text{ induction rule: do-full1-cp-step-induct })$
apply $(\text{ rename-tac } S, \text{ case-tac do-cp-step' } S = S)$
apply $(\text{ simp add: do-full1-cp-step.simps })$
by $(\text{ smt Un-iff append-assoc do-cp-step'-def do-cp-step-neq-raw-trail-increase do-full1-cp-step.simps rough-state-of-state-of-do-cp-step set-append })$

lemma $\text{do-cp-step-raw-conflicting}$:
 $\text{ raw-conflicting } (\text{ rough-state-of } S) \neq \text{ None} \implies \text{ do-cp-step' } S = S$
unfolding $\text{do-cp-step'-def do-cp-step-def}$ **by** simp

lemma $\text{do-full1-cp-step-raw-conflicting}$:
 $\text{ raw-conflicting } (\text{ rough-state-of } S) \neq \text{ None} \implies \text{ do-full1-cp-step } S = S$
unfolding $\text{do-cp-step'-def do-cp-step-def}$
apply $(\text{ induction rule: do-full1-cp-step-induct })$
by $(\text{ rename-tac } S, \text{ case-tac } S \neq \text{ do-cp-step' } S)$
 $(\text{ auto simp add: do-full1-cp-step.simps do-cp-step-raw-conflicting })$

lemma $\text{do-decide-step-not-raw-conflicting-one-more-decide}$:
assumes
 $\text{ raw-conflicting } S = \text{ None}$ **and**
 $\text{ do-decide-step } S \neq S$
shows $\text{Suc } (\text{ length } (\text{ filter is-decided } (\text{ raw-trail } S)))$
 $= \text{ length } (\text{ filter is-decided } (\text{ raw-trail } (\text{ do-decide-step } S)))$
using $\text{assms unfolding do-other-step'-def}$
by $(\text{ cases } S) (\text{ auto simp: Let-def split: if-split-asm option.splits dest!: find-first-unused-var-Some-not-all-incl })$

lemma $\text{do-decide-step-not-raw-conflicting-one-more-decide-bt}$:
assumes $\text{ raw-conflicting } S \neq \text{ None}$ **and**
 $\text{ do-decide-step } S \neq S$
shows $\text{ length } (\text{ filter is-decided } (\text{ raw-trail } S)) < \text{ length } (\text{ filter is-decided } (\text{ raw-trail } (\text{ do-decide-step } S)))$
using $\text{assms unfolding do-other-step'-def}$ **by** $(\text{ cases } S, \text{ cases raw-conflicting } S)$
 $(\text{ auto simp add: Let-def split: if-split-asm option.splits })$

lemma $\text{count-decided-raw-trail-toS}$:
 $\text{ count-decided } (\text{ raw-trail } (\text{ toS } S)) = \text{ count-decided } (\text{ raw-trail } S)$
by $(\text{ cases } S) (\text{ auto simp: comp-def })$

lemma *do-other-step-not-raw-conflicting-one-more-decide-bt:*

assumes

raw-conflicting (rough-state-of S) \neq None **and**
raw-conflicting (rough-state-of (do-other-step' S)) = None **and**
do-other-step' $S \neq S$

shows count-decided (raw-trail (rough-state-of S))

> count-decided (raw-trail (rough-state-of (do-other-step' S)))

proof (cases S , goal-cases)

case (1 y) **note** $S = \text{this}(1)$ **and** $\text{inv} = \text{this}(2)$

obtain $M N U k E$ **where** $y: y = (M, N, U, k, \text{Some } E)$

using *assms*(1) S **inv by** (cases y , cases *raw-conflicting* y) *auto*

have M : rough-state-of (state-of ($M, N, U, k, \text{Some } E$)) = ($M, N, U, k, \text{Some } E$)

using $\text{inv } y$ **by** (*auto simp add: state-of-inverse*)

have bt : do-other-step' $S = \text{state-of}$ (do-backtrack-step (rough-state-of S))

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 (raw-trail (rough-state-of (do-other-step' S)), raw-init-clss (rough-state-of (do-other-step' S)),

raw-learned-clss (rough-state-of (do-other-step' S)),

raw-backtrack-lvl (rough-state-of (do-other-step' S)), None)

= rough-state-of (do-other-step' S)

using *assms*(2) **by** (*metis* (no-types) *state-conv*)

then show ?thesis

using $f3$ 2 **by** (*metis* (no-types) do-decide-step.simps(2) do-resolve-step-raw-trail-is-None

do-skip-step-raw-trail-is-None rough-state-of-inverse)

qed

have

bt : raw-backtrack-lvl (toS y) = count-decided (raw-trail (toS y))

using inv **unfolding** *cdcl_W-restart-all-struct-inv-def cdcl_W-restart-M-level-inv-def*

cdcl_W-restart-conflicting-def **by** *simp-all*

have *confl-y*: raw-conflicting (toS (rough-state-of (do-other-step' (state-of y)))) = None

using *assms*(2) $y S$ *raw-conflicting-noTrue-iff-toS* **by** *blast*

have *backtrack* (toS (rough-state-of S))

(toS (rough-state-of (do-other-step' (state-of y)))) \vee

resolve (toS (rough-state-of S))

(toS (rough-state-of (do-other-step' (state-of y)))) \vee

skip (toS (rough-state-of S))

(toS (rough-state-of (do-other-step' (state-of y))))

proof –

have $f1$: ($M, N, U, k, \text{Some } E$) = rough-state-of S

by (*simp add: M S y*)

then have $f2$: $\text{do-other-step } (M, N, U, k, \text{Some } E) \neq (M, N, U, k, \text{Some } E)$
by ($\text{metis assms}(3)$ $\text{rough-state-of-do-other-step' rough-state-of-inject}$)
have $\text{cdcl}_W\text{-restart-all-struct-inv}$ ($\text{toS } (M, N, U, k, \text{Some } E)$)
using $f1$ **by** simp
then have $\text{cdcl}_W\text{-o}$ ($\text{toS } (M, N, U, k, \text{Some } E)$) ($\text{toS } (\text{do-other-step } (M, N, U, k, \text{Some } E))$)
using $f2$ do-other-step **by** blast
then have $f3$: $\text{cdcl}_W\text{-o}$ ($\text{toS } (\text{rough-state-of } S)$)
($\text{toS } (\text{rough-state-of } (\text{do-other-step' } (\text{state-of } (M, N, U, k, \text{Some } E))))$)
using $f1$ **by** ($\text{simp add: rough-state-of-do-other-step'}$)
have $\neg \text{decide}$ ($\text{toS } (\text{rough-state-of } S)$)
($\text{toS } (\text{rough-state-of } (\text{do-other-step' } (\text{state-of } (M, N, U, k, \text{Some } E))))$)
using $f1$ **by** ($\text{metis (no-types) do-decide-step.simps}(2)$ do-decide-step-no)
then show $?thesis$
using $f3$ $\text{cdcl}_W\text{-o-rule-cases } y$ **by** blast
qed
then have bt : backtrack ($\text{toS } (\text{rough-state-of } S)$)
($\text{toS } (\text{rough-state-of } (\text{do-other-step' } (\text{state-of } y))))$
using conflict-y **by** ($\text{cases rough-state-of } S$) ($\text{auto elim!: resolveE skipE}$)
moreover
have no-dup ($\text{raw-trail } (\text{rough-state-of } S)$)
using $\text{rough-state-of[of } S]$ **unfolding** $\text{cdcl}_W\text{-restart-all-struct-inv-def}$ $\text{cdcl}_W\text{-restart-M-level-inv-def}$
by ($\text{cases } S$) ($\text{auto simp: comp-def}$)
have $\text{cdcl}_W\text{-restart-M-level-inv}$ ($\text{toS } (\text{rough-state-of } S)$) **and**
 $\text{cdcl}_W\text{-restart-M-level-inv}$ ($\text{toS } (\text{rough-state-of } (\text{do-other-step' } (\text{state-of } y))))$
using inv apply ($\text{simp add: cdcl}_W\text{-restart-all-struct-inv-def } S$)
using $\text{cdcl}_W\text{-restart-all-struct-inv-def}$ $\text{cdcl}_W\text{-restart-all-struct-inv-rough-state}$ **by** blast
then show $?case$
using $\text{backtrack-lvl-backtrack-decrease}[OF - bt]$
using S **unfolding** $\text{cdcl}_W\text{-restart-M-level-inv-def}$
by ($\text{simp add: comp-def count-decided-raw-trail-toS}$)
qed

lemma $\text{do-other-step-not-raw-conflicting-one-more-decide}$:

assumes $\text{raw-conflicting } (\text{rough-state-of } S) = \text{None}$ **and**
 $\text{do-other-step' } S \neq S$
shows $1 + \text{length } (\text{filter is-decided } (\text{raw-trail } (\text{rough-state-of } S)))$
 $= \text{length } (\text{filter is-decided } (\text{raw-trail } (\text{rough-state-of } (\text{do-other-step' } S))))$
proof ($\text{cases } S$, goal-cases)
case ($1\ y$) **note** $S = \text{this}(1)$ **and** $\text{inv} = \text{this}(2)$
obtain $M\ N\ U\ k$ **where** $y: y = (M, N, U, k, \text{None})$ **using** $\text{assms}(1)$ S inv **by** ($\text{cases } y$) auto
have M : $\text{rough-state-of } (\text{state-of } (M, N, U, k, \text{None})) = (M, N, U, k, \text{None})$
using $\text{inv } y$ **by** ($\text{auto simp add: state-of-inverse}$)
have $\text{state-of } (\text{do-decide-step } (M, N, U, k, \text{None})) \neq \text{state-of } (M, N, U, k, \text{None})$
using $\text{assms}(2)$ **unfolding** $\text{do-other-step'-def } y$ $\text{inv } S$ **by** ($\text{auto simp add: } M$)
then have $f4$: $\text{do-skip-step } (\text{rough-state-of } S) = \text{rough-state-of } S$
unfolding $S\ M\ y$ **by** ($\text{metis (full-types) do-skip-step.simps}(4)$)
have $f5$: $\text{do-resolve-step } (\text{rough-state-of } S) = \text{rough-state-of } S$
unfolding $S\ M\ y$ **by** ($\text{metis (no-types) do-resolve-step.simps}(4)$)
have $f6$: $\text{do-backtrack-step } (\text{rough-state-of } S) = \text{rough-state-of } S$
unfolding $S\ M\ y$ **by** ($\text{metis (no-types) do-backtrack-step.simps}(2)$)
have $\text{do-other-step } (\text{rough-state-of } S) \neq \text{rough-state-of } S$
using $\text{assms}(2)$ **unfolding** $S\ M\ y$ $\text{do-other-step'-def}$ **by** (metis (no-types))
then show $?case$
using $f6\ f5\ f4$ **by** ($\text{simp add: assms}(1)$ $\text{do-decide-step-not-raw-conflicting-one-more-decide}$
 $\text{do-other-step'-def}$)
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 *raw-conflicting-do-resolve-step-iff*[iff]:
raw-conflicting (*do-resolve-step* *S*) = *None* \longleftrightarrow *raw-conflicting* *S* = *None*
by (*cases S rule: do-resolve-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *raw-conflicting-do-skip-step-iff*[iff]:
raw-conflicting (*do-skip-step* *S*) = *None* \longleftrightarrow *raw-conflicting* *S* = *None*
by (*cases S rule: do-skip-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *raw-conflicting-do-decide-step-iff*[iff]:
raw-conflicting (*do-decide-step* *S*) = *None* \longleftrightarrow *raw-conflicting* *S* = *None*
by (*cases S rule: do-decide-step.cases*)
(auto simp add: Let-def split: option.splits)

lemma *raw-conflicting-do-backtrack-step-imp*[simp]:
do-backtrack-step *S* \neq *S* \implies *raw-conflicting* (*do-backtrack-step* *S*) = *None*
by (*cases S rule: do-backtrack-step.cases*)
(auto simp add: Let-def split: list.splits option.splits ann-lit.splits)

lemma *do-skip-step-eq-iff-raw-trail-eq*:
do-skip-step *S* = *S* \longleftrightarrow *raw-trail* (*do-skip-step* *S*) = *raw-trail* *S*
by (*cases S rule: do-skip-step.cases*) *auto*

lemma *do-decide-step-eq-iff-raw-trail-eq*:
do-decide-step *S* = *S* \longleftrightarrow *raw-trail* (*do-decide-step* *S*) = *raw-trail* *S*
by (*cases S rule: do-decide-step.cases*) *(auto split: option.split)*

lemma *do-backtrack-step-eq-iff-raw-trail-eq*:
assumes *no-dup* (*raw-trail* *S*)
shows *do-backtrack-step* *S* = *S* \longleftrightarrow *raw-trail* (*do-backtrack-step* *S*) = *raw-trail* *S*
using *assms* **apply** (*cases S rule: do-backtrack-step.cases*)
by (*auto split: option.split list.splits ann-lit.splits*
simp: comp-def
dest!: bt-cut-in-get-all-ann-decomposition)

lemma *do-resolve-step-eq-iff-raw-trail-eq*:
do-resolve-step *S* = *S* \longleftrightarrow *raw-trail* (*do-resolve-step* *S*) = *raw-trail* *S*
by (*cases S rule: do-resolve-step.cases*) *auto*

lemma *do-other-step-eq-iff-raw-trail-eq*:
assumes *no-dup* (*raw-trail* *S*)
shows *raw-trail* (*do-other-step* *S*) = *raw-trail* *S* \longleftrightarrow *do-other-step* *S* = *S*
using *assms*
by (*auto simp add: Let-def do-skip-step-eq-iff-raw-trail-eq[symmetric]*
do-decide-step-eq-iff-raw-trail-eq[symmetric] *do-backtrack-step-eq-iff-raw-trail-eq[symmetric]*
do-resolve-step-eq-iff-raw-trail-eq[symmetric])

lemma *do-full1-cp-step-do-other-step'-normal-form*[*dest!*]:
assumes *H*: *do-full1-cp-step* (*do-other-step'* *S*) = *S*

shows $do_other_step' S = S \wedge do_full1_cp_step S = S$
proof –
 let $?T = do_other_step' S$
 { **assume** $confl$: $raw_conflicting (rough_state_of ?T) \neq None$
 then have tr : $raw_trail (rough_state_of (do_full1_cp_step ?T)) = raw_trail (rough_state_of ?T)$
 using $do_full1_cp_step_raw_conflicting[of ?T]$ **by** $auto$
 have $raw_trail (rough_state_of (do_full1_cp_step (do_other_step' S))) = raw_trail (rough_state_of S)$
 using $arg_cong[OF H, of \lambda S. raw_trail (rough_state_of S)]$.
 then have $raw_trail (rough_state_of (do_other_step' S)) = raw_trail (rough_state_of S)$
 by ($auto simp add: do_full1_cp_step_raw_conflicting confl$)
 then have $do_other_step' S = S$
 using $assms confl$
 by ($simp add: do_other_step_eq_iff_raw_trail_eq do_other_step'_def$
 $do_full1_cp_step_raw_conflicting$
 $del: do_other_step.simps$)
 }
moreover {
 assume $eq[simp]$: $do_other_step' S = S$
 obtain c **where** c : $raw_trail (rough_state_of (do_full1_cp_step S)) = c @ raw_trail (rough_state_of S)$
 using $do_full1_cp_step_neq_raw_trail_increase$ **by** $auto$

 moreover have $raw_trail (rough_state_of (do_full1_cp_step S)) = raw_trail (rough_state_of S)$
 using $arg_cong[OF H, of \lambda S. raw_trail (rough_state_of S)]$ **by** $simp$
 finally have $c = []$ **by** $blast$
 then have $do_full1_cp_step S = S$ **using** $assms$ **by** $auto$
 }
moreover {
 assume $confl$: $raw_conflicting (rough_state_of ?T) = None$ **and** neq : $do_other_step' S \neq S$
 obtain c **where**
 c : $raw_trail (rough_state_of (do_full1_cp_step ?T)) = c @ raw_trail (rough_state_of ?T)$ **and**
 nm : $\forall m \in set\ c. \neg is_decided\ m$
 using $do_full1_cp_step_neq_raw_trail_increase$ **by** $auto$
 have $length (filter\ is_decided (raw_trail (rough_state_of (do_full1_cp_step ?T))))$
 $= length (filter\ is_decided (raw_trail (rough_state_of ?T)))$ **using** nm **unfolding** c **by** $force$
 moreover have $length (filter\ is_decided (raw_trail (rough_state_of S)))$
 $\neq length (filter\ is_decided (raw_trail (rough_state_of ?T)))$
 using $do_other_step_not_raw_conflicting_one_more_decide[OF - neq]$
 $do_other_step_not_raw_conflicting_one_more_decide_bt[of S, OF - confl neq]$
 by $linarith$
 finally have $False$ **unfolding** H **by** $blast$
 }
ultimately show $?thesis$ **by** $blast$
qed

lemma $do_cdcl_W_stgy_step_no$:
assumes S : $do_cdcl_W_stgy_step S = S$
shows $no_step\ cdcl_W_stgy (toS (rough_state_of S))$

proof –
 {
 fix S'
 assume $full1\ cdcl_W_cp (toS (rough_state_of S)) S'$
 then have $False$
 using $do_full1_cp_step_full[of S]$ **unfolding** $full_def\ S\ rtranclp_unfold\ full1_def$
 by ($smt\ assms\ do_cdcl_W_stgy_step_def\ tranclpD$)
 }
qed


```

moreover {
  fix  $S' S''$ 
  assume  $cdcl_W-o$  ( $toS$  ( $rough-state-of$   $S$ ))  $S'$  and
   $no-step$   $propagate$  ( $toS$  ( $rough-state-of$   $S$ )) and
   $no-step$   $conflict$  ( $toS$  ( $rough-state-of$   $S$ )) and
   $full$   $cdcl_W-cp$   $S' S''$ 
  then have  $False$ 
  using  $assms$  unfolding  $do-cdcl_W-stgy-step-def$ 
  by ( $smt$   $cdcl_W-restart-all-struct-inv-rough-state$   $do-full1-cp-step-do-other-step'-normal-form$ 
     $do-other-step-no$   $rough-state-of-do-other-step'$ )
}
ultimately show  $?thesis$  using  $assms$  by ( $force$   $simp$ :  $cdcl_W-cp.simps$   $cdcl_W-stgy.simps$ )
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  $cdcl_W-cp-is-rtrancp-cdcl_W-restart$ :  $cdcl_W-cp S T \implies cdcl_W-restart^{**} S T$ 
apply ( $induction$   $rule$ :  $cdcl_W-cp.induct$ )
using  $conflict$  apply  $blast$ 
using  $propagate$  by  $blast$ 

```

```

lemma  $rtrancp-cdcl_W-cp-is-rtrancp-cdcl_W-restart$ :  $cdcl_W-cp^{**} S T \implies cdcl_W-restart^{**} S T$ 
apply ( $induction$   $rule$ :  $rtrancp.induct$ )
apply  $simp$ 
by ( $fastforce$   $dest!$ :  $cdcl_W-cp-is-rtrancp-cdcl_W-restart$ )

```

```

lemma  $cdcl_W-stgy-is-rtrancp-cdcl_W-restart$ :
   $cdcl_W-stgy S T \implies cdcl_W-restart^{**} S T$ 
apply ( $induction$   $rule$ :  $cdcl_W-stgy.induct$ )
using  $cdcl_W-stgy.conflict'$   $rtrancp-cdcl_W-stgy-rtrancp-cdcl_W-restart$  apply  $blast$ 
unfolding  $full-def$  by ( $fastforce$   $dest!$ :  $other$   $rtrancp-cdcl_W-cp-is-rtrancp-cdcl_W-restart$ )

```

```

lemma  $cdcl_W-stgy-init-raw-init-clss$ :
   $cdcl_W-stgy S T \implies cdcl_W-restart-M-level-inv S \implies raw-init-clss S = raw-init-clss T$ 
using  $cdcl_W-stgy-no-more-init-clss$  by  $blast$ 

```

```

lemma  $clauses-toS-rough-state-of-do-cdcl_W-stgy-step[simp]$ :
   $raw-init-clss$  ( $toS$  ( $rough-state-of$  ( $do-cdcl_W-stgy-step$  ( $state-of$  ( $rough-state-from-init-state-of$   $S$ ))))))
  =  $raw-init-clss$  ( $toS$  ( $rough-state-from-init-state-of$   $S$ )) (is - =  $raw-init-clss$  ( $toS$   $?S$ ))
apply ( $cases$   $do-cdcl_W-stgy-step$  ( $state-of$   $?S$ ) =  $state-of$   $?S$ )
apply  $simp$ 
by ( $metis$   $cdcl_W-restart-all-struct-inv-def$   $cdcl_W-restart-all-struct-inv-rough-state$   $cdcl_W-stgy-no-more-init-clss$ 
   $do-cdcl_W-stgy-step$   $toS-rough-state-of-state-of-rough-state-from-init-state-of$ )

```

```

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  $cdcl_W-stgy^{**}$  ( $S0-cdcl_W-restart$  ( $raw-init-clss$  ( $toS$  ( $rough-state-from-init-state-of$   $S$ ))))
    ( $toS$  ( $rough-state-from-init-state-of$   $S$ ))
  using  $rough-state-from-init-state-of[of S]$  by  $auto$ 

```

moreover have $cdcl_W\text{-stgy}^{**}$
 $(toS \text{ (rough-state-from-init-state-of } S))$
 $(toS \text{ (rough-state-of (do-cdcl}_W\text{-stgy-step}$
 $\text{ (state-of (rough-state-from-init-state-of } S))))))$
using $do\text{-cdcl}_W\text{-stgy-step[of state-of ?S]$
by $(cases \text{ do-cdcl}_W\text{-stgy-step (state-of ?S) = state-of ?S) auto$
ultimately show $?thesis$
unfolding $do\text{-cdcl}_W\text{-stgy-step'-def id-of-I-to-def}$
by $(auto intro!: \text{state-from-init-state-of-inverse})$
qed

All rules together function $do\text{-all-cdcl}_W\text{-stgy}$ **where**

$do\text{-all-cdcl}_W\text{-stgy } S =$

$(let \ T = do\text{-cdcl}_W\text{-stgy-step' } S \text{ in}$
 $if \ T = S \text{ then } S \text{ else } do\text{-all-cdcl}_W\text{-stgy } T)$

by fast+

termination

proof $(relation \ \{(T, S).$

$(cdcl_W\text{-restart-measure (toS (rough-state-from-init-state-of } T)),$
 $cdcl_W\text{-restart-measure (toS (rough-state-from-init-state-of } S)))$
 $\in \text{lexn less-than } 3\}$, $goal\text{-cases}$)

case 1

show $?case$ **by** $(rule \text{ wf-if-measure-f}) (auto intro!: \text{wf-lexn wf-less})$

next

case $(2 \ S \ T)$ **note** $T = this(1)$ **and** $ST = this(2)$

let $?S = \text{rough-state-from-init-state-of } S$

have $S: cdcl_W\text{-stgy}^{**} (S0\text{-cdcl}_W\text{-restart (raw-init-clss (toS ?S))) (toS ?S))$

using $\text{rough-state-from-init-state-of[of } S]$ **by** $auto$

moreover have $cdcl_W\text{-stgy (toS (rough-state-from-init-state-of } S))$

$(toS \text{ (rough-state-from-init-state-of } T))$

proof –

have $\bigwedge c. \text{rough-state-of (state-of (rough-state-from-init-state-of } c)) =$
 $\text{rough-state-from-init-state-of } c$

using $\text{rough-state-from-init-state-of state-of-inverse}$ **by** fastforce

then have $\text{diff: } do\text{-cdcl}_W\text{-stgy-step (state-of (rough-state-from-init-state-of } S))$
 $\neq \text{state-of (rough-state-from-init-state-of } S)$

using $ST \ T$ **by** $(metis \text{ (no-types) id-of-I-to-def rough-state-from-init-state-of-inject}$
 $\text{rough-state-from-init-state-of-do-cdcl}_W\text{-stgy-step'})$

have $\text{rough-state-of (do-cdcl}_W\text{-stgy-step (state-of (rough-state-from-init-state-of } S)))$
 $= \text{rough-state-from-init-state-of (do-cdcl}_W\text{-stgy-step' } S)$

by $(simp \text{ add: id-of-I-to-def rough-state-from-init-state-of-do-cdcl}_W\text{-stgy-step'})$

then show $?thesis$

using $do\text{-cdcl}_W\text{-stgy-step } T \text{ diff}$ **unfolding** $\text{id-of-I-to-def do-cdcl}_W\text{-stgy-step}$ **by** fastforce

qed

moreover

have $cdcl_W\text{-restart-all-struct-inv (toS (rough-state-from-init-state-of } S))$

using $\text{rough-state-from-init-state-of[of } S]$ **by** $auto$

then have $cdcl_W\text{-restart-all-struct-inv (S0-cdcl}_W\text{-restart (raw-init-clss (toS (rough-state-from-init-state-of } S))))$

by $(cases \text{ rough-state-from-init-state-of } S)$

$(auto \text{ simp add: cdcl}_W\text{-restart-all-struct-inv-def distinct-cdcl}_W\text{-restart-state-def})$

ultimately show $?case$

using $\text{trancp-cdcl}_W\text{-stgy-S0-decreasing}$

by $(auto intro!: \text{cdcl}_W\text{-stgy-step-decreasing[of - - S0-cdcl}_W\text{-restart (raw-init-clss (toS ?S))])$
 $\text{simp del: cdcl}_W\text{-restart-measure.simps})$

qed

```

thm do-all-cdclW-stgy.induct
lemma do-all-cdclW-stgy.induct:
  ( $\bigwedge S. (do-cdcl_W-stgy-step' S \neq S \implies P (do-cdcl_W-stgy-step' S)) \implies P S) \implies P a0$ )
using do-all-cdclW-stgy.induct by metis

lemma no-step-cdclW-stgy-cdclW-restart-all:
  fixes S :: 'a cdclW-restart-state-inv-from-init-state
  shows 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 \neq S)
proof -
  fix Sa :: 'a cdclW-restart-state-inv-from-init-state
  assume a1:  $\neg do-cdcl_W-stgy-step' Sa \neq Sa$ 
  { fix pp
    have (if True then Sa else do-all-cdclW-stgy Sa) = do-all-cdclW-stgy Sa
      using a1 by auto
    then have  $\neg cdcl_W-stgy (toS (rough-state-from-init-state-of (do-all-cdcl_W-stgy Sa)))$  pp
      using a1 by (metis (no-types) do-cdclW-stgy-step-no id-of-I-to-def
        rough-state-from-init-state-of-do-cdclW-stgy-step' rough-state-of-inverse) }
    then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa)))
      by fastforce
  }
next
  fix Sa :: 'a cdclW-restart-state-inv-from-init-state
  assume a1: do-cdclW-stgy-step' Sa \neq Sa
     $\implies no-step cdcl_W-stgy (toS (rough-state-from-init-state-of$ 
       $(do-all-cdcl_W-stgy (do-cdcl_W-stgy-step' Sa))))$ 
  assume a2: do-cdclW-stgy-step' Sa \neq Sa
  have do-all-cdclW-stgy Sa = do-all-cdclW-stgy (do-cdclW-stgy-step' Sa)
    by (metis (full-types) do-all-cdclW-stgy.simps)
  then show no-step cdclW-stgy (toS (rough-state-from-init-state-of (do-all-cdclW-stgy Sa)))
    using a2 a1 by presburger
qed

lemma do-all-cdclW-stgy-is-rtrancpl-cdclW-stgy:
  cdclW-stgy** (toS (rough-state-from-init-state-of S))
  (toS (rough-state-from-init-state-of (do-all-cdclW-stgy S)))
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  $\longrightarrow$ 
        rough-state-from-init-state-of (do-cdclW-stgy-step' S)
         $= rough-state-of (state-of (rough-state-from-init-state-of S))$ 
        using rough-state-from-init-state-of-do-cdclW-stgy-step'
        by (simp add: id-of-I-to-def rough-state-from-init-state-of-do-cdclW-stgy-step')
      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)))
         $= cdcl_W-stgy (toS (rough-state-of (id-of-I-to S)))$ 
        (toS (rough-state-of (do-cdclW-stgy-step (id-of-I-to S))))

```

```

    using rough-state-from-init-state-of-do-cdclW-stgy-step'
    toS-rough-state-of-state-of-rough-state-from-init-state-of
    by (simp add: id-of-I-to-def rough-state-from-init-state-of-do-cdclW-stgy-step')
  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-restart-all-struct-inv (toS ([], map remdups N, [], 0, None))
    unfolding cdclW-restart-all-struct-inv-def distinct-cdclW-restart-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-rtranclp-cdclW-stgy[of
      state-from-init-state-of ([], map remdups N, [], 0, None)] inv
      no-step-cdclW-stgy-cdclW-restart-all
    apply (auto simp del: do-all-cdclW-stgy.simps simp: state-from-init-state-of-inverse
      r[symmetric] comp-def)[]
    using do-all-cdclW-stgy-is-rtranclp-cdclW-stgy[of
      state-from-init-state-of ([], map remdups N, [], 0, None)] inv
      no-step-cdclW-stgy-cdclW-restart-all
    by (force simp: state-from-init-state-of-inverse r[symmetric] comp-def)
  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-restart-all-struct-inv (toS S)
      by (metis (no-types) cdclW-restart-all-struct-inv-rough-state
        toS-rough-state-of-state-of-rough-state-from-init-state-of)
    then have cons: consistent-interp (lits-of-l M')
      unfolding cdclW-restart-all-struct-inv-def cdclW-restart-M-level-inv-def S[symmetric] by auto
  moreover
    have raw-init-clss (toS ([], ?N, [], 0, None)) = raw-init-clss (toS S)
      apply (rule rtranclp-cdclW-stgy-no-more-init-clss)
      using 1 unfolding full-def by (auto simp add: rtranclp-cdclW-stgy-rtranclp-cdclW-restart)
    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-Abstract-Clause-Representation
imports Main Partial-Clausal-Logic
begin

```

```

type-synonym 'v clause = 'v literal multiset
type-synonym 'v clauses = 'v clause multiset

```

7.1.6 Abstract Clause Representation

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

We assume the following:

- there is an equivalent to adding and removing a literal and to taking the union of clauses.

```

locale raw-cls =
  fixes
    mset-cl :: 'cls  $\Rightarrow$  'v clause
begin
end

```

Instantiation of the previous locale, in an unnamed context to avoid polluting with simp rules

```

context
begin
  interpretation list-cl: raw-cls mset
    by unfold-locales

  interpretation cls-cl: raw-cls id
    by unfold-locales
end

```

Over the abstract clauses, we have the following properties:

- We can insert a clause
- We can take the union (used only in proofs for the definition of *clauses*)
- there is an operator indicating whether the abstract clause is contained or not
- if a concrete clause is contained the abstract clauses, then there is an abstract clause

```

locale raw-clss =
  raw-cls mset-cl

```

```

for
   $mset-cls :: 'cls \Rightarrow 'v \text{ clause} +$ 
fixes
   $mset-clss :: 'clss \Rightarrow 'v \text{ clauses}$  and
   $in-clss :: 'cls \Rightarrow 'clss \Rightarrow \text{bool}$  and
   $insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ 
assumes
   $insert-clss[simp]: mset-clss (insert-clss L C) = mset-clss C + \{\#mset-cl L\# \}$  and
   $in-clss-mset-clss[dest]: in-clss a C \implies mset-cls a \in \# mset-clss C$  and
   $in-mset-clss-exists-preimage: b \in \# mset-clss C \implies \exists b'. in-clss b' C \wedge mset-cls b' = b$ 
begin

end

experiment
begin
  fun remove-first where
    remove-first - [] = [] |
    remove-first C (C' # L) = (if mset C = mset C' then L else C' # remove-first C L)

  lemma mset-map-mset-remove-first:
     $mset (map \text{ mset } (remove-first a C)) = remove1-mset (mset a) (mset (map \text{ mset } C))$ 
    by (induction C) (auto simp: ac-simps remove1-mset-single-add)

  interpretation clss-clss: raw-clss id
     $id \text{ op } \in \# \lambda L. C. C + \{\#L\# \}$ 
    by unfold-locales (auto simp: ac-simps)

  interpretation list-clss: raw-clss mset
     $\lambda L. mset (map \text{ mset } L) \lambda L. C. L \in \text{set } C \text{ op } \#$ 
    by unfold-locales (auto simp: ac-simps union-mset-list mset-map-mset-remove-first ex-mset)
end

end
theory CDCL-W-Abstract-State
imports CDCL-Abstract-Clause-Representation List-More CDCL-W-Level Wellfounded-More
  CDCL-WNOT CDCL-Abstract-Clause-Representation

begin

```

7.2 Weidenbach's CDCL with Abstract Clause Representation

We first instantiate the locale of Weidenbach's locale. Then we define another abstract state: the goal of this state is to be used for implementations. We add more assumptions on the function about the state. For example *cons-trail* is restricted to undefined literals.

7.2.1 Instantiation of the Multiset Version

```

type-synonym  $'v \text{ cdcl}_W\text{-restart-mset} = ('v, 'v \text{ clause}) \text{ ann-lit list } \times$ 
   $'v \text{ clauses } \times$ 
   $'v \text{ clauses } \times$ 
   $\text{nat } \times 'v \text{ clause option}$ 

```

We use definition, otherwise we could not use the simplification theorems we have already shown.

definition *trail* :: 'v cdcl_W-restart-mset \Rightarrow ('v, 'v clause) ann-lit list **where**
trail $\equiv \lambda(M, -). M$

definition *init-clss* :: 'v cdcl_W-restart-mset \Rightarrow 'v clauses **where**
init-clss $\equiv \lambda(-, N, -). N$

definition *learned-clss* :: 'v cdcl_W-restart-mset \Rightarrow 'v clauses **where**
learned-clss $\equiv \lambda(-, -, U, -). U$

definition *backtrack-lvl* :: 'v cdcl_W-restart-mset \Rightarrow nat **where**
backtrack-lvl $\equiv \lambda(-, -, -, k, -). k$

definition *conflicting* :: 'v cdcl_W-restart-mset \Rightarrow 'v clause option **where**
conflicting $\equiv \lambda(-, -, -, -, C). C$

definition *cons-trail* :: ('v, 'v clause) ann-lit \Rightarrow 'v cdcl_W-restart-mset \Rightarrow 'v cdcl_W-restart-mset **where**
cons-trail $\equiv \lambda L (M, R). (L \# M, R)$

definition *tl-trail* **where**
tl-trail $\equiv \lambda(M, R). (tl\ M, R)$

definition *add-learned-cls* **where**
add-learned-cls $\equiv \lambda C (M, N, U, R). (M, N, \{\#C\# \} + U, R)$

definition *remove-cls* **where**
remove-cls $\equiv \lambda C (M, N, U, R). (M, removeAll-mset\ C\ N, removeAll-mset\ C\ U, R)$

definition *update-backtrack-lvl* **where**
update-backtrack-lvl $\equiv \lambda k (M, N, U, -, D). (M, N, U, k, D)$

definition *update-conflicting* **where**
update-conflicting $\equiv \lambda D (M, N, U, k, -). (M, N, U, k, D)$

definition *init-state* **where**
init-state $\equiv \lambda N. ([], N, \{\#\}, 0, None)$

lemmas *cdcl_W-restart-mset-state* = *trail-def cons-trail-def tl-trail-def add-learned-cls-def*
remove-cls-def update-backtrack-lvl-def update-conflicting-def init-clss-def learned-clss-def
backtrack-lvl-def conflicting-def init-state-def

interpretation *cdcl_W-restart-mset: state_W-ops* **where**

trail = *trail* **and**

init-clss = *init-clss* **and**

learned-clss = *learned-clss* **and**

backtrack-lvl = *backtrack-lvl* **and**

conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**

tl-trail = *tl-trail* **and**

add-learned-cls = *add-learned-cls* **and**

remove-cls = *remove-cls* **and**

update-backtrack-lvl = *update-backtrack-lvl* **and**

update-conflicting = *update-conflicting* **and**

init-state = *init-state*

.

interpretation *cdcl_W-restart-mset*: *state_W* **where**

trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**
tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*
by *unfold-locales* (*auto simp*: *cdcl_W-restart-mset-state*)

interpretation *cdcl_W-restart-mset*: *conflict-driven-clause-learning_W* **where**

trail = *trail* **and**
init-clss = *init-clss* **and**
learned-clss = *learned-clss* **and**
backtrack-lvl = *backtrack-lvl* **and**
conflicting = *conflicting* **and**

cons-trail = *cons-trail* **and**
tl-trail = *tl-trail* **and**
add-learned-cls = *add-learned-cls* **and**
remove-cls = *remove-cls* **and**
update-backtrack-lvl = *update-backtrack-lvl* **and**
update-conflicting = *update-conflicting* **and**
init-state = *init-state*
by *unfold-locales auto*

lemma *cdcl_W-restart-mset-state-eq-eq*: *cdcl_W-restart-mset.state-eq* = (*op* =)

apply (*intro ext*)
unfolding *cdcl_W-restart-mset.state-eq-def*
by (*auto simp*: *cdcl_W-restart-mset-state*)

notation *cdcl_W-restart-mset.state-eq* (**infix** \sim_m 49)

7.2.2 Abstract Relation and Relation Theorems

This locales makes the lifting from the relation defined with multiset *R* and the version with an abstract state *R-abs*. We are lifting many different relations (each rule and the the strategy).

locale *relation-implied-relation-abs* =

fixes

R :: '*v cdcl_W-restart-mset* \Rightarrow '*v cdcl_W-restart-mset* \Rightarrow *bool* **and**

R-abs :: '*st* \Rightarrow '*st* \Rightarrow *bool* **and**

state :: '*st* \Rightarrow '*v cdcl_W-restart-mset* **and**

inv :: '*v cdcl_W-restart-mset* \Rightarrow *bool*

assumes

relation-compatible-state:

inv (*state S*) \Longrightarrow *R-abs S T* \Longrightarrow *R (state S) (state T)* **and**

relation-compatible-abs:

$\bigwedge S S' T. \text{inv } S \Longrightarrow S \sim_m \text{state } S' \Longrightarrow R S T \Longrightarrow \exists U. R\text{-abs } S' U \wedge T \sim_m \text{state } U$ **and**

relation-invariant:

$\bigwedge S T. R S T \implies \text{inv } S \implies \text{inv } T$ **and**
relation-abs-right-compatible:
 $\bigwedge S T U. \text{inv } (\text{state } S) \implies R\text{-abs } S T \implies \text{state } T \sim_m \text{state } U \implies R\text{-abs } S U$

begin

lemma *relation-compatible-eq:*

assumes

inv: $\text{inv } (\text{state } S)$ **and**

abs: $R\text{-abs } S T$ **and**

SS': $\text{state } S \sim_m \text{state } S'$ **and**

TT': $\text{state } T \sim_m \text{state } T'$

shows $R\text{-abs } S' T'$

proof –

have $R (\text{state } S) (\text{state } T)$

using *relation-compatible-state inv abs* **by** *blast*

then obtain U **where** $S'U: R\text{-abs } S' U$ **and** $TU: \text{state } T \sim_m \text{state } U$

using *relation-compatible-abs[OF inv SS']* **by** *blast*

then show *?thesis*

using *relation-abs-right-compatible[OF - S'U, of T'] TT' inv SS'[unfolded cdcl_W-restart-mset-state-eq-eq]*
 cdcl_W-restart-mset.state-eq-trans[of state T' state T state U]
 by (*auto simp add: cdcl_W-restart-mset.state-eq-sym*)

qed

lemma *rtranclp-relation-invariant:*

$R^{++} S T \implies \text{inv } S \implies \text{inv } T$

by (*induction rule: trancplp-induct*) (*auto simp: relation-invariant*)

lemma *rtranclp-abs-rtranclp:*

$R\text{-abs}^{**} S T \implies \text{inv } (\text{state } S) \implies R^{**} (\text{state } S) (\text{state } T)$

apply (*induction rule: rtranclp-induct*)

apply *simp*

by (*metis relation-compatible-state rtranclp.simps rtranclpD rtranclp-relation-invariant*)

lemma *trancplp-relation-trancplp-relation-abs-compatible:*

fixes $S :: 'st$

assumes

$R: R^{++} (\text{state } S) T$ **and**

inv: $\text{inv } (\text{state } S)$

shows $\exists U. R\text{-abs}^{++} S U \wedge T \sim_m \text{state } U$

using R

proof (*induction rule: trancplp-induct*)

case (*base T*)

then show *?case*

using *relation-compatible-abs[of state S S T] inv* **by** *auto*

next

case (*step T U*) **note** $st = \text{this}(1)$ **and** $R = \text{this}(2)$ **and** $IH = \text{this}(3)$

obtain V **where**

$SV: R\text{-abs}^{++} S V$ **and** $TV: T \sim_m \text{state } V$

using IH **by** *auto*

then obtain W **where**

$VW: R\text{-abs } V W$ **and** $UW: U \sim_m \text{state } W$

using *relation-compatible-abs[OF - TV R] inv rtranclp-relation-invariant[OF st]* **by** *blast*

have $R\text{-abs}^{++} S W$

using $SV VW$ **by** *auto*

then show *?case* **using** UW **by** *blast*

qed

lemma *rtrancpl-relation-rtrancpl-relation-abs-compatible*:
fixes $S :: 'st$
assumes
 $R: R^{**} (state\ S)\ T$ **and**
 $inv: inv\ (state\ S)$
shows $\exists U. R-abs^{**}\ S\ U \wedge T \sim_m state\ U$
using $R\ inv$ **by** (*auto simp: rtrancpl-unfold dest: trancpl-relation-trancpl-relation-abs-compatible*)

lemma *no-step-iff*:
 $inv\ (state\ S) \implies no-step\ R\ (state\ S) \longleftrightarrow no-step\ R-abs\ S$
using *relation-compatible-state relation-compatible-abs cdcl_W-restart-mset.state-eq-ref*
by *blast*

lemma *trancpl-relation-compatible-eq-and-inv*:
assumes
 $inv: inv\ (state\ S)$ **and**
 $st: R-abs^{++}\ S\ T$ **and**
 $SS': state\ S \sim_m state\ S'$ **and**
 $TU: state\ T \sim_m state\ U$
shows $R-abs^{++}\ S'\ U \wedge inv\ (state\ U)$
using $st\ TU$
proof (*induction arbitrary: U rule: trancpl-induct*)
case (*base T*)
moreover then have $inv\ (state\ U)$
by (*metis (full-types) cdcl_W-restart-mset-state-eq-eq inv relation-compatible-state relation-invariant*)
ultimately show *?case*
using *relation-compatible-eq[of S T S' U] SS' inv*
by (*auto simp: trancpl.r-into-trancpl*)
next
case (*step T T'*) **note** $st = this(1)$ **and** $R = this(2)$ **and** $IH = this(3)$ **and** $TU = this(4)$
have $R-abs^{++}\ S'\ T$ **and** $invT: inv\ (state\ T)$ **using** $IH[of\ T]$ **by** *auto*
moreover have $R-abs\ T\ U$
using *relation-compatible-eq[of T T' T U] R TU inv rtrancpl-relation-invariant invT* **by** *simp*
moreover have $inv\ (state\ U)$
using *calculation(3) invT relation-compatible-state relation-invariant* **by** *blast*
ultimately show *?case* **by** *auto*
qed

lemma
assumes
 $inv: inv\ (state\ S)$ **and**
 $st: R-abs^{++}\ S\ T$ **and**
 $SS': state\ S \sim_m state\ S'$ **and**
 $TU: state\ T \sim_m state\ U$
shows
 $trancpl-relation-compatible-eq: R-abs^{++}\ S'\ U$ **and**
 $trancpl-relation-abs-invariant: inv\ (state\ U)$
using *trancpl-relation-compatible-eq-and-inv[OF assms]* **by** *blast+*

lemma *trancpl-abs-trancpl*: $R-abs^{++}\ S\ T \implies inv\ (state\ S) \implies R^{++}\ (state\ S)\ (state\ T)$
apply (*induction rule: trancpl-induct*)
apply (*auto simp add: relation-compatible-state*)[]
apply *clarsimp*
apply (*erule trancpl.trancpl-into-trancpl*)

using *relation-compatible-state tranclp-relation-abs-invariant* **by** *blast*

lemma *full1-iff*:

assumes *inv*: *inv* (state *S*)

shows *full1* *R* (state *S*) (state *T*) \longleftrightarrow *full1* *R-abs* *S* *T* (**is** $?R \longleftrightarrow ?R\text{-abs}$)

proof

assume $?R$

then have *st*: R^{++} (state *S*) (state *T*) **and** *ns*: *no-step* *R* (state *T*) **unfolding** *full1-def* **by** *auto*

have *invT*: *inv* (state *T*)

using *inv* *rtranclp-relation-invariant* *st* **by** *blast*

then have *R-abs*⁺⁺ *S* *T*

using *tranclp-relation-tranclp-relation-abs-compatible*[*OF st*] *inv*

tranclp-relation-compatible-eq[*of S - S T*] *cdcl_W-restart-mset.state-eq-sym* **by** *blast*

moreover have *no-step* *R-abs* *T*

using *ns* *inv* *no-step-iff* *invT* **by** *blast*

ultimately show $?R\text{-abs}$

unfolding *full1-def* **by** *blast*

next

assume $?R\text{-abs}$

then have *st*: $R\text{-abs}^{++}$ *S* *T* **and** *ns*: *no-step* *R-abs* *T* **unfolding** *full1-def* **by** *auto*

have R^{++} (state *S*) (state *T*)

using *st* *tranclp-abs-tranclp* *inv* **by** *blast*

moreover

have *invT*: *inv* (state *T*)

using *inv* *tranclp-relation-abs-invariant* *st* **by** *blast*

then have *no-step* *R* (state *T*)

using *ns* *inv* *no-step-iff* **by** *blast*

ultimately show $?R$

unfolding *full1-def* **by** *blast*

qed

lemma *full1-iff-compatible*:

assumes *inv*: *inv* (state *S*) **and** *SS'*: $S' \sim_m$ state *S* **and** *TT'*: $T' \sim_m$ state *T*

shows *full1* *R* *S'* *T'* \longleftrightarrow *full1* *R-abs* *S* *T* (**is** $?R \longleftrightarrow ?R\text{-abs}$)

using *full1-iff* *assms* **unfolding** *cdcl_W-restart-mset-state-eq-eq* **by** *simp*

lemma *full-if-full-abs*:

assumes *inv* (state *S*) **and** *full* *R-abs* *S* *T*

shows *full* *R* (state *S*) (state *T*)

using *assms* *full1-iff* *cdcl_W-restart-mset-state-eq-eq* *relation-compatible-abs*

unfolding *full-unfold* **by** *blast*

The converse does *not* hold, since we cannot prove that $S = T$ given *state* *S* = *state* *S*.

lemma *full-abs-if-full*:

assumes *inv* (state *S*) **and** *full* *R* (state *S*) (state *T*)

shows *full* *R-abs* *S* *T* \vee (*state* *S* \sim_m state *T* \wedge *no-step* *R* (state *S*))

using *assms* *full1-iff* *relation-compatible-abs* **unfolding** *full-unfold* **by** *auto*

lemma *full-exists-full-abs*:

assumes *inv*: *inv* (state *S*) **and** *full*: *full* *R* (state *S*) *T*

obtains *U* **where** *full* *R-abs* *S* *U* **and** $T \sim_m$ state *U*

proof –

consider

(0) *state* *S* = *T* **and** *no-step* *R* (state *S*) |

(full1) *full1* *R* (state *S*) *T*

using *full* **unfolding** *full-unfold* *cdcl_W-restart-mset-state-eq-eq* **by** *fast*

```

then show ?thesis
proof cases
  case 0
  then show ?thesis using that[of S] unfolding full-def
    using cdclW-restart-mset.state-eq-ref inv relation-compatible-state rtrncpl.rtrncpl-refl by blast
next
  case full1
  then obtain U where
    R-abs++ S U and T ~m state U
    using trncpl-relation-trncpl-relation-abs-compatible inv unfolding full1-def
    by blast
  then show ?thesis
    using full1 that[of U] full1-iff[OF inv] full1-is-full full-def
    unfolding cdclW-restart-mset-state-eq-eq by blast
qed
qed

lemma full1-exists-full1-abs:
  assumes inv: inv (state S) and full1: full1 R (state S) T
  obtains U where full1 R-abs S U and T ~m state U
proof -
  obtain U where
    R-abs++ S U and T ~m state U
    using trncpl-relation-trncpl-relation-abs-compatible inv full1 unfolding full1-def
    by blast
  then show ?thesis
    using full1 that[of U] full1-iff[OF inv] unfolding cdclW-restart-mset-state-eq-eq by blast
qed

lemma full1-right-compatible:
  assumes inv (state S) and
    full1: full1 R-abs S T and TV: state T ~m state V
  shows full1 R-abs S V
  by (metis (full-types) TV assms(1) cdclW-restart-mset-state-eq-eq full1 full1-iff)

lemma full-right-compatible:
  assumes inv: inv (state S) and
    full-ST: full R-abs S T and TU: state T ~m state U
  shows full R-abs S U ∨ (S = T ∧ no-step R-abs S)
proof -
  consider
    (0) S = T and no-step R-abs T |
    (full1) full1 R-abs S T
  using full-ST unfolding full-unfold by blast
then show ?thesis
proof cases
  case full1
  then show ?thesis
    using full1-right-compatible[OF inv, of T U] TU full-unfold by blast
next
  case 0
  then show ?thesis by fast
qed
qed

end

```

```

locale relation-relation-abs =
  fixes
     $R :: 'v \text{ cdcl}_W\text{-restart-mset} \Rightarrow 'v \text{ cdcl}_W\text{-restart-mset} \Rightarrow \text{bool}$  and
     $R\text{-abs} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$  and
     $\text{state} :: 'st \Rightarrow 'v \text{ cdcl}_W\text{-restart-mset}$  and
     $\text{inv} :: 'v \text{ cdcl}_W\text{-restart-mset} \Rightarrow \text{bool}$ 
  assumes
    relation-compatible-state:
       $\text{inv} (\text{state } S) \Longrightarrow R (\text{state } S) (\text{state } T) \longleftrightarrow R\text{-abs } S \ T$  and
    relation-compatible-abs:
       $\bigwedge S \ S' \ T. \text{inv } S \Longrightarrow S \sim_m \text{state } S' \Longrightarrow R \ S \ T \Longrightarrow \exists U. R\text{-abs } S' \ U \wedge T \sim_m \text{state } U$  and
    relation-invariant:
       $\bigwedge S \ T. R \ S \ T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$ 
begin

lemma relation-compatible-eq:
   $\text{inv} (\text{state } S) \Longrightarrow R\text{-abs } S \ T \Longrightarrow \text{state } S \sim_m \text{state } S' \Longrightarrow \text{state } T \sim_m \text{state } T' \Longrightarrow R\text{-abs } S' \ T'$ 
  by (simp add: cdclW-restart-mset-state-eq-eq relation-compatible-state[symmetric])

lemma relation-right-compatible:
   $\text{inv} (\text{state } S) \Longrightarrow R\text{-abs } S \ T \Longrightarrow \text{state } T \sim_m \text{state } U \Longrightarrow R\text{-abs } S \ U$ 
  by (simp add: cdclW-restart-mset-state-eq-eq relation-compatible-state[symmetric])

sublocale relation-implied-relation-abs
  apply unfold-locales
  using relation-compatible-eq relation-compatible-state relation-compatible-abs relation-invariant
  relation-right-compatible by blast+

end

```

7.2.3 The State

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

```

locale abs-stateW-ops =
  raw-clss mset-cls
  mset-clss in-clss insert-clss
  +
  raw-cls mset-ccls
for
  — Clause
   $\text{mset-cls} :: 'cls \Rightarrow 'v \text{ clause}$  and

  — Multiset of Clauses
   $\text{mset-clss} :: 'clss \Rightarrow 'v \text{ clauses}$  and
   $\text{in-clss} :: 'cls \Rightarrow 'clss \Rightarrow \text{bool}$  and
   $\text{insert-clss} :: 'cls \Rightarrow 'clss \Rightarrow 'clss$  and

   $\text{mset-ccls} :: 'ccls \Rightarrow 'v \text{ clause}$ 
  +
fixes
   $\text{ccls-of-cls} :: 'cls \Rightarrow 'ccls$  and

```

cls-of-ccls :: 'ccls \Rightarrow 'cls **and**
conc-trail :: 'st \Rightarrow ('v, 'v clause) ann-lits **and**
hd-raw-conc-trail :: 'st \Rightarrow ('v, 'cls) ann-lit **and**
raw-clauses :: 'st \Rightarrow 'ccls **and**
conc-backtrack-lvl :: 'st \Rightarrow nat **and**
raw-conc-conflicting :: 'st \Rightarrow 'ccls option **and**

conc-learned-ccls :: 'st \Rightarrow 'v clauses **and**

cons-conc-trail :: ('v, 'cls) ann-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-conc-trail :: 'st \Rightarrow 'st **and**
add-conc-conflict-to-learned-cls :: 'st \Rightarrow 'st **and**
remove-cls :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
update-conc-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st **and**
mark-conflicting :: 'ccls \Rightarrow 'st \Rightarrow 'st **and**
reduce-conc-trail-to :: ('v, 'v clause) ann-lits \Rightarrow 'st \Rightarrow 'st **and**
resolve-conflicting :: 'v literal \Rightarrow 'cls \Rightarrow 'st \Rightarrow 'st **and**

conc-init-state :: 'ccls \Rightarrow 'st **and**
restart-state :: 'st \Rightarrow 'st
assumes
mset-ccls-ccls-of-cls[simp]:
 mset-ccls (ccls-of-cls C) = *mset-cls* C **and**
mset-cls-cls-of-ccls[simp]:
 mset-cls (cls-of-ccls D) = *mset-ccls* D **and**
ex-mset-cls: $\exists a. \text{mset-cls } a = E$
begin

fun *mmset-of-mlit* :: ('v, 'cls) ann-lit \Rightarrow ('v, 'v clause) ann-lit
 where
mmset-of-mlit (Propagated L C) = Propagated L (*mset-cls* C) |
mmset-of-mlit (Decided L) = Decided L

lemma *lit-of-mmset-of-mlit*[simp]:
 lit-of (mmset-of-mlit a) = *lit-of* a
 by (cases a) auto

lemma *lit-of-mmset-of-mlit-set-lit-of-l*[simp]:
 lit-of ' mmset-of-mlit ' set M' = *lits-of-l* M'
 by (induction M') auto

lemma *map-mmset-of-mlit-true-annots-true-cls*[simp]:
 map mmset-of-mlit M' \models_{as} C \longleftrightarrow M' \models_{as} C
 by (simp add: true-annots-true-cls lits-of-def)

definition *conc-init-ccls* $\equiv \lambda S. \text{mset-ccls } (\text{raw-clauses } S) - \text{conc-learned-ccls } S$
abbreviation *conc-conflicting* $\equiv \lambda S. \text{map-option mset-ccls } (\text{raw-conc-conflicting } S)$

notation *in-ccls* (**infix** ! \in ! 50)
notation *insert-ccls* (**infix** !++! 50)

abbreviation *conc-clauses* :: 'st \Rightarrow 'v clauses **where**
conc-clauses S $\equiv \text{mset-ccls } (\text{raw-clauses } S)$

definition *state* :: 'st \Rightarrow 'v cdcl_W-restart-mset **where**

state = ($\lambda S. (conc-trail\ S, conc-init-clss\ S, conc-learned-clss\ S, conc-backtrack-lvl\ S, conc-conflicting\ S)$)

end

We are using an abstract state to abstract away the detail of the implementation: we do not need to know how the clauses are represented internally, we just need to know that they can be converted to multisets.

Weidenbach state is a five-tuple composed of:

1. the trail is a list of decided literals;
2. the initial set of clauses (that is not changed during the whole calculus);
3. the learned clauses (clauses can be added or remove);
4. the maximum level of the trail;
5. the conflicting clause (if any has been found so far).

There are two different clause representation: one for the conflicting clause (*'ccls*, standing for conflicting clause) and one for the initial and learned clauses (*'cls*, standing for clause). The representation of the clauses annotating literals in the trail is slightly different: being able to convert it to *'v CDCL-Abstract-Clause-Representation.clause* is enough (needed for function *hd-raw-conc-trail* below).

There are several axioms to state the independance of the different fields of the state: for example, adding a clause to the learned clauses does not change the trail.

We define the following operations on the elements

- trail: *cons-trail*, *tl-trail*, and *reduce-conc-trail-to*.
- initial set of clauses: a clause can be removed.
- learned clauses: *add-conc-conflict-to-learned-cls* moves the conflicting clause to the learned clauses.
- backtrack level: it can be arbitrary set.
- conflicting clause: there is *resolve-conflicting* that does a resolve step, *mark-conflicting* setting a conflict, and *add-conc-conflict-to-learned-cls* setting the conflicting clause to *None*.

To ease the representation, we consider the clauses all together, where some of them are learned. This eases representation like arrays where the initial set of clause is at the beginning and avoid having an explicit *op* \cup operator.

locale *abs-state_W* =

abs-state_W-ops

— functions for clauses:

mset-cls

mset-clss in-clss insert-clss

— functions for the conflicting clause:

mset-ccls

— Conversion between conflicting and non-conflicting
ccls-of-cls cls-of-ccls

— functions about the state:

— getter:
conc-trail hd-raw-conc-trail raw-clauses conc-backtrack-lvl
raw-conc-conflicting conc-learned-clss

— setter:
cons-conc-trail tl-conc-trail add-conc-conf-to-learned-clss remove-clss update-conc-backtrack-lvl
mark-conflicting reduce-conc-trail-to resolve-conflicting

— Some specific states:
conc-init-state
restart-state

for

mset-cls :: 'cls \Rightarrow 'v clause and

mset-clss :: 'clss \Rightarrow 'v clauses and
in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool and
insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and

mset-ccls :: 'ccls \Rightarrow 'v clause and

ccls-of-cls :: 'cls \Rightarrow 'ccls and
cls-of-ccls :: 'ccls \Rightarrow 'cls and

conc-trail :: 'st \Rightarrow ('v, 'v clause) ann-lits and
hd-raw-conc-trail :: 'st \Rightarrow ('v, 'cls) ann-lit and
raw-clauses :: 'st \Rightarrow 'clss and
conc-backtrack-lvl :: 'st \Rightarrow nat and
raw-conc-conflicting :: 'st \Rightarrow 'ccls option and
conc-learned-clss :: 'st \Rightarrow 'v clauses and

cons-conc-trail :: ('v, 'cls) ann-lit \Rightarrow 'st \Rightarrow 'st and
tl-conc-trail :: 'st \Rightarrow 'st and
add-conc-conf-to-learned-clss :: 'st \Rightarrow 'st and
remove-clss :: 'cls \Rightarrow 'st \Rightarrow 'st and
update-conc-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st and
mark-conflicting :: 'ccls \Rightarrow 'st \Rightarrow 'st and
reduce-conc-trail-to :: ('v, 'v clause) ann-lits \Rightarrow 'st \Rightarrow 'st and
resolve-conflicting :: 'v literal \Rightarrow 'cls \Rightarrow 'st \Rightarrow 'st and

conc-init-state :: 'clss \Rightarrow 'st and
restart-state :: 'st \Rightarrow 'st +

assumes

— Definition of *hd-raw-trail*:
hd-raw-conc-trail:
conc-trail S \neq [] \implies mmset-of-mlit (hd-raw-conc-trail S) = hd (conc-trail S) and

cons-conc-trail:
 $\bigwedge S'. \text{undefined-lit (conc-trail st) (lit-of L)} \implies$
 $\text{state st} = (M, S') \implies$
 $\text{state (cons-conc-trail L st)} = (\text{mmset-of-mlit L} \# M, S') \text{ and}$

tl-conc-trail:

$\bigwedge S'. \text{state } st = (M, S') \implies \text{state } (\text{tl-conc-trail } st) = (\text{tl } M, S') \text{ and}$

remove-cls:

$\bigwedge S'. \text{state } st = (M, N, U, S') \implies$
 $\text{state } (\text{remove-cls } C \text{ } st) =$
 $(M, \text{removeAll-mset } (\text{mset-cls } C) \text{ } N, \text{removeAll-mset } (\text{mset-cls } C) \text{ } U, S') \text{ and}$

add-conc-confl-to-learned-cls:

$\text{no-dup } (\text{conc-trail } st) \implies \text{state } st = (M, N, U, k, \text{Some } F) \implies$
 $\text{state } (\text{add-conc-confl-to-learned-cls } st) =$
 $(M, N, \{\#F\# \} + U, k, \text{None}) \text{ and}$

update-conc-backtrack-lvl:

$\bigwedge S'. \text{state } st = (M, N, U, k, S') \implies$
 $\text{state } (\text{update-conc-backtrack-lvl } k' \text{ } st) = (M, N, U, k', S') \text{ and}$

mark-conflicting:

$\text{state } st = (M, N, U, k, \text{None}) \implies$
 $\text{state } (\text{mark-conflicting } E \text{ } st) = (M, N, U, k, \text{Some } (\text{mset-ccls } E)) \text{ and}$

conc-conflicting-mark-conflicting[simp]:

$\text{raw-conc-conflicting } (\text{mark-conflicting } E \text{ } st) = \text{Some } E \text{ and}$

resolve-conflicting:

$\text{state } st = (M, N, U, k, \text{Some } F) \implies -L' \in \# F \implies L' \in \# \text{mset-cls } D \implies$
 $\text{state } (\text{resolve-conflicting } L' \text{ } D \text{ } st) =$
 $(M, N, U, k, \text{Some } (\text{cdcl}_W\text{-restart-mset.resolve-cls } L' \text{ } F \text{ } (\text{mset-cls } D))) \text{ and}$

conc-init-state:

$\text{state } (\text{conc-init-state } Ns) = ([], \text{mset-clss } Ns, \{\#\}, 0, \text{None}) \text{ and}$

— Properties about restarting *restart-state*:

conc-trail-restart-state[simp]: $\text{conc-trail } (\text{restart-state } S) = [] \text{ and}$
conc-init-clss-restart-state[simp]: $\text{conc-init-clss } (\text{restart-state } S) = \text{conc-init-clss } S \text{ and}$
conc-learned-clss-restart-state[intro]:
 $\text{conc-learned-clss } (\text{restart-state } S) \subseteq \# \text{conc-learned-clss } S \text{ and}$
conc-backtrack-lvl-restart-state[simp]: $\text{conc-backtrack-lvl } (\text{restart-state } S) = 0 \text{ and}$
conc-conflicting-restart-state[simp]: $\text{conc-conflicting } (\text{restart-state } S) = \text{None} \text{ and}$

— Properties about *reduce-conc-trail-to*:

reduce-conc-trail-to[simp]:

$\bigwedge S'. \text{conc-trail } st = M2 \text{ } @ \text{ } M1 \implies \text{state } st = (M, S') \implies$
 $\text{state } (\text{reduce-conc-trail-to } M1 \text{ } st) = (M1, S') \text{ and}$

learned-clauses:

$\text{conc-learned-clss } S \subseteq \# \text{conc-clauses } S$

begin

lemma

— Properties about the trail *conc-trail*:

conc-trail-cons-conc-trail[simp]:

$\text{undefined-lit } (\text{conc-trail } st) \text{ } (\text{lit-of } L) \implies$
 $\text{conc-trail } (\text{cons-conc-trail } L \text{ } st) = \text{mmset-of-mlit } L \text{ } \# \text{conc-trail } st \text{ and}$

conc-trail-tl-conc-trail[simp]:

$\text{conc-trail } (\text{tl-conc-trail } st) = \text{tl } (\text{conc-trail } st) \text{ and}$

conc-trail-add-conc-confl-to-learned-cls[simp]:

$no_dup (conc_trail\ st) \implies conc_conflicting\ st \neq None \implies$
 $conc_trail (add_conc_confl_to_learned_cls\ st) = conc_trail\ st$ **and**
 $conc_trail_remove_cls[simp]:$
 $conc_trail (remove_cls\ C\ st) = conc_trail\ st$ **and**
 $conc_trail_update_conc_backtrack_lvl[simp]:$
 $conc_trail (update_conc_backtrack_lvl\ k\ st) = conc_trail\ st$ **and**
 $conc_trail_mark_conflicting[simp]:$
 $raw_conc_conflicting\ st = None \implies conc_trail (mark_conflicting\ E\ st) = conc_trail\ st$ **and**
 $conc_trail_resolve_conflicting[simp]:$
 $conc_conflicting\ st = Some\ F \implies -L' \in \# F \implies L' \in \# mset_cls\ D \implies$
 $conc_trail (resolve_conflicting\ L'\ D\ st) = conc_trail\ st$ **and**

— Properties about the initial clauses *conc-init-clss*:

$conc_init_clss_cons_conc_trail[simp]:$
 $undefined_lit (conc_trail\ st) (lit_of\ L) \implies$
 $conc_init_clss (cons_conc_trail\ L\ st) = conc_init_clss\ st$
and

— Properties about the learned clauses *conc-learned-clss*:

$conc_learned_clss_cons_conc_trail[simp]:$
 $undefined_lit (conc_trail\ st) (lit_of\ L) \implies$
 $conc_learned_clss (cons_conc_trail\ L\ st) = conc_learned_clss\ st$ **and**
 $conc_learned_clss_tl_conc_trail[simp]:$
 $conc_learned_clss (tl_conc_trail\ st) = conc_learned_clss\ st$ **and**
 $conc_learned_clss_add_conc_confl_to_learned_cls[simp]:$
 $no_dup (conc_trail\ st) \implies conc_conflicting\ st = Some\ C' \implies$
 $conc_learned_clss (add_conc_confl_to_learned_cls\ st) = \{\#C'\#\} + conc_learned_clss\ st$ **and**
 $conc_learned_clss_remove_cls[simp]:$
 $conc_learned_clss (remove_cls\ C\ st) = removeAll_mset (mset_cls\ C) (conc_learned_clss\ st)$ **and**
 $conc_learned_clss_update_conc_backtrack_lvl[simp]:$
 $conc_learned_clss (update_conc_backtrack_lvl\ k\ st) = conc_learned_clss\ st$ **and**
 $conc_learned_clss_mark_conflicting[simp]:$
 $raw_conc_conflicting\ st = None \implies$
 $conc_learned_clss (mark_conflicting\ E\ st) = conc_learned_clss\ st$ **and**
 $conc_learned_clss_clss_resolve_conflicting[simp]:$
 $conc_conflicting\ st = Some\ F \implies -L' \in \# F \implies L' \in \# mset_cls\ D \implies$
 $conc_learned_clss (resolve_conflicting\ L'\ D\ st) = conc_learned_clss\ st$ **and**

— Properties about the backtracking level *conc-backtrack-lvl*:

$conc_backtrack_lvl_cons_conc_trail[simp]:$
 $undefined_lit (conc_trail\ st) (lit_of\ L) \implies$
 $conc_backtrack_lvl (cons_conc_trail\ L\ st) = conc_backtrack_lvl\ st$ **and**
 $conc_backtrack_lvl_tl_conc_trail[simp]:$
 $conc_backtrack_lvl (tl_conc_trail\ st) = conc_backtrack_lvl\ st$ **and**
 $conc_backtrack_lvl_add_conc_confl_to_learned_cls[simp]:$
 $no_dup (conc_trail\ st) \implies conc_conflicting\ st \neq None \implies$
 $conc_backtrack_lvl (add_conc_confl_to_learned_cls\ st) = conc_backtrack_lvl\ st$ **and**
 $conc_backtrack_lvl_remove_cls[simp]:$
 $conc_backtrack_lvl (remove_cls\ C\ st) = conc_backtrack_lvl\ st$ **and**
 $conc_backtrack_lvl_update_conc_backtrack_lvl[simp]:$
 $conc_backtrack_lvl (update_conc_backtrack_lvl\ k\ st) = k$ **and**
 $conc_backtrack_lvl_mark_conflicting[simp]:$
 $raw_conc_conflicting\ st = None \implies$
 $conc_backtrack_lvl (mark_conflicting\ E\ st) = conc_backtrack_lvl\ st$ **and**
 $conc_backtrack_lvl_clss_clss_resolve_conflicting[simp]:$
 $conc_conflicting\ st = Some\ F \implies -L' \in \# F \implies L' \in \# mset_cls\ D \implies$

conc-backtrack-lvl (*resolve-conflicting* $L' D$ st) = *conc-backtrack-lvl* st **and**

— Properties about the conflicting clause *conc-conflicting*:

conc-conflicting-cons-conc-trail[*simp*]:

undefined-lit (*conc-trail* st) (*lit-of* L) \implies

conc-conflicting (*cons-conc-trail* L st) = *conc-conflicting* st **and**

conc-conflicting-tl-conc-trail[*simp*]:

conc-conflicting (*tl-conc-trail* st) = *conc-conflicting* st **and**

conc-conflicting-add-conc-conf-to-learned-cls[*simp*]:

no-dup (*conc-trail* st) \implies *conc-conflicting* st = *Some* C' \implies

conc-conflicting (*add-conc-conf-to-learned-cls* st) = *None*

and

raw-conc-conflicting-add-conc-conf-to-learned-cls[*simp*]:

no-dup (*conc-trail* st) \implies *conc-conflicting* st = *Some* C' \implies

raw-conc-conflicting (*add-conc-conf-to-learned-cls* st) = *None* **and**

conc-conflicting-remove-cls[*simp*]:

conc-conflicting (*remove-cls* C st) = *conc-conflicting* st **and**

conc-conflicting-update-conc-backtrack-lvl[*simp*]:

conc-conflicting (*update-conc-backtrack-lvl* k st) = *conc-conflicting* st **and**

conc-conflicting-clss-clss-resolve-conflicting[*simp*]:

conc-conflicting st = *Some* F \implies $-L' \in \# F \implies L' \in \# \text{mset-cls } D \implies$

conc-conflicting (*resolve-conflicting* $L' D$ st) =

Some (*cdcl_W-restart-mset.resolve-cls* $L' F$ (*mset-cls* D)) **and**

— Properties about the initial state *conc-init-state*:

conc-init-state-conc-trail[*simp*]: *conc-trail* (*conc-init-state* Ns) = [] **and**

conc-init-state-clss[*simp*]: *conc-init-clss* (*conc-init-state* Ns) = *mset-clss* Ns **and**

conc-init-state-conc-learned-clss[*simp*]: *conc-learned-clss* (*conc-init-state* Ns) = {#} **and**

conc-init-state-conc-backtrack-lvl[*simp*]: *conc-backtrack-lvl* (*conc-init-state* Ns) = 0 **and**

conc-init-state-conc-conflicting[*simp*]: *conc-conflicting* (*conc-init-state* Ns) = *None* **and**

— Properties about *reduce-conc-trail-to*:

trail-reduce-conc-trail-to[*simp*]:

conc-trail st = $M2 @ M1 \implies$ *conc-trail* (*reduce-conc-trail-to* $M1$ st) = $M1$ **and**

conc-learned-clss-reduce-conc-trail-to[*simp*]:

conc-trail st = $M2 @ M1 \implies$

conc-learned-clss (*reduce-conc-trail-to* $M1$ st) = *conc-learned-clss* st **and**

conc-backtrack-lvl-reduce-conc-trail-to[*simp*]:

conc-trail st = $M2 @ M1 \implies$

conc-backtrack-lvl (*reduce-conc-trail-to* $M1$ st) = *conc-backtrack-lvl* st **and**

conc-conflicting-reduce-conc-trail-to[*simp*]:

conc-trail st = $M2 @ M1 \implies$

conc-conflicting (*reduce-conc-trail-to* $M1$ st) = *conc-conflicting* st

using *cons-conc-trail*[*of* st L *conc-trail* st *snd* (*state* st)] *tl-conc-trail*[*of* st]

add-conc-conf-to-learned-cls[*of* st *conc-trail* st - - -]

update-conc-backtrack-lvl[*of* st - - - - k]

mark-conflicting[*of* st - - - - E]

remove-cls[*of* st - - - - C]

conc-init-state[*of* Ns]

reduce-conc-trail-to[*of* st]

resolve-conflicting[*of* st - - - - $F L' D$]

unfolding *state-def* *Product-Type.prod.inject* **by** *auto*

lemma

conc-init-clss-tl-conc-trail[*simp*]:

```

    conc-init-clss (tl-conc-trail st) = conc-init-clss st and
conc-init-clss-add-conc-confl-to-learned-clss[simp]:
    no-dup (conc-trail st)  $\implies$  conc-conflicting st  $\neq$  None  $\implies$ 
        conc-init-clss (add-conc-confl-to-learned-clss st) = conc-init-clss st and
conc-init-clss-remove-clss[simp]:
    conc-init-clss (remove-clss C st) = removeAll-mset (mset-clss C) (conc-init-clss st) and
conc-init-clss-update-conc-backtrack-lvl[simp]:
    conc-init-clss (update-conc-backtrack-lvl k st) = conc-init-clss st and
conc-init-clss-mark-conflicting[simp]:
    raw-conc-conflicting st = None  $\implies$ 
        conc-init-clss (mark-conflicting E st) = conc-init-clss st and
conc-init-clss-resolve-conflicting[simp]:
    conc-conflicting st = Some F  $\implies$   $\neg L' \in \# F \implies L' \in \#$  mset-clss D  $\implies$ 
        conc-init-clss (resolve-conflicting L' D st) = conc-init-clss st and
conc-init-clss-reduce-conc-trail-to[simp]:
    conc-trail st = M2 @ M1  $\implies$ 
        conc-init-clss (reduce-conc-trail-to M1 st) = conc-init-clss st
using tl-conc-trail[of st]
add-conc-confl-to-learned-clss[of st conc-trail st - - -]
update-conc-backtrack-lvl[of st - - - - k]
mark-conflicting[of st - - - - E]
remove-clss[of st - - - - C]
reduce-conc-trail-to[of st M2 M1]
resolve-conflicting[of st - - - - F L' D]
unfolding state-def Product-Type.prod.inject by fast+

```

abbreviation *incr-lvl* :: 'st \Rightarrow 'st **where**
incr-lvl S \equiv update-conc-backtrack-lvl (conc-backtrack-lvl S + 1) S

abbreviation *state-eq* :: 'st \Rightarrow 'st \Rightarrow bool (**infix** \sim 36) **where**
S \sim *T* \equiv state S \sim_m state T

lemma *state-eq-sym*:
S \sim *T* \longleftrightarrow *T* \sim *S*
using cdcl_W-restart-mset.state-eq-sym **by** blast

lemma *state-eq-trans*:
S \sim *T* \implies *T* \sim *U* \implies *S* \sim *U*
using cdcl_W-restart-mset.state-eq-trans **by** blast

lemma *conc-clauses-init-learned*: conc-clauses S = conc-init-clss S + conc-learned-clss S
using learned-clauses[*of* S] **by** (auto *simp*: conc-init-clss-def multiset-eq-iff subseq-mset-def)

lemma
shows
state-eq-conc-trail: *S* \sim *T* \implies conc-trail S = conc-trail T **and**
state-eq-conc-init-clss: *S* \sim *T* \implies conc-init-clss S = conc-init-clss T **and**
state-eq-conc-learned-clss: *S* \sim *T* \implies conc-learned-clss S = conc-learned-clss T **and**
state-eq-conc-backtrack-lvl: *S* \sim *T* \implies conc-backtrack-lvl S = conc-backtrack-lvl T **and**
state-eq-conc-conflicting: *S* \sim *T* \implies conc-conflicting S = conc-conflicting T **and**
state-eq-clauses: *S* \sim *T* \implies conc-clauses S = conc-clauses T **and**
state-eq-undefined-lit:
S \sim *T* \implies undefined-lit (conc-trail S) L = undefined-lit (conc-trail T) L
unfolding state-def cdcl_W-restart-mset.state-eq-def conc-clauses-init-learned
by (auto *simp*: cdcl_W-restart-mset-state)

We combine all simplification rules about $op \sim$ in a single list of theorems. While they are handy as simplification rule as long as we are working on the state, they also cause a *huge* slow-down in all other cases.

lemmas *state-simp = state-eq-conc-trail state-eq-conc-init-clss state-eq-conc-learned-clss*
state-eq-conc-backtrack-lvl state-eq-conc-conflicting state-eq-clauses state-eq-undefined-lit

lemma *atms-of-ms-conc-learned-clss-restart-state-in-atms-of-ms-conc-learned-clssI[intro]:*
 $x \in \text{atms-of-mm } (\text{conc-learned-clss } (\text{restart-state } S)) \implies x \in \text{atms-of-mm } (\text{conc-learned-clss } S)$
by (*meson atms-of-ms-mono conc-learned-clss-restart-state set-mset-mono subsetCE*)

lemma *clauses-reduce-conc-trail-to[simp]:*
 $\text{conc-trail } S = M2 @ M1 \implies \text{conc-clauses } (\text{reduce-conc-trail-to } M1 S) = \text{conc-clauses } S$
unfolding *conc-clauses-init-learned* **by** *auto*

lemma *in-get-all-ann-decomposition-conc-trail-update-conc-trail[simp]:*
assumes *H: (L # M1, M2) ∈ set (get-all-ann-decomposition (conc-trail S))*
shows *conc-trail (reduce-conc-trail-to M1 S) = M1*
using *assms* **by** *auto*

lemma *raw-conc-conflicting-cons-conc-trail[simp]:*
assumes *undefined-lit (conc-trail S) (lit-of L)*
shows
 $\text{raw-conc-conflicting } (\text{cons-conc-trail } L S) = \text{None} \longleftrightarrow \text{raw-conc-conflicting } S = \text{None}$
using *assms conc-conflicting-cons-conc-trail[of S L] map-option-is-None* **by** *fastforce+*

lemma *raw-conc-conflicting-update-backtrack-lvl[simp]:*
 $\text{raw-conc-conflicting } (\text{update-conc-backtrack-lvl } k S) = \text{None} \longleftrightarrow \text{raw-conc-conflicting } S = \text{None}$
using *map-option-is-None conc-conflicting-update-conc-backtrack-lvl[of k S]* **by** *fastforce+*

end — end of *state_W* locale

7.2.4 CDCL Rules

locale *abs-conflict-driven-clause-learning_W =*
abs-state_W
— functions for clauses:
mset-cls
mset-clss in-clss insert-clss

— functions for the conflicting clause:
mset-ccls

— conversion
ccls-of-cls cls-of-ccls

— functions for the state:
— access functions:
conc-trail hd-raw-conc-trail raw-clauses conc-backtrack-lvl
raw-conc-conflicting conc-learned-clss
— changing state:
cons-conc-trail tl-conc-trail add-conc-conf-to-learned-cls remove-cls update-conc-backtrack-lvl
mark-conflicting reduce-conc-trail-to resolve-conflicting

— get state:
conc-init-state

```

  restart-state
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and

  mset-clss :: 'clss  $\Rightarrow$  'v clauses and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and

  mset-ccls :: 'ccls  $\Rightarrow$  'v clause and

  ccls-of-cls :: 'cls  $\Rightarrow$  'ccls and
  cls-of-ccls :: 'ccls  $\Rightarrow$  'cls and

  conc-trail :: 'st  $\Rightarrow$  ('v, 'v clause) ann-lits and
  hd-raw-conc-trail :: 'st  $\Rightarrow$  ('v, 'cls) ann-lit and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  conc-backtrack-lvl :: 'st  $\Rightarrow$  nat and
  raw-conc-conflicting :: 'st  $\Rightarrow$  'ccls option and
  conc-learned-clss :: 'st  $\Rightarrow$  'v clauses and

  cons-conc-trail :: ('v, 'cls) ann-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-conc-trail :: 'st  $\Rightarrow$  'st and
  add-conc-conf-to-learned-cls :: 'st  $\Rightarrow$  'st and
  remove-cls :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  update-conc-backtrack-lvl :: nat  $\Rightarrow$  'st  $\Rightarrow$  'st and
  mark-conflicting :: 'ccls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  reduce-conc-trail-to :: ('v, 'v clause) ann-lits  $\Rightarrow$  'st  $\Rightarrow$  'st and
  resolve-conflicting :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and

  conc-init-state :: 'clss  $\Rightarrow$  'st and
  restart-state :: 'st  $\Rightarrow$  'st
begin

lemma clauses-state-conc-clauses[simp]: cdclW-restart-mset.clauses (state S) = conc-clauses S
  apply (cases state S)
  unfolding cdclW-restart-mset.clauses-def conc-clauses-init-learned
  unfolding cdclW-restart-mset-state state-def
  by simp

lemma conflicting-None-iff-raw-conc-conflicting[simp]:
  conflicting (state S) = None  $\longleftrightarrow$  raw-conc-conflicting S = None
  unfolding state-def conflicting-def by simp

lemma trail-state-add-conc-conf-to-learned-cls:
  no-dup (conc-trail S)  $\implies$  conc-conflicting S  $\neq$  None  $\implies$ 
    trail (state (add-conc-conf-to-learned-cls S)) = trail (state S)
  unfolding trail-def state-def by simp

lemma trail-state-update-backtrack-lvl:
  trail (state (update-conc-backtrack-lvl i S)) = trail (state S)
  unfolding trail-def state-def by simp

lemma trail-state-update-conflicting:
  raw-conc-conflicting S = None  $\implies$  trail (state (mark-conflicting i S)) = trail (state S)
  unfolding trail-def state-def by simp

```

lemma *trail-state-conc-trail*[simp]:
trail (state S) = conc-trail S
unfolding *trail-def state-def* **by** *auto*

lemma *init-clss-state-conc-init-clss*[simp]:
init-clss (state S) = conc-init-clss S
unfolding *init-clss-def state-def* **by** *auto*

lemma *learned-clss-state-conc-learned-clss*[simp]:
learned-clss (state S) = conc-learned-clss S
unfolding *learned-clss-def state-def* **by** *auto*

lemma *tl-trail-state-tl-con-trail*[simp]:
tl-trail (state S) = state (tl-conc-trail S)
by (*auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *add-learned-clss-state-add-conc-confl-to-learned-clss*[simp]:
assumes *no-dup (conc-trail S) and raw-conc-conflicting S = Some D*
shows *update-conflicting None (add-learned-clss (mset-ccls D) (state S)) =*
state (add-conc-confl-to-learned-clss S)
using *assms by (auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *state-cons-cons-trail-cons-trail*[simp]:
undefined-lit (trail (state S)) (lit-of L) \implies
cons-trail (mmset-of-mlit L) (state S) = state (cons-conc-trail L S)
by (*auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *state-cons-cons-trail-cons-trail-propagated*[simp]:
undefined-lit (trail (state S)) K \implies
cons-trail (Propagated K (mset-clss C)) (state S) = state (cons-conc-trail (Propagated K C) S)
using *state-cons-cons-trail-cons-trail[of S Propagated K C] by simp*

lemma *state-cons-cons-trail-cons-trail-propagated-ccls*[simp]:
undefined-lit (trail (state S)) K \implies
cons-trail (Propagated K (mset-ccls C)) (state S) =
state (cons-conc-trail (Propagated K (cls-of-ccls C)) S)
using *state-cons-cons-trail-cons-trail[of S Propagated K (cls-of-ccls C)] by simp*

lemma *state-cons-cons-trail-cons-trail-decided*[simp]:
undefined-lit (trail (state S)) K \implies
cons-trail (Decided K) (state S) = state (cons-conc-trail (Decided K) S)
using *state-cons-cons-trail-cons-trail[of S Decided K] by simp*

lemma *state-mark-conflicting-update-conflicting*[simp]:
assumes *raw-conc-conflicting S = None*
shows
update-conflicting (Some (mset-ccls D)) (state S) = state (mark-conflicting D S)
update-conflicting (Some (mset-clss D')) (state S) =
state (mark-conflicting ((cls-of-clss D')) S)
using *assms by (auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*

init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *update-backtrack-lvl-state[simp]:*
update-backtrack-lvl i (state S) = state (update-conc-backtrack-lvl i S)
by (*auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *conc-conflicting-conflicting[simp]:*
conflicting (state S) = conc-conflicting S
by (*auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *update-conflicting-resolve-state-mark-conflicting[simp]:*
raw-conc-conflicting S = Some D' \implies $\neg L \in \# \text{mset-ccls } D' \implies L \in \# \text{mset-cls } E' \implies$
update-conflicting (Some (remove1-mset ($\neg L$) (mset-ccls D') $\# \cup$ remove1-mset L (mset-cls E'))
(state (tl-conc-trail S))) =
state (resolve-conflicting L E' (tl-conc-trail S))
by (*auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *add-learned-update-backtrack-update-conflicting[simp]:*
no-dup (conc-trail S) \implies raw-conc-conflicting S = Some D' \implies add-learned-cls (mset-ccls D')
(update-backtrack-lvl i
(update-conflicting None
(state S))) =
state (add-conc-conflict-to-learned-cls (update-conc-backtrack-lvl i S))
by (*auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail*
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp)

lemma *conc-backtrack-lvl-backtrack-lvl[simp]:*
backtrack-lvl (state S) = conc-backtrack-lvl S
unfolding *state-def* **by** (*auto simp: cdcl_W-restart-mset-state*)

lemma *state-state:*
cdcl_W-restart-mset.state (state S) = (trail (state S), init-clss (state S), learned-clss (state S),
backtrack-lvl (state S), conflicting (state S))
by (*simp*)

lemma *state-reduce-conc-trail-to-reduce-conc-trail-to[simp]:*
assumes [*simp*]: *conc-trail S = M2 @ M1*
shows *cdcl_W-restart-mset.reduce-trail-to M1 (state S) = state (reduce-conc-trail-to M1 S) (is ?RS = ?SR)*

proof —

have 1: *trail ?SR = trail ?RS*
apply (*subst state-def*)
apply (*auto simp add: cdcl_W-restart-mset.trail-reduce-trail-to-drop*)
apply (*auto simp: trail-def*)
done

have 2: *init-clss ?SR = init-clss ?RS*
by *simp*

have 3: *learned-clss* ?*SR* = *learned-clss* ?*RS*
by *simp*

have 4: *backtrack-lvl* ?*SR* = *backtrack-lvl* ?*RS*
by *simp*

have 5: *conflicting* ?*SR* = *conflicting* ?*RS*
by *simp*

show ?*thesis*

using 1 2 3 4 5 **apply** –

apply (*subst* (*asm*) *trail-def*, *subst* (*asm*) *trail-def*)

apply (*subst* (*asm*) *init-clss-def*, *subst* (*asm*) *init-clss-def*)

apply (*subst* (*asm*) *learned-clss-def*, *subst* (*asm*) *learned-clss-def*)

apply (*subst* (*asm*) *backtrack-lvl-def*, *subst* (*asm*) *backtrack-lvl-def*)

apply (*subst* (*asm*) *conflicting-def*, *subst* (*asm*) *conflicting-def*)

apply (*cases* *state* (*reduce-conc-trail-to* *M1* *S*))

apply (*cases* *cdcl_W-restart-mset.reduce-trail-to* *M1* (*state* *S*))

by *simp*

qed

lemma *state-conc-init-state*: *state* (*conc-init-state* *N*) = *init-state* (*mset-clss* *N*)

by (*auto simp: cdcl_W-restart-mset-state state-def simp del: trail-state-conc-trail
init-clss-state-conc-init-clss
learned-clss-state-conc-learned-clss local.state-simp*)

More robust version of *in-mset-clss-exists-preimage*:

lemma *in-clauses-preimage*:

assumes *b*: *b* ∈ # *cdcl_W-restart-mset.clauses* (*state* *C*)

shows ∃ *b'*. *b'* !∈! *raw-clauses* *C* ∧ *mset-cls* *b'* = *b*

proof –

have *b* ∈ # *conc-clauses* *C*

using *b* **by** *auto*

from *in-mset-clss-exists-preimage*[*OF this*] **show** ?*thesis* .

qed

lemma *state-reduce-conc-trail-to-reduce-conc-trail-to-decomp*[*simp*]:

assumes (*P* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*conc-trail* *S*))

shows *cdcl_W-restart-mset.reduce-trail-to* *M1* (*state* *S*) = *state* (*reduce-conc-trail-to* *M1* *S*)

using *assms* **by** *auto*

inductive *propagate-abs* :: '*st* ⇒ '*st* ⇒ *bool* **for** *S* :: '*st* **where**

propagate-abs-rule: *conc-conflicting* *S* = *None* ⇒

E !∈! *raw-clauses* *S* ⇒

L ∈ # *mset-cls* *E* ⇒

conc-trail *S* ⊨_{as} *CNot* (*mset-cls* *E* – {#*L*#}) ⇒

undefined-lit (*conc-trail* *S*) *L* ⇒

T ~ *cons-conc-trail* (*Propagated* *L* *E*) *S* ⇒

propagate-abs *S* *T*

inductive-cases *propagate-absE*: *propagate-abs* *S* *T*

lemma *propagate-propagate-abs*:

cdcl_W-restart-mset.propagate (*state* *S*) (*state* *T*) ⇔ *propagate-abs* *S* *T* (**is** ?*mset* ⇔ ?*abs*)

proof

```

assume ?abs
then obtain  $E\ L$  where
   $\text{confl: conc-conflicting } S = \text{None}$  and
   $E: E \notin \text{raw-clauses } S$  and
   $L: L \in \# \text{ mset-cls } E$  and
   $\text{tr-}E: \text{conc-trail } S \models_{\text{as}} \text{CNot } (\text{mset-cls } E - \{\#L\# \})$  and
   $\text{undef: undefined-lit } (\text{conc-trail } S) L$  and
   $T: T \sim \text{cons-conc-trail } (\text{Propagated } L\ E) S$ 
by ( $\text{auto elim: propagate-absE}$ )

show ?mset
apply ( $\text{rule cdcl}_W\text{-restart-mset.propagate-rule}$ )
  using  $\text{confl}$  apply  $\text{auto}[]$ 
  using  $E$  apply  $\text{auto}[]$ 
  using  $L$  apply  $\text{auto}[]$ 
  using  $\text{tr-}E$  apply  $\text{auto}[]$ 
  using  $\text{undef}$  apply ( $\text{auto simp:}[]$ )
using  $\text{undef } T$  unfolding  $\text{cdcl}_W\text{-restart-mset-state-eq-eq state-def cons-trail-def}$  by  $\text{simp}$ 
next
assume ?mset
then obtain  $E\ L$  where
   $\text{conc-conflicting } S = \text{None}$  and
   $E \notin \text{raw-clauses } S$  and
   $L \in \# \text{ mset-cls } E$  and
   $\text{conc-trail } S \models_{\text{as}} \text{CNot } (\text{mset-cls } E - \{\#L\# \})$  and
   $\text{undefined-lit } (\text{conc-trail } S) L$  and
   $\text{state } T \sim_m \text{cons-trail } (\text{Propagated } L\ (\text{mset-cls } E)) (\text{state } S)$ 
by ( $\text{auto elim!: cdcl}_W\text{-restart-mset.propagateE dest!: in-clauses-preimage}$ 
   $\text{simp: cdcl}_W\text{-restart-mset.clauses-def}$ )
then show ?abs
by ( $\text{auto intro!: propagate-abs-rule}$ )
qed

lemma  $\text{propagate-compatible-abs:}$ 
assumes  $SS': S \sim_m \text{state } S'$  and  $\text{abs: cdcl}_W\text{-restart-mset.propagate } S\ T$ 
obtains  $U$  where  $\text{propagate-abs } S'\ U$  and  $T \sim_m \text{state } U$ 
proof –
obtain  $E\ L$  where
   $\text{confl: conflicting } S = \text{None}$  and
   $E: E \in \# \text{ cdcl}_W\text{-restart-mset.clauses } S$  and
   $L: L \in \# E$  and
   $\text{tr: trail } S \models_{\text{as}} \text{CNot } (E - \{\#L\# \})$  and
   $\text{undef: undefined-lit } (\text{trail } S) L$  and
   $T: T \sim_m \text{cons-trail } (\text{Propagated } L\ E) S$ 
using  $\text{abs}$  by ( $\text{auto elim!: cdcl}_W\text{-restart-mset.propagateE dest!: in-clauses-preimage}$ 
   $\text{simp: cdcl}_W\text{-restart-mset.clauses-def}$ )
then obtain  $E'$  where
   $E': E' \notin \text{raw-clauses } S'$  and  $[\text{simp}]: E = \text{mset-cls } E'$ 
by ( $\text{metis } SS' \text{ cdcl}_W\text{-restart-mset.state-eq-clauses in-clauses-preimage}$ )
let  $?U = \text{cons-conc-trail } (\text{Propagated } L\ E') S'$ 
have  $\text{propagate-abs } S'\ ?U$ 
apply ( $\text{rule propagate-abs-rule}$ )
  using  $\text{confl } SS'$  apply  $\text{simp}$ 
  using  $E' SS'$  apply  $\text{simp}$ 
  using  $L$  apply  $\text{simp}$ 
  using  $\text{tr } SS'$  apply  $\text{simp}$ 

```

```

    using undef SS' apply simp
    using undef SS' by simp
  moreover have  $T \sim_m \text{state } ?U$ 
    using  $T \text{ SS' undef by (auto simp: cdcl}_W\text{-restart-mset-state-eq-eq)}$ 
  ultimately show thesis using that by blast
qed

```

interpretation *propagate-abs: relation-relation-abs cdcl_W-restart-mset.propagate propagate-abs state*
 $\lambda\cdot. \text{True}$
apply *unfold-locales*
apply (*simp add: propagate-propagate-abs*)
using *propagate-compatible-abs* **by** *blast*

inductive *conflict-abs* :: $'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
conflict-abs-rule:
 $\text{conc-conflicting } S = \text{None} \implies$
 $D \notin \# \text{ raw-clauses } S \implies$
 $\text{conc-trail } S \models_{\text{as}} \text{CNot } (\text{mset-cls } D) \implies$
 $T \sim \text{mark-conflicting } (\text{ccls-of-cls } D) S \implies$
 $\text{conflict-abs } S T$

inductive-cases *conflict-absE*: $\text{conflict-abs } S T$

lemma *conflict-conflict-abs*:

$\text{cdcl}_W\text{-restart-mset.conflict } (\text{state } S) (\text{state } T) \longleftrightarrow \text{conflict-abs } S T$ (**is** $?mset \longleftrightarrow ?abs$)

proof

```

  assume ?abs
  then obtain D where
    confl: conc-conflicting S = None and
    D: D ∉ # raw-clauses S and
    tr-D: conc-trail S ⊨as CNot (mset-cls D) and
    T: T ∼ mark-conflicting (ccls-of-cls D) S
    by (auto elim!: conflict-absE)
  show ?mset
    apply (rule cdclW-restart-mset.conflict-rule)
      using confl apply simp
      using D apply auto[]
      using tr-D apply simp
      using T confl apply auto
    done
  next
    assume ?mset
    then obtain D where
      confl: conflicting (state S) = None and
      D: D ∈ # cdclW-restart-mset.clauses (state S) and
      tr-D: trail (state S) ⊨as CNot D and
      T: state T ∼m update-conflicting (Some D) (state S)
      by (cases state S) (auto elim: cdclW-restart-mset.conflictE)
    obtain D' where D': D' ∉ # raw-clauses S and DD'[simp]: D = mset-cls D'
      using D by (auto dest!: in-mset-clss-exists-preimage)[]
    show ?abs
      apply (rule conflict-abs-rule)
        using confl apply simp
        using D' apply simp
        using tr-D apply simp
        using T confl by auto

```

qed

lemma *conflict-compatible-abs*:

assumes SS' : $S \sim_m \text{state } S'$ **and** *conflict*: $\text{cdcl}_W\text{-restart-mset.conflict } S \ T$
obtains U **where** *conflict-abs* $S' \ U$ **and** $T \sim_m \text{state } U$

proof –

obtain D **where**

conf: *conflicting* $S = \text{None}$ **and**
 D : $D \in \# \text{ cdcl}_W\text{-restart-mset.clauses } S$ **and**
 $\text{tr-}D$: $\text{trail } S \models_{\text{as}} \text{CNot } D$ **and**
 T : $T \sim_m \text{update-conflicting } (\text{Some } D) \ S$

using *conflict* **by** (*auto elim*: $\text{cdcl}_W\text{-restart-mset.conflictE}$)

obtain D' **where** D' : $D' \notin \text{raw-clauses } S'$ **and** $DD'[\text{simp}]$: $D = \text{mset-cls } D'$

using $D \ SS'$ **by** (*auto dest*!: $\text{in-mset-clss-exists-preimage}$)[]

let $?U = \text{mark-conflicting } (\text{ccls-of-cls } D') \ S'$

have *conflict-abs* $S' \ ?U$

apply (*rule conflict-abs-rule*)

using *confl* SS' **apply** *simp*

using $D' \ SS'$ **apply** *simp*

using $\text{tr-}D \ SS'$ **apply** *simp*

using T **by** *auto*

moreover have $T \sim_m \text{state } ?U$

using $T \ SS' \ \text{confl}$ **by** (*auto simp*: $\text{cdcl}_W\text{-restart-mset.state-eq-eq}$)

ultimately show *thesis* **using** *that*[*of* $?U$] **by** *fast*

qed

interpretation *conflict-abs*: *relation-relation-abs* $\text{cdcl}_W\text{-restart-mset.conflict}$ *conflict-abs* *state*

$\lambda\cdot. \text{True}$

apply *unfold-locals*

apply (*simp add*: *conflict-conflict-abs*)

using *conflict-compatible-abs* **by** *metis*

inductive *backtrack-abs* :: $'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**

backtrack-abs-rule:

raw-conc-conflicting $S = \text{Some } D \implies$

$L \in \# \text{ mset-ccls } D \implies$

$(\text{Decided } K \ \# \ M1, \ M2) \in \text{set } (\text{get-all-ann-decomposition } (\text{conc-trail } S)) \implies$

$\text{get-level } (\text{conc-trail } S) \ L = \text{conc-backtrack-lvl } S \implies$

$\text{get-level } (\text{conc-trail } S) \ L = \text{get-maximum-level } (\text{conc-trail } S) \ (\text{mset-ccls } D) \implies$

$\text{get-maximum-level } (\text{conc-trail } S) \ (\text{mset-ccls } D - \{\#L\# \}) \equiv i \implies$

$\text{get-level } (\text{conc-trail } S) \ K = i + 1 \implies$

$T \sim \text{cons-conc-trail } (\text{Propagated } L \ (\text{cls-of-ccls } D))$

$(\text{reduce-conc-trail-to } M1$

$(\text{add-conc-conflict-to-learned-cls}$

$(\text{update-conc-backtrack-lvl } i \ S))) \implies$

backtrack-abs $S \ T$

inductive-cases *backtrack-absE*: *backtrack-abs* $S \ T$

lemma *backtrack-backtrack-abs*:

assumes *inv*: $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv } (\text{state } S)$

shows $\text{cdcl}_W\text{-restart-mset.backtrack } (\text{state } S) \ (\text{state } T) \longleftrightarrow \text{backtrack-abs } S \ T$ (**is** $?conc \longleftrightarrow ?abs$)

proof

assume $?abs$

then obtain $D \ L \ K \ M1 \ M2 \ i$ **where**

D : *raw-conc-conflicting* $S = \text{Some } D$ **and**

L: $L \in \# \text{ mset-clcs } D$ **and**
decomp: $(Decided\ K \ \# \ M1, \ M2) \in \text{set } (get\text{-all-ann-decomposition } (conc\text{-trail } S))$ **and**
lev-L: $get\text{-level } (conc\text{-trail } S) \ L = conc\text{-backtrack-lvl } S$ **and**
lev-Max: $get\text{-level } (conc\text{-trail } S) \ L = get\text{-maximum-level } (conc\text{-trail } S) \ (\text{mset-clcs } D)$ **and**
i: $get\text{-maximum-level } (conc\text{-trail } S) \ (\text{mset-clcs } D - \{\#L\# \}) \equiv i$ **and**
lev-K: $get\text{-level } (conc\text{-trail } S) \ K = i + 1$ **and**
T: $T \sim cons\text{-conc-trail } (Propagated\ L \ (cls\text{-of-clcs } D))$
 $(reduce\text{-conc-trail-to } M1$
 $(add\text{-conc-conflict-to-learned-clcs}$
 $(update\text{-conc-backtrack-lvl } i \ S)))$
by $(auto\ elim!: backtrack\text{-absE})$
have $n\text{-d: no-dup } (trail \ (state \ S))$
using $lev\text{-L inv unfolding } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv-def } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-M-lev}$
by $simp$
have $atm\text{-of } L \notin atm\text{-of } 'lits\text{-of-l } M1$
apply $(rule \ cdcl_W\text{-restart-mset.backtrack-lit-skipped}[of \ - \ state \ S])$
using $lev\text{-L inv unfolding } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv-def } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-M-lev}$
apply $simp$
using $decomp \text{ apply } simp$
using $lev\text{-L inv unfolding } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv-def } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-M-lev}$
apply $simp$
using $lev\text{-L inv unfolding } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv-def } cdcl_W\text{-restart-mset.cdcl}_W\text{-restart-M-lev}$
apply $simp$
using $lev\text{-K apply } simp$
done
then have $undef: undefined\text{-lit } M1 \ L$
by $(auto \ simp \ add: defined\text{-lit-map } lits\text{-of-def})$
obtain c **where** $tr: conc\text{-trail } S = c \ @ \ M2 \ @ \ Decided\ K \ \# \ M1$
using $decomp$ **by** $auto$
show $?conc$
apply $(rule \ cdcl_W\text{-restart-mset.backtrack-rule})$
using D **apply** $simp$
using L **apply** $simp$
using $decomp$ **apply** $simp$
using $lev\text{-L}$ **apply** $simp$
using $lev\text{-Max}$ **apply** $simp$
using i **apply** $simp$
using $lev\text{-K}$ **apply** $simp$
using $T \text{ undef } n\text{-d } tr \ D$ **unfolding** $cdcl_W\text{-restart-mset.state-eq-def}$
by $auto$
next
assume $?conc$
then obtain $L \ D \ K \ M1 \ M2 \ i$ **where**
 $conflict: conflicting \ (state \ S) = Some \ D$ **and**
L: $L \in \# \ D$ **and**
decomp: $(Decided\ K \ \# \ M1, \ M2) \in \text{set } (get\text{-all-ann-decomposition } (trail \ (state \ S)))$ **and**
lev-L: $get\text{-level } (trail \ (state \ S)) \ L = backtrack\text{-lvl } (state \ S)$ **and**
lev-max: $get\text{-level } (trail \ (state \ S)) \ L = get\text{-maximum-level } (trail \ (state \ S)) \ (D)$ **and**
i: $get\text{-maximum-level } (trail \ (state \ S)) \ (D - \{\#L\# \}) \equiv i$ **and**
lev-K: $get\text{-level } (trail \ (state \ S)) \ K = i + 1$ **and**
T: $state \ T \sim_m cons\text{-trail } (Propagated\ L \ (D))$
 $(cdcl_W\text{-restart-mset.reduce-trail-to } M1$
 $(add\text{-learned-clcs } D$
 $(update\text{-backtrack-lvl } i$
 $(update\text{-conflicting } None \ (state \ S))))$
by $(auto \ elim: cdcl_W\text{-restart-mset.backtrackE})$

obtain D' **where**
confl': raw-conc-conflicting $S = \text{Some } D'$ **and** $D[\text{simp}]$: $D = \text{mset-ccls } D'$
using *confl* **by** *auto*
have $n\text{-d}$: *no-dup* (*trail* (*state* S))
using *lev-L inv unfolding* *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-def cdcl_W-restart-mset.cdcl_W-restart-M-lev*
by *simp*
have *atm-of* $L \notin \text{atm-of ' lits-of-l } M1$
apply (*rule cdcl_W-restart-mset.backtrack-lit-skipped[of - state S]*)
using *lev-L inv unfolding* *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-def cdcl_W-restart-mset.cdcl_W-restart-M-lev*
apply *simp*
using *decomp* **apply** *simp*
using *lev-L inv unfolding* *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-def cdcl_W-restart-mset.cdcl_W-restart-M-lev*
apply *simp*
using *lev-L inv unfolding* *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-def cdcl_W-restart-mset.cdcl_W-restart-M-lev*
apply *simp*
using *lev-K* **apply** *simp*
done
then have *undef*: *undefined-lit* $M1 L$
by (*auto simp add: defined-lit-map lits-of-def*)
show *?abs*
apply (*rule backtrack-abs-rule*)
using *confl'* **apply** *simp*
using L **apply** *simp*
using *decomp* **apply** *simp*
using *lev-L* **apply** *simp*
using *lev-max* **apply** *simp*
using i **apply** *simp*
using *lev-K* **apply** *simp*
using T *undef n-d decomp confl'* **by** *auto*
qed

lemma *backtrack-exists-backtrack-abs-step*:

assumes *bt*: *cdcl_W-restart-mset.backtrack* $S T$ **and** *inv*: *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv*
 S **and**

SS' : $S \sim_m \text{state } S'$

obtains U **where** *backtrack-abs* $S' U$ **and** $T \sim_m \text{state } U$

proof –

from *bt* **obtain** $L D K M1 M2 i$ **where**

confl: *conflicting* $S = \text{Some } D$ **and**

L : $L \in \# D$ **and**

decomp: (*Decided* $K \# M1, M2$) $\in \text{set } (\text{get-all-ann-decomposition } (\text{trail } S))$ **and**

lev-L: *get-level* (*trail* S) $L = \text{backtrack-lvl } S$ **and**

lev-max: *get-level* (*trail* S) $L = \text{get-maximum-level } (\text{trail } S) (D)$ **and**

i : *get-maximum-level* (*trail* S) $(D - \{\#L\# \}) \equiv i$ **and**

lev-K: *get-level* (*trail* S) $K = i + 1$ **and**

T : $T \sim_m \text{cons-trail } (\text{Propagated } L (D))$
(cdcl_W-restart-mset.reduce-trail-to $M1$
(add-learned-cls D
(update-backtrack-lvl i
(update-conflicting *None* $S)))$)

by (*auto elim: cdcl_W-restart-mset.backtrackE*)

obtain D' **where**

confl': raw-conc-conflicting $S' = \text{Some } D'$ **and** $D[\text{simp}]$: $D = \text{mset-ccls } D'$

using *confl* SS' **by** *auto*

have $n\text{-d}$: *no-dup* (*trail* (*state* S'))

using *lev-L inv* SS' **unfolding** *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-def cdcl_W-restart-mset.cdcl_W-restart-M-lev*

```

  by simp
have atm-of L  $\notin$  atm-of ' lits-of-l M1
  apply (rule cdclW-restart-mset.backtrack-lit-skipped[of - state S'])
    using lev-L inv SS' unfolding cdclW-restart-mset.cdclW-restart-all-struct-inv-def cdclW-restart-mset.cdclW-restart-M
      apply simp
      using decomp SS' apply simp
    using lev-L inv SS' unfolding cdclW-restart-mset.cdclW-restart-all-struct-inv-def cdclW-restart-mset.cdclW-restart-M
      apply simp
    using lev-L inv SS' unfolding cdclW-restart-mset.cdclW-restart-all-struct-inv-def cdclW-restart-mset.cdclW-restart-M
      apply simp
    using lev-K SS' apply simp
  done
then have undef: undefined-lit M1 L
  by (auto simp add: defined-lit-map lits-of-def)

let ?U = cons-conc-trail (Propagated L (cls-of-ccls D'))
  (reduce-conc-trail-to M1
    (add-conc-confl-to-learned-ccls
      (update-conc-backtrack-lvl i S')))
have backtrack-abs S' ?U
  apply (rule backtrack-abs-rule)
    using confl' apply simp
    using L apply simp
    using decomp SS' apply simp
    using lev-L SS' apply simp
    using lev-max SS' apply simp
    using i SS' apply simp
    using lev-K SS' apply simp
  using T undef n-d decomp by auto
moreover have T  $\sim_m$  state ?U
  using undef decomp T n-d SS'[unfolded cdclW-restart-mset-state-eq-eq] confl' by auto
ultimately show thesis using that[of ?U] by fast
qed

```

interpretation backtrack-abs: relation-relation-abs cdcl_W-restart-mset.backtrack backtrack-abs state
 cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv
 apply unfold-locales
 apply (simp add: backtrack-backtrack-abs)
 using backtrack-exists-backtrack-abs-step applymetis
 using cdcl_W-restart-mset.backtrack cdcl_W-restart-mset.bj cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-inv
 by blast

inductive decide-abs :: 'st \Rightarrow 'st \Rightarrow bool **for** S :: 'st **where**
 decide-abs-rule:
 conc-conflicting S = None \implies
 undefined-lit (conc-trail S) L \implies
 atm-of L \in atms-of-mm (conc-init-class S) \implies
 T \sim cons-conc-trail (Decided L) (incr-lvl S) \implies
 decide-abs S T

inductive-cases decide-absE: decide-abs S T

lemma decide-decide-abs:
 cdcl_W-restart-mset.decide (state S) (state T) \longleftrightarrow decide-abs S T
 by (auto elim!: cdcl_W-restart-mset.decideE decide-absE intro!: cdcl_W-restart-mset.decide-rule decide-abs-rule)

interpretation *decide-abs: relation-relation-abs cdcl_W-restart-mset.decide decide-abs state*
 $\lambda-. \text{True}$
apply *unfold-locales*
apply (*simp add: decide-decide-abs*)
apply (*metis (full-types) cdcl_W-restart-mset.decide.cases cdcl_W-restart-mset-state-eq-eq*
conc-trail-update-conc-backtrack-lvl decide-decide-abs
state-cons-cons-trail-cons-trail-decided trail-state-conc-trail update-backtrack-lvl-state)
using *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-inv cdcl_W-restart-mset.decide cdcl_W-restart-mset.other*
by *blast*

inductive *skip-abs :: 'st \Rightarrow 'st \Rightarrow bool for S :: 'st where*

skip-abs-rule:

conc-trail S = Propagated L C' # M \Rightarrow
raw-conc-conflicting S = Some E \Rightarrow
 $-L \notin \# \text{ mset-ccls } E \Rightarrow$
mset-ccls E $\neq \{\#\}$ \Rightarrow
T \sim tl-conc-trail S \Rightarrow
skip-abs S T

inductive-cases *skip-absE: skip-abs S T*

lemma *skip-skip-abs:*

cdcl_W-restart-mset.skip (state S) (state T) \longleftrightarrow skip-abs S T (is ?conc \longleftrightarrow ?abs)

proof

assume *?abs*

then show *?conc*

by (*auto elim!: skip-absE intro!: cdcl_W-restart-mset.skip-rule*)

next

assume *?conc*

then obtain *L C' E M where*

tr: trail (state S) = Propagated L C' # M and

confl: conflicting (state S) = Some E and

L: $-L \notin \# E$ and

E: E $\neq \{\#\}$ and

T: state T \sim_m tl-trail (state S)

by (*auto elim: cdcl_W-restart-mset.skipE*)

obtain *E' where*

confl': raw-conc-conflicting S = Some E' and [simp]: E = mset-ccls E'

using *confl* **by** *auto*

show *?abs*

apply (*rule skip-abs-rule*)

using *tr* **apply** *simp*

using *confl'* **apply** *simp*

using *L* **apply** *simp*

using *E* **apply** *simp*

using *T* **by** *simp*

qed

lemma *skip-exists-skip-abs:*

assumes *skip: cdcl_W-restart-mset.skip S T and SS': S \sim_m state S'*

obtains *U where skip-abs S' U and T \sim_m state U*

proof $-$

obtain *L C' E M where*

tr: trail S = Propagated L C' # M and

confl: conflicting S = Some E and

L: $-L \notin \# E$ and

$E: E \neq \{\#\}$ **and**
 $T: T \sim_m \text{tl-trail } S$
using *skip* **by** (*auto elim: cdcl_W-restart-mset.skipE*)
obtain E' **where**
 $\text{confl}' : \text{raw-conc-conflicting } S' = \text{Some } E' \text{ and } [\text{simp}]: E = \text{mset-ccls } E'$
using $\text{confl } SS'$ **by** *auto*
have $\text{skip-abs } S' (\text{tl-conc-trail } S')$
apply (*rule skip-abs-rule*)
using $tr SS'$ **apply** *simp*
using $\text{confl}' SS'$ **apply** *simp*
using $L SS'$ **apply** *simp*
using E **apply** *simp*
using T **by** *simp*
then show *?thesis*
using *that[$\text{of } \text{tl-conc-trail } S$] $T SS'$ [*unfolded cdcl_W-restart-mset-state-eq-eq*]]* **by** *auto*
qed

interpretation *skip-abs: relation-relation-abs cdcl_W-restart-mset.skip skip-abs state*
 $\lambda-. \text{True}$
apply *unfold-locales*
apply (*simp add: skip-skip-abs*)
using *skip-exists-skip-abs* **apply** *metis*
using *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-inv cdcl_W-restart-mset.skip cdcl_W-restart-mset.other*
by *blast*

inductive *resolve-abs :: 'st \Rightarrow 'st \Rightarrow bool* **for** $S :: 'st$ **where**
resolve-abs-rule: conc-trail $S \neq [] \implies$
hd-raw-conc-trail $S = \text{Propagated } L E \implies$
 $L \in \# \text{mset-cls } E \implies$
raw-conc-conflicting $S = \text{Some } D' \implies$
 $-L \in \# \text{mset-ccls } D' \implies$
get-maximum-level (conc-trail S) (remove1-mset ($-L$) (mset-ccls D')) = conc-backtrack-lvl $S \implies$
 $T \sim \text{resolve-conflicting } L E (\text{tl-conc-trail } S) \implies$
resolve-abs $S T$

inductive-cases *resolve-absE: resolve-abs $S T$*

lemma *resolve-resolve-abs:*

cdcl_W-restart-mset.resolve (state S) (state T) \longleftrightarrow resolve-abs $S T$ (is ?conc \longleftrightarrow ?abs)

proof

assume *?conc*

then obtain $L E D$ **where**

tr: trail (state S) $\neq []$ and

hd: cdcl_W-restart-mset.hd-trail (state S) = Propagated $L E$ and

LE: $L \in \# E$ and

confl: conflicting (state S) = Some D and

LD: $-L \in \# D$ and

lvl-max: get-maximum-level (trail (state S)) ((remove1-mset ($-L$) D)) = backtrack-lvl (state S) and

$T: \text{state } T \sim_m \text{update-conflicting (Some (cdcl_W-restart-mset.resolve-cls } L D E)) (\text{tl-trail (state } S))$

by (*auto elim!: cdcl_W-restart-mset.resolveE*)

obtain E' **where**

hd': hd-raw-conc-trail $S = \text{Propagated } L E'$ and

[simp]: $E = \text{mset-cls } E'$

apply (*cases hd-raw-conc-trail S*)

using *hd-raw-conc-trail[$\text{of } S$] tr hd* **by** *simp-all*

obtain D' **where**

```

  confl': raw-conc-conflicting  $S = \text{Some } D'$  and
  [simp]:  $D = \text{mset-ccls } D'$ 
  using confl by auto
show ?abs
  apply (rule resolve-abs-rule)
    using tr apply simp
    using hd' apply simp
    using LE apply simp
    using confl' apply simp
    using LD apply simp
    using lvl-max apply simp
    using T confl' LE LD by simp
next
  assume ?abs
  then show ?conc
    using hd-raw-conc-trail[of S] by (auto elim!: resolve-absE intro!: cdclW-restart-mset.resolve-rule)
qed

lemma resolve-exists-resolve-abs:
  assumes
    res: cdclW-restart-mset.resolve S T and
    SS':  $S \sim_m \text{state } S'$ 
  obtains U where resolve-abs S' U and  $T \sim_m \text{state } U$ 
proof -
  obtain L E D where
    tr: trail S  $\neq []$  and
    hd: cdclW-restart-mset.hd-trail S = Propagated L E and
    LE:  $L \in \# E$  and
    confl: conflicting S = Some D and
    LD:  $-L \in \# D$  and
    lvl-max: get-maximum-level (trail S) ((remove1-mset (-L) D)) = backtrack-lvl S and
    T:  $T \sim_m \text{update-conflicting } (\text{Some } (\text{cdcl}_W\text{-restart-mset.resolve-clc } L \ D \ E)) \ (\text{tl-trail } S)$ 
  using res
  by (auto elim!: cdclW-restart-mset.resolveE)
  obtain E' where
    hd': hd-raw-conc-trail S' = Propagated L E' and
    [simp]:  $E = \text{mset-clc } E'$ 
  apply (cases hd-raw-conc-trail S')
  using hd-raw-conc-trail[of S'] tr hd SS' by simp-all
  obtain D' where
    confl': raw-conc-conflicting S' = Some D' and
    [simp]:  $D = \text{mset-ccls } D'$ 
  using confl SS' by auto
  let ?U = resolve-conflicting L E' (tl-conc-trail S')
  have resolve-abs S' ?U
  apply (rule resolve-abs-rule)
    using tr SS' apply simp
    using hd' apply simp
    using LE apply simp
    using confl' apply simp
    using LD apply simp
    using lvl-max SS' apply simp
    using T by simp
  moreover have  $T \sim_m \text{state } ?U$ 
  using T SS' confl LE LD unfolding cdclW-restart-mset.state-eq-def by fastforce
  ultimately show thesis using that[of ?U] by fast

```

qed

interpretation *resolve-abs: relation-relation-abs cdcl_W-restart-mset.resolve resolve-abs state*
 $\lambda-. \text{ True}$
apply *unfold-locales*
apply (*simp add: resolve-resolve-abs*)
using *resolve-exists-resolve-abs apply metis*
using *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-inv cdcl_W-restart-mset.resolve cdcl_W-restart-mset.other*
by *blast*

inductive *restart :: 'st \Rightarrow 'st \Rightarrow bool for S :: 'st where*
restart: conc-conflicting S = None \implies
 $\neg \text{conc-trail } S \models_{\text{asm}} \text{conc-clauses } S \implies$
 $T \sim \text{restart-state } S \implies$
restart S T

inductive-cases *restartE: restart S T*

We add the condition $C \notin \# \text{conc-init-clss } S$, to maintain consistency even without the strategy.

inductive *forget :: 'st \Rightarrow 'st \Rightarrow bool where*
forget-rule:

conc-conflicting S = None \implies
 $C \notin \# \text{raw-conc-learned-clss } S \implies$
 $\neg (\text{conc-trail } S) \models_{\text{asm}} \text{clauses } S \implies$
 $\text{mset-cls } C \notin \text{set } (\text{get-all-mark-of-propagated } (\text{conc-trail } S)) \implies$
 $\text{mset-cls } C \notin \# \text{conc-init-clss } S \implies$
 $T \sim \text{remove-cls } C S \implies$
forget S T

inductive-cases *forgetE: forget S T*

inductive *cdcl_W-restart-abs-rf :: 'st \Rightarrow 'st \Rightarrow bool for S :: 'st where*
restart: restart-abs S T \implies cdcl_W-restart-abs-rf S T |
forget: forget-abs S T \implies cdcl_W-restart-abs-rf S T

inductive *cdcl_W-restart-abs-bj :: 'st \Rightarrow 'st \Rightarrow bool where*
skip: skip-abs S S' \implies cdcl_W-restart-abs-bj S S' |
resolve: resolve-abs S S' \implies cdcl_W-restart-abs-bj S S' |
backtrack: backtrack-abs S S' \implies cdcl_W-restart-abs-bj S S'

inductive-cases *cdcl_W-restart-abs-bjE: cdcl_W-restart-abs-bj S T*

lemma *cdcl_W-restart-abs-bj-cdcl_W-restart-abs-bj:*
 $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv } (\text{state } S) \implies$
 $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-bj } (\text{state } S) (\text{state } T) \longleftrightarrow \text{cdcl}_W\text{-restart-abs-bj } S T$
by (*auto simp: cdcl_W-restart-mset.cdcl_W-bj.simps cdcl_W-restart-abs-bj.simps*
backtrack-backtrack-abs skip-skip-abs resolve-resolve-abs)

interpretation *cdcl_W-restart-abs-bj: relation-relation-abs cdcl_W-restart-mset.cdcl_W-bj cdcl_W-restart-abs-bj*
state
 $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv}$
apply *unfold-locales*
apply (*simp add: cdcl_W-restart-abs-bj-cdcl_W-restart-abs-bj*)
apply (*metis (no-types, hide-lams) backtrack-exists-backtrack-abs-step cdcl_W-restart-abs-bj.simps*
cdcl_W-restart-mset.cdcl_W-bj.simps resolve-exists-resolve-abs skip-abs.relation-compatible-abs)
using *cdcl_W-restart-mset.bj cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-inv cdcl_W-restart-mset.other*

by *blast*

inductive *cdcl_W-restart-abs-o* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**
decide: *decide-abs* *S S'* \Rightarrow *cdcl_W-restart-abs-o* *S S'* |
bj: *cdcl_W-restart-abs-bj* *S S'* \Rightarrow *cdcl_W-restart-abs-o* *S S'*

inductive *cdcl_W-restart-abs* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**
propagate: *propagate-abs* *S S'* \Rightarrow *cdcl_W-restart-abs* *S S'* |
conflict: *conflict-abs* *S S'* \Rightarrow *cdcl_W-restart-abs* *S S'* |
other: *cdcl_W-restart-abs-o* *S S'* \Rightarrow *cdcl_W-restart-abs* *S S'* |
rf: *cdcl_W-restart-abs-rf* *S S'* \Rightarrow *cdcl_W-restart-abs* *S S'*

7.2.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have add a strategy and show the inclusion in the multiset version.

inductive *cdcl_W-merge-abs-cp* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**
conflict': *conflict-abs* *S T* \Rightarrow *full cdcl_W-restart-abs-bj* *T U* \Rightarrow *cdcl_W-merge-abs-cp* *S U* |
propagate': *propagate-abs⁺⁺* *S S'* \Rightarrow *cdcl_W-merge-abs-cp* *S S'*

lemma *cdcl_W-merge-cp-cdcl_W-restart-abs-merge-cp*:

assumes

cp: *cdcl_W-merge-abs-cp* *S T* **and**

inv: *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv* (state *S*)

shows *cdcl_W-restart-mset.cdcl_W-merge-cp* (state *S*) (state *T*)

using *cp*

proof (induction rule: *cdcl_W-merge-abs-cp.induct*)

case (*conflict'* *T U*) **note** *confl* = *this(1)* **and** *bj* = *this(2)*

then have *cdcl_W-restart-mset.conflict* (state *S*) (state *T*)

by (*auto simp: conflict-conflict-abs propagate-propagate-abs cdcl_W-restart-abs-bj-cdcl_W-restart-abs-bj*)

moreover

have *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv* (state *T*)

using *cdcl_W-restart-mset.conflict cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-inv confl inv*

unfolding *conflict-conflict-abs[symmetric]* **by** *blast*

then have *full cdcl_W-restart-mset.cdcl_W-bj* (state *T*) (state *U*)

using *bj* **by** (*auto simp: cdcl_W-restart-abs-bj.full-if-full-abs*)

ultimately show ?*case* **by** (*auto intro: cdcl_W-restart-mset.cdcl_W-merge-cp.intros*)

next

case (*propagate'* *T*)

then show ?*case*

by (*auto simp: propagate-abs.tranclp-abs-tranclp intro: cdcl_W-restart-mset.cdcl_W-merge-cp.propagate'*)

qed

lemma *cdcl_W-merge-cp-abs-exists-cdcl_W-merge-cp*:

assumes

cp: *cdcl_W-restart-mset.cdcl_W-merge-cp* (state *S*) *T* **and**

inv: *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv* (state *S*)

obtains *U* **where** *cdcl_W-merge-abs-cp* *S U* **and** *T* \sim_m state *U*

using *cp*

proof (induction rule: *cdcl_W-restart-mset.cdcl_W-merge-cp.induct*)

case (*conflict'* *T U*) **note** *confl* = *this(1)* **and** *bj* = *this(2)* **and** *that* = *this(3)*

obtain *V* **where** *SV: conflict-abs* *S V* **and** *TV: T* \sim_m state *V*

using *conflict-abs.relation-compatible-abs[of state S S]* *confl* **by** *blast*

have *inv-V: cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv* (state *V*) **and**

$inv-T: cdcl_W\text{-restart-mset}.cdcl_W\text{-restart-all-struct-inv } T$
using $TV\ bj\ cdcl_W\text{-restart-mset}.cdcl_W\text{-stgy.simps } cdcl_W\text{-restart-mset}.cdcl_W\text{-stgy-cdcl}_W\text{-restart-all-struct-inv}$
 $cdcl_W\text{-restart-mset}.conflict-is-full1-cdcl_W\text{-cp } confl\ inv$ **unfolding** $cdcl_W\text{-restart-mset-state-eq-eq}$ **by**
 $blast+$
then obtain T' **where** $full\ cdcl_W\text{-restart-abs-bj } V\ T'$ **and** $U \sim_m state\ T'$
using $TV\ bj\ cdcl_W\text{-restart-abs-bj}.full\ exists-full-abs[of\ V\ U]$ **unfolding** $cdcl_W\text{-restart-mset-state-eq-eq}$
by $blast$
then show $?thesis$ **using** $that\ cdcl_W\text{-merge-abs-cp}.conflict'[of\ S\ V\ T']\ SV$ **by** $fast$
next
case $(propagate'\ T)$
then show $?case$
using $cdcl_W\text{-merge-abs-cp}.propagate'$
 $propagate-abs.transcp-relation-transcp-relation-abs-compatible$ **by** $blast$
qed

lemma $no\text{-step-cdcl}_W\text{-merge-cp-no-step-cdcl}_W\text{-restart-abs-merge-cp}:$

assumes

$inv: cdcl_W\text{-restart-mset}.cdcl_W\text{-restart-all-struct-inv } (state\ S)$

shows $no\text{-step } cdcl_W\text{-merge-abs-cp } S \longleftrightarrow no\text{-step } cdcl_W\text{-restart-mset}.cdcl_W\text{-merge-cp } (state\ S)$

(**is** $?abs \longleftrightarrow ?conc$)

proof

assume $?abs$

show $?conc$

proof ($rule\ ccontr$)

assume $\neg ?thesis$

then obtain T **where** $cdcl_W\text{-restart-mset}.cdcl_W\text{-merge-cp } (state\ S)\ T$

by $blast$

then show $False$

using $cdcl_W\text{-merge-cp-abs-exists-cdcl}_W\text{-merge-cp}[of\ S\ T]\ (\?abs)\ inv$ **by** $auto$

qed

next

assume $?conc$

then show $?abs$

using $cdcl_W\text{-merge-cp-cdcl}_W\text{-restart-abs-merge-cp } inv$ **by** $blast$

qed

lemma $cdcl_W\text{-merge-abs-cp-right-compatible}:$

$cdcl_W\text{-merge-abs-cp } S\ V \implies cdcl_W\text{-restart-mset}.cdcl_W\text{-restart-all-struct-inv } (state\ S) \implies$

$V \sim W \implies cdcl_W\text{-merge-abs-cp } S\ W$

proof ($induction\ rule: cdcl_W\text{-merge-abs-cp}.induct$)

case $(conflict'\ T\ U)$ **note** $confl = this(1)$ **and** $full = this(2)$ **and** $inv = this(3)$ **and** $UW = this(4)$

have $inv-T: cdcl_W\text{-restart-mset}.cdcl_W\text{-restart-all-struct-inv } (state\ T)$

using $cdcl_W\text{-restart-mset}.cdcl_W\text{-stgy.simps } cdcl_W\text{-restart-mset}.cdcl_W\text{-stgy-cdcl}_W\text{-restart-all-struct-inv}$
 $cdcl_W\text{-restart-mset}.conflict-is-full1-cdcl_W\text{-cp } confl\ conflict-conflict-abs\ inv$ **by** $blast$

then have $full\ cdcl_W\text{-restart-abs-bj } T\ W \vee (T = U \wedge no\text{-step } cdcl_W\text{-restart-abs-bj } T)$

using $cdcl_W\text{-restart-abs-bj}.full\ right-compatible[OF\ -\ full\ UW]\ full$ **by** $blast$

then consider

$(full)\ full\ cdcl_W\text{-restart-abs-bj } T\ W \mid$

$(0)\ T = U$ **and** $no\text{-step } cdcl_W\text{-restart-abs-bj } T$

by $blast$

then show $?case$

proof $cases$

case $full$

then show $?thesis$ **using** $confl$ **by** ($blast\ intro: cdcl_W\text{-merge-abs-cp}.intros$)

next

case 0

then have *conflict-abs S W* **and** *no-step cdcl_W-restart-abs-bj W*
using *confl UW conflict-abs.relation-right-compatible* **apply** *blast*
using *full unfolding full-def*
by (*metis (mono-tags, lifting) 0(1) UW inv-T cdcl_W-restart-abs-bj-cdcl_W-restart-abs-bj*
cdcl_W-restart-mset-state-eq-eq)
moreover then have *full cdcl_W-restart-abs-bj W W*
unfolding *full-def* **by** *auto*
ultimately show *?thesis* **by** (*blast intro: cdcl_W-merge-abs-cp.intros*)
qed
next
case (*propagate'*)
then show *?case* **using** *propagate-abs.tranclp-relation-compatible-eq*
by (*blast intro: cdcl_W-merge-abs-cp.propagate'*)
qed

interpretation *cdcl_W-merge-abs-cp: relation-implied-relation-abs*
cdcl_W-restart-mset.cdcl_W-merge-cp cdcl_W-merge-abs-cp state cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv
apply *unfold-locales*
using *cdcl_W-merge-cp-cdcl_W-restart-abs-merge-cp* **apply** *blast*
using *cdcl_W-merge-cp-abs-exists-cdcl_W-merge-cp* **unfolding** *cdcl_W-restart-mset-state-eq-eq* **apply**
blast
using *cdcl_W-restart-mset.rtranclp-cdcl_W-restart-all-struct-inv-inv*
cdcl_W-restart-mset.rtranclp-cdcl_W-merge-cp-rtranclp-cdcl_W-restart **apply** *blast*
using *cdcl_W-merge-abs-cp-right-compatible* **unfolding** *cdcl_W-restart-mset-state-eq-eq* **by** *blast*

inductive *cdcl_W-merge-abs-stgy* **for** *S :: 'st* **where**
fw-s-cp: full1 cdcl_W-merge-abs-cp S T \implies cdcl_W-merge-abs-stgy S T |
fw-s-decide: decide-abs S T \implies no-step cdcl_W-merge-abs-cp S \implies full cdcl_W-merge-abs-cp T U
 \implies *cdcl_W-merge-abs-stgy S U*

lemma *cdcl_W-cp-cdcl_W-restart-abs-cp:*
assumes *stgy: cdcl_W-merge-abs-stgy S T* **and**
inv: cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv (state S)
shows *cdcl_W-restart-mset.cdcl_W-merge-stgy (state S) (state T)*
using *stgy*
proof (*induction rule: cdcl_W-merge-abs-stgy.induct*)
case (*fw-s-cp T*)
show *?case*
apply (*rule cdcl_W-restart-mset.cdcl_W-merge-stgy.fw-s-cp*)
using *fw-s-cp inv* **by** (*simp add: cdcl_W-merge-abs-cp.full1-iff*)
next
case (*fw-s-decide T U*) **note** *dec = this(1)* **and** *ns = this(2)* **and** *full = this(3)*
have *dec': cdcl_W-restart-mset.decide (state S) (state T)*
using *dec decide-decide-abs* **by** *blast*
then have *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv (state T)*
using *inv cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-inv*
by (*blast dest: cdcl_W-restart-mset.cdcl_W-restart.other cdcl_W-restart-mset.cdcl_W-o.decide*)
then have *full cdcl_W-restart-mset.cdcl_W-merge-cp (state T) (state U)*
using *full cdcl_W-merge-abs-cp.full-if-full-abs* **by** *blast*
then show *?case*
using *dec' cdcl_W-restart-mset.cdcl_W-merge-stgy.fw-s-decide[of state S state T state U]* *ns inv*
by (*simp add: no-step-cdcl_W-merge-cp-no-step-cdcl_W-restart-abs-merge-cp*)
qed

lemma *cdcl_W-merge-abs-stgy-exists-cdcl_W-merge-stgy:*

assumes
inv: $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv } S$ **and**
SS': $S \sim_m \text{state } S'$ **and**
st: $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-merge-stgy } S \ T$
shows $\exists U. \text{cdcl}_W\text{-merge-abs-stgy } S' \ U \wedge T \sim_m \text{state } U$
using *st*
proof (*induction rule*: $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-merge-stgy.induct}$)
case (*fw-s-cp* *T*)
then show ?*case* **using** $\text{cdcl}_W\text{-merge-abs-cp.full1-exists-full1-abs[of } S' \ T]$ *inv*
unfolding $SS'[\text{unfolded cdcl}_W\text{-restart-mset-state-eq-eq}]$ **by** (*metis* $\text{cdcl}_W\text{-merge-abs-stgy.fw-s-cp}$)
next
case (*fw-s-decide* *T U*) **note** $\text{dec} = \text{this}(1)$ **and** $n\text{-s} = \text{this}(2)$ **and** $\text{full} = \text{this}(3)$
have $SS': S = \text{state } S'$
using SS' **unfolding** $\text{cdcl}_W\text{-restart-mset-state-eq-eq}$.
obtain T' **where** $\text{decide-abs } S' \ T'$ **and** $TT': T \sim_m \text{state } T'$
using *dec* $\text{decide-abs.relation-compatible-abs[of } S \ S' \ T]$ SS' **by** *auto*
moreover
have $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv } (\text{state } T')$
using SS' *calculation*(1) $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart.intros}(3)$ $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv-i}$
 $\text{cdcl}_W\text{-restart-mset.decide decide-decide-abs inv}$ **by** *blast*
then obtain U' **where** $\text{full cdcl}_W\text{-merge-abs-cp } T' \ U'$ **and** $U \sim_m \text{state } U'$
using *full* $\text{cdcl}_W\text{-merge-abs-cp.full-exists-full-abs}$ **unfolding** $TT'[\text{unfolded cdcl}_W\text{-restart-mset-state-eq-eq}]$
by *blast*
moreover have $\text{no-step cdcl}_W\text{-merge-abs-cp } S'$
using $n\text{-s cdcl}_W\text{-merge-abs-cp.no-step-iff inv}$ **unfolding** SS' **by** *blast*
ultimately show ?*case*
using $\text{cdcl}_W\text{-merge-abs-stgy.fw-s-decide[of } S' \ T' \ U]$ **by** *fast*
qed

lemma $\text{cdcl}_W\text{-merge-abs-stgy-right-compatible}$:

assumes
inv: $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv } (\text{state } S)$ **and**
st: $\text{cdcl}_W\text{-merge-abs-stgy } S \ T$ **and**
TU: $T \sim V$
shows $\text{cdcl}_W\text{-merge-abs-stgy } S \ V$
using *st TU*
proof (*induction rule*: $\text{cdcl}_W\text{-merge-abs-stgy.induct}$)
case (*fw-s-cp* *T*)
then show ?*thesis*
using $\text{cdcl}_W\text{-merge-abs-cp.full1-right-compatible cdcl}_W\text{-merge-abs-stgy.fw-s-cp inv}$ **by** *blast*
next
case (*fw-s-decide* *T U*) **note** $\text{dec} = \text{this}(1)$ **and** $n\text{-s} = \text{this}(2)$ **and** $\text{full} = \text{this}(3)$ **and** $UV = \text{this}(4)$
have $\text{inv-T: cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv } (\text{state } T)$
using *dec inv* $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart-all-struct-inv-inv[of state } S \ \text{state } T]$
by (*auto* *dest*!: $\text{cdcl}_W\text{-restart-mset.cdcl}_W\text{-o.decide cdcl}_W\text{-restart-mset.cdcl}_W\text{-restart.other}$
 $\text{simp: decide-decide-abs[symmetric]}$)
then have $\text{full cdcl}_W\text{-merge-abs-cp } T \ V \vee (T = U \wedge \text{no-step cdcl}_W\text{-merge-abs-cp } T)$
using $\text{cdcl}_W\text{-merge-abs-cp.full-right-compatible[of } T \ U \ V]$ $\text{full } UV$ **by** *blast*
then consider
 $(\text{full}) \text{ full cdcl}_W\text{-merge-abs-cp } T \ V \mid$
 $(0) \ T = U$ **and** $\text{no-step cdcl}_W\text{-merge-abs-cp } T$
by *blast*
then show ?*case*
proof *cases*
case *full*
then show ?*thesis*

```

    using n-s dec by (blast intro: cdclW-merge-abs-stgy.intros)
next
case 0 note TU = this(1) and n-s' = this(2)
have decide-abs S V
  using TU dec UV decide-abs.relation-abs-right-compatible by auto
moreover
  have cdclW-restart-mset.cdclW-restart-all-struct-inv (state V)
    using inv-T by (metis (full-types) TU cdclW-restart-mset-state-eq-eq fw-s-decide.prems)
  then have full cdclW-merge-abs-cp V V
    using n-s' TU UV[unfolded cdclW-restart-mset-state-eq-eq]
    unfolding full-def by (metis cdclW-merge-abs-cp.no-step-iff rtranclp-unfold)
  ultimately show ?thesis using n-s by (blast intro: cdclW-merge-abs-stgy.intros)
qed
qed

interpretation cdclW-merge-abs-stgy: relation-implied-relation-abs
cdclW-restart-mset.cdclW-merge-stgy cdclW-merge-abs-stgy state cdclW-restart-mset.cdclW-restart-all-struct-inv
apply unfold-locales
  using cdclW-cp-cdclW-restart-abs-cp apply blast
  using cdclW-merge-abs-stgy-exists-cdclW-merge-stgy apply blast
using cdclW-restart-mset.cdclW-merge-stgy-rtranclp-cdclW-restart cdclW-restart-mset.rtranclp-cdclW-restart-all-struct
apply blast
using cdclW-merge-abs-stgy-right-compatible by blast

lemma cdclW-merge-abs-stgy-final-State-conclusive:
fixes T :: 'st
assumes
  full: full cdclW-merge-abs-stgy (conc-init-state N) T and
  n-d: distinct-mset-mset (mset-cls N)
shows (conc-conflicting T = Some {#} ∧ unsatisfiable (set-mset (mset-cls N)))
  ∨ (conc-conflicting T = None ∧ conc-trail T ⊨asm mset-cls N
    ∧ satisfiable (set-mset (mset-cls N)))
proof -
  have cdclW-restart-mset.cdclW-restart-all-struct-inv (state (conc-init-state N))
    using n-d unfolding cdclW-restart-mset.cdclW-restart-all-struct-inv-def by (auto simp: state-conc-init-state)
  then show ?thesis
    using cdclW-restart-mset.full-cdclW-merge-stgy-final-state-conclusive'[of mset-cls N state T]
    cdclW-merge-abs-stgy.full-if-full-abs[of conc-init-state N T] full
    by (auto simp: state-conc-init-state n-d)
qed

end

end

```

7.3 2-Watched-Literal

```

theory CDCL-Two-Watched-Literals
imports CDCL-W-Abstract-State
begin

```

First we define here the core of the two-watched literal data structure:

1. A clause is composed of (at most) two watched literals.

2. It is sufficient to find the candidates for propagation and conflict from the clauses such that the new literal is watched.

While this is the principle behind the two-watched literals, an implementation has to remember the candidates that have been found so far while updating the data structure.

We will directly on the two-watched literals data structure with lists: it could be also seen as a state over some abstract clause representation we would later refine as lists. However, as we need a way to select element from a clause, working on lists is better.

7.3.1 Essence of 2-WL

Data structure 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 literal list*) (*unwatched*: *'v literal list*)

datatype *'v twl-state* =

TWL-State (*raw-trail*: (*'v, 'v twl-clause*) *ann-lits*)
 (*raw-init-clss*: *'v twl-clause list*)
 (*raw-learned-clss*: *'v twl-clause list*) (*backtrack-lvl*: *nat*)
 (*raw-conflicting*: *'v literal list option*)

fun *mmset-of-mlit* :: (*'v, 'v twl-clause*) *ann-lit* \Rightarrow (*'v, 'v clause*) *ann-lit*

where

mmset-of-mlit (*Propagated L C*) = *Propagated L* (*mset* (*watched C @ unwatched C*)) |
mmset-of-mlit (*Decided L*) = *Decided L*

lemma *lit-of-mmset-of-mlit[simp]*: *lit-of* (*mmset-of-mlit x*) = *lit-of x*

by (*cases x*) *auto*

lemma *lits-of-mmset-of-mlit[simp]*: *lits-of* (*mmset-of-mlit ' S*) = *lits-of S*

by (*auto simp: lits-of-def image-image*)

abbreviation *trail* **where**

trail S \equiv *map mmset-of-mlit* (*raw-trail S*)

abbreviation *clauses-of-l* **where**

clauses-of-l \equiv $\lambda L. \text{mset } (\text{map mset } L)$

definition *raw-clause* :: *'v twl-clause* \Rightarrow *'v literal list* **where**

raw-clause C \equiv *watched C @ unwatched C*

definition *clause* :: *'v twl-clause* \Rightarrow *'v clause* **where**

clause C \equiv *mset* (*raw-clause C*)

lemma *clause-def-lambda*:

clause = ($\lambda C. \text{mset } (\text{raw-clause } C)$)

by (*auto simp: clause-def*)

abbreviation *raw-clss-l* :: *'a twl-clause list* \Rightarrow *'a clauses* **where**

raw-clss-l C \equiv *mset* (*map clause C*)

abbreviation *raw-clauses* :: 'v twl-state \Rightarrow 'v twl-clause list **where**
raw-clauses *S* \equiv *raw-init-clss* *S* @ *raw-learned-clss* *S*

abbreviation *raw-clss* :: 'v twl-state \Rightarrow 'v clauses **where**
raw-clss *S* \equiv *raw-clss-l* (*raw-clauses* *S*)

interpretation *raw-cl* clause .

lemma *mset-map-clause-remove1-cond*:
raw-clss-l (*remove1-cond* ($\lambda D.$ clause *D* = clause *a*) *Cs*) = *remove1-mset* (clause *a*) (*raw-clss-l* *Cs*)
apply (*induction* *Cs*)
apply *simp*
by (*auto simp*; *ac-simps remove1-mset-single-add raw-clause-def clause-def*)

interpretation *raw-clss*
clause
raw-clss-l
 $\lambda L C. L \in \text{set } C \text{ op } \#$
apply (*unfold-locales*)
using *mset-map-clause-remove1-cond* **by** (*auto simp*; *hd-map comp-def map-tl ac-simps raw-clause-def union-mset-list mset-map-mset-remove1-cond ex-mset clause-def-lambda*)

lemma *ex-mset-unwatched-watched*:

$\exists a. \text{mset } (\text{unwatched } a) + \text{mset } (\text{watched } a) = E$

proof –

obtain *e* **where** *mset* *e* = *E*

using *ex-mset* **by** *blast*

then have *mset* (*unwatched* (*TWL-Clause* [] *e*)) + *mset* (*watched* (*TWL-Clause* [] *e*)) = *E*

by *auto*

then show *?thesis* **by** *fast*

qed

abbreviation *conc-learned-clss* **where**

conc-learned-clss $\equiv \lambda S. \text{mset } (\text{map } \text{clause } (\text{raw-learned-clss } S))$

interpretation *twl*: *abs-state_W-ops*

clause

raw-clss-l

$\lambda L C. L \in \text{set } C \text{ op } \#$

mset

raw-clause $\lambda C. \text{TWL-Clause } [] C$

trail $\lambda S. \text{hd } (\text{raw-trail } S)$

$(\lambda S. \text{raw-init-clss } S @ \text{raw-learned-clss } S) \text{ backtrack-lvl raw-conflicting}$

conc-learned-clss

rewrites

twl.mmset-of-mlit = *mmset-of-mlit*

proof *goal-cases*

case 1

show *H*: *?case*

apply *unfold-locales* **apply** (*auto simp*; *hd-map comp-def map-tl ac-simps raw-clause-def mset-map-mset-remove1-cond ex-mset-unwatched-watched clause-def*)[3]

using *mset-map-mset-remove1-cond*

done

```

case 2
show ?case
  apply (rule ext)
  apply (rename-tac x)
  apply (case-tac x)
  apply (simp-all add: abs-stateW-ops.mset-of-mlit.simps[OF H] raw-clause-def clause-def)
done
qed

```

declare *CDCL-Two-Watched-Literals.twl.mset-ccls-ccls-of-cl*[*simp del*]

definition

candidates-propagate :: 'v twl-state \Rightarrow ('v literal \times 'v twl-clause) set
where
candidates-propagate *S* =
 $\{(L, C) \mid L \ C.$
 $C \in \text{set } (\text{raw-clauses } S) \wedge$
 $\text{set } (\text{watched } C) - (\text{uminus ' lits-of-l } (\text{trail } S)) = \{L\} \wedge$
 $\text{undefined-lit } (\text{raw-trail } S) \ L\}$

definition *candidates-conflict* :: 'v twl-state \Rightarrow 'v twl-clause set **where**

candidates-conflict *S* =
 $\{C. C \in \text{set } (\text{raw-clauses } S) \wedge$
 $\text{set } (\text{watched } C) \subseteq \text{uminus ' lits-of-l } (\text{raw-trail } S)\}$

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*

Invariants

The structural invariants states that there are at most two watched elements, that the watched literals are distinct, and that there are 2 watched literals if there are at least than two different literals in the full clauses.

primrec *struct-wf-tw-cl* :: 'v twl-clause \Rightarrow bool **where**

struct-wf-tw-cl (*TWL-Clause W UW*) \longleftrightarrow
 $\text{distinct } W \wedge \text{length } W \leq 2 \wedge (\text{length } W < 2 \longrightarrow \text{set } UW \subseteq \text{set } W)$

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*, *L* does not get swapped with a watched literal *L'* such that $-L'$ is in the trail. This corresponds to the laziness of the data structure.

Remark that *M* is a trail: literals at the end were the first to be added to the trail.

primrec *watched-only-lazy-updates* :: ('v, 'mark) ann-lits \Rightarrow

'v twl-clause \Rightarrow bool

where

watched-only-lazy-updates M (TWL-Clause W UW) \longleftrightarrow
 $(\forall L' \in \text{set } W. \forall L \in \text{set } UW.$
 $-L' \in \text{lits-of-l } M \longrightarrow -L \in \text{lits-of-l } M \longrightarrow L \notin \text{set } W \longrightarrow$
 $\text{index } (\text{map lit-of } M) (-L') \leq \text{index } (\text{map lit-of } M) (-L))$

If the negation of a watched literal is included in the trail, then the negation of every unwatched literals is also included in the trail. Otherwise, the data-structure has to be updated.

```

primrec watched-wf-twl-clb :: ('a, 'b) ann-lits  $\Rightarrow$  'a twl-clause  $\Rightarrow$ 
  bool where
  watched-wf-twl-clb M (TWL-Clause W UW)  $\longleftrightarrow$ 
    ( $\forall L \in \text{set } W. -L \in \text{lits-of-l } M \longrightarrow (\forall L' \in \text{set } UW. L' \notin \text{set } W \longrightarrow -L' \in \text{lits-of-l } M)$ )

```

Here are the invariant strictly related to the 2-WL data structure.

```

primrec wf-twl-clb :: ('v, 'mark) ann-lits  $\Rightarrow$  'v twl-clause  $\Rightarrow$  bool where
  wf-twl-clb M (TWL-Clause W UW)  $\longleftrightarrow$ 
    struct-wf-twl-clb (TWL-Clause W UW)  $\wedge$  watched-wf-twl-clb M (TWL-Clause W UW)  $\wedge$ 
    watched-only-lazy-updates M (TWL-Clause W UW)

```

lemma wf-twl-clb-annotation-independant:

assumes M: map lit-of M = map lit-of M'

shows wf-twl-clb M (TWL-Clause W UW) \longleftrightarrow wf-twl-clb M' (TWL-Clause W UW)

proof –

have lits-of-l M = lits-of-l 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-clb-wf-twl-clb-tl:

assumes wf: wf-twl-clb M C **and** n-d: no-dup M

shows wf-twl-clb (tl M) C

proof (cases M)

case Nil

then show ?thesis **using** wf

by (cases C) (simp add: wf-twl-clb.simps[of tl -])

next

case (Cons l M') **note** M = this(1)

obtain W UW **where** C: C = TWL-Clause W UW

by (cases C)

{ **fix** L L'

assume

LW: L \in set W **and**

LM: $-L \in \text{lits-of-l } M'$ **and**

L'UW: L' \in set UW **and**

L' \notin set W

then have

L'M: $-L' \in \text{lits-of-l } M$

using wf **by** (auto simp: C M)

have watched-only-lazy-updates M C

using wf **by** (auto simp: C)

then have

index (map lit-of M) ($-L$) \leq index (map lit-of M) ($-L'$)

using LM L'M L'UW LW $\langle L' \notin \text{set } W \rangle$ C M **unfolding** lits-of-def

by (fastforce simp: lits-of-def)

then have $-L' \in \text{lits-of-l } M'$

using $\langle L' \notin \text{set } W \rangle$ LW L'M **by** (auto simp: C M split: if-split-asm)

}

moreover

{

fix L' L

```

assume
   $L' \in \text{set } W$  and
   $L \in \text{set } UW$  and
   $L'M: - L' \in \text{lits-of-l } M'$  and
   $- L \in \text{lits-of-l } M'$  and
   $L \notin \text{set } W$ 
moreover
  have  $\text{lit-of } l \neq - L'$ 
  using  $n\text{-d unfolding } M$ 
    by (metis (no-types)  $L'M M$  Decided-Propagated-in-iff-in-lits-of-l defined-lit-map
      distinct.simps(2) list.simps(9) set-map)
  moreover have watched-only-lazy-updates  $M C$ 
    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: if-split-asm)
}
moreover have distinct  $W$  and  $\text{length } W \leq 2$  and  $(\text{length } W < 2 \longrightarrow \text{set } UW \subseteq \text{set } W)$ 
  using wf by (auto simp: C M)
ultimately show ?thesis by (auto simp add: M C)
qed

```

```

lemma wf-twl-cls-append:
assumes
   $n\text{-d: no-dup } (M' @ M)$  and
   $wf: wf\text{-twl-cls } (M' @ M) C$ 
shows  $wf\text{-twl-cls } M C$ 
using wf n-d apply (induction M')
  apply simp
using wf-twl-cls-wf-twl-cls-tl by fastforce

```

```

definition wf-twl-state :: 'v twl-state  $\Rightarrow$  bool where
  wf-twl-state  $S \longleftrightarrow$ 
     $(\forall C \in \text{set } (\text{raw-clauses } S). wf\text{-twl-cls } (\text{raw-trail } S) C) \wedge no\text{-dup } (\text{raw-trail } S)$ 

```

```

lemma wf-candidates-propagate-sound:
assumes wf: wf-twl-state S and
   $cand: (L, C) \in \text{candidates-propagate } S$ 
shows  $\text{raw-trail } S \models_{as} CNot (\text{mset } (\text{removeAll } L (\text{raw-clause } C))) \wedge \text{undefined-lit } (\text{raw-trail } S) L$ 
  (is ?Not  $\wedge$  ?undef)

```

proof

```

def  $M \equiv \text{raw-trail } S$ 
def  $N \equiv \text{raw-init-clss } S$ 
def  $U \equiv \text{raw-learned-clss } S$ 

```

```

note  $MNU\text{-defs } [simp] = M\text{-def } N\text{-def } U\text{-def}$ 

```

have *cw*:

```

   $C \in \text{set } (N @ U)$ 
   $\text{set } (\text{watched } C) - \text{uminus ' lits-of-l } M = \{L\}$ 
   $\text{undefined-lit } M L$ 
  using cand unfolding candidates-propagate-def MNU-defs by auto

```

```

obtain  $W UW$  where  $cw\text{-eq: } C = \text{TWL-Clause } W UW$ 
  by (cases C)

```

```

have  $l\text{-w: } L \in \text{set } W$ 

```

```

using cw(2) cw-eq by auto

have wf-c: wf-twl-cls M C
  using wf cw(1) unfolding wf-twl-state-def by simp

have w-nw:
  distinct W
  length W < 2  $\implies$  set UW  $\subseteq$  set W
   $\bigwedge L L'. L \in \text{set } W \implies \neg L \in \text{lits-of-l } M \implies L' \in \text{set } UW \implies L' \notin \text{set } W \implies \neg L' \in \text{lits-of-l } M$ 
  using wf-c unfolding cw-eq by (auto simp: image-image)

have  $\forall L' \in \text{set } (\text{raw-clause } C) - \{L\}. \neg L' \in \text{lits-of-l } M$ 
proof (cases length W < 2)
  case True
  moreover have size W  $\neq$  0
    using cw(2) cw-eq by auto
  ultimately have size W = 1
    by linarith
  then have w: W = [L]
    using l-w by (auto simp: length-list-Suc-0)
  from True have set UW  $\subseteq$  set W
    using w-nw(2) by blast
  then show ?thesis
    using w cw(1) cw-eq by (auto simp: raw-clause-def)
next
  case sz2: False
  show ?thesis
proof
  fix L'
  assume l': L'  $\in$  set (raw-clause C) - {L}
  have ex-la:  $\exists La. La \neq L \wedge La \in \text{set } W$ 
  proof (cases W)
    case w: Nil
    then show ?thesis
      using l-w by auto
  next
    case lb: (Cons Lb W')
    show ?thesis
    proof (cases W')
      case Nil
      then show ?thesis
        using lb sz2 by simp
    next
      case lc: (Cons Lc W'')
      then show ?thesis
        by (metis distinct-length-2-or-more lb list.set-intros(1) list.set-intros(2) w-nw(1))
  qed
qed
then obtain La where la: La  $\neq$  L La  $\in$  set W
  by blast
then have La  $\in$  uminus ' lits-of-l M
  using cw(2)[unfolded cw-eq, simplified, folded M-def]  $\langle La \in \text{set } W \rangle \langle La \neq L \rangle$  by auto
then have nla:  $\neg La \in \text{lits-of-l } M$ 
  by (auto simp: image-image)
then show  $\neg L' \in \text{lits-of-l } M$ 

```

```

proof –
  have f1:  $L' \in \text{set } (\text{raw-clause } C)$ 
    using l' by blast
  have f2:  $L' \notin \{L\}$ 
    using l' by fastforce
  have  $\bigwedge l. L. - (l::'a \text{ literal}) \in L \vee l \notin \text{uminus } 'L$ 
    by force
  then show ?thesis
    using cw(1) cw-eq w-nw(3) raw-clause-def by (metis DiffI Un-iff cw(2) f1 f2 la(2) nla
      set-append twl-clause.sel(1) twl-clause.sel(2))
  qed
qed
qed
then show ?Not
  unfolding true-annots-def by (auto simp: image-image Ball-def CNot-def)

show ?undef
  using cw(3) unfolding M-def by blast
qed

lemma wf-candidates-propagate-complete:
  assumes wf: wf-twl-state S and
    c-mem:  $C \in \text{set } (\text{raw-clauses } S)$  and
    l-mem:  $L \in \text{set } (\text{raw-clause } C)$  and
    unsat:  $\text{trail } S \models_{\text{as}} \text{CNot } (\text{mset-set } (\text{set } (\text{raw-clause } C) - \{L\}))$  and
    undef: undefined-lit (raw-trail S) L
  shows  $(L, C) \in \text{candidates-propagate } S$ 
proof –
  def M  $\equiv$  raw-trail S
  def N  $\equiv$  raw-init-clss S
  def U  $\equiv$  raw-learned-clss S

  note MNU-defs [simp] = M-def N-def U-def

  obtain W UW where cw-eq:  $C = \text{TWL-Clause } W \text{ UW}$ 
    by (cases C, blast)

  have wf-c: wf-twl-clss M C
    using wf c-mem unfolding wf-twl-state-def by simp

  have w-nw:
    distinct W
    length W < 2  $\implies \text{set } UW \subseteq \text{set } W$ 
     $\bigwedge L L'. L \in \text{set } W \implies -L \in \text{lits-of-l } M \implies L' \in \text{set } UW \implies L' \notin \text{set } W \implies -L' \in \text{lits-of-l } M$ 
    using wf-c unfolding cw-eq by (auto simp: image-image)

  have unit-set:  $\text{set } W - (\text{uminus } ' \text{lits-of-l } M) = \{L\}$  (is ?W = ?L)
proof
  show ?W  $\subseteq \{L\}$ 
  proof
    fix L'
    assume l':  $L' \in ?W$ 
    then have l'-mem-w:  $L' \in \text{set } W$ 
      by (simp add: in-diffD)
    have  $L' \notin \text{uminus } ' \text{lits-of-l } M$ 
      using l' by blast

```

```

then have  $\neg M \models_a \{\#-L'\# \}$ 
  by (auto simp: lits-of-def uminus-lit-swap image-image)
moreover have  $L' \in \text{set } (raw\_clause\ C)$ 
  using c-mem cw-eq l'-mem-w by (auto simp: raw-clause-def)
ultimately have  $L' = L$ 
  using unsat[unfolded CNot-def true-annots-def, simplified]
  unfolding M-def by fastforce
then show  $L' \in \{L\}$ 
  by simp
qed
next
show  $\{L\} \subseteq ?W$ 
proof clarify
  have  $L \in \text{set } W$ 
  proof (cases W)
    case Nil
    then show ?thesis
      using w-nw(2) cw-eq l-mem by (auto simp: raw-clause-def)
  next
    case (Cons La W')
    then show ?thesis
      proof (cases La = L)
        case True
        then show ?thesis
          using Cons by simp
      next
        case False
        have  $-La \in \text{lits-of-l } M$ 
          using False Cons cw-eq unsat[unfolded CNot-def true-annots-def, simplified]
          by (fastforce simp: raw-clause-def)
        then show ?thesis
          using Cons cw-eq l-mem undef w-nw(3)
          by (auto simp: Decided-Propagated-in-iff-in-lits-of-l raw-clause-def)
      qed
    qed
  moreover have  $L \notin \# \text{ mset-set } (uminus\ ' \text{ lits-of-l } M)$ 
    using undef by (auto simp: Decided-Propagated-in-iff-in-lits-of-l image-image)
  ultimately show  $L \in ?W$ 
    by simp
  qed
qed

show ?thesis
  unfolding candidates-propagate-def using unit-set undef c-mem unfolding cw-eq M-def
  by (auto simp: image-image cw-eq intro!: exI[of - C])
qed

lemma wf-candidates-conflict-sound:
  assumes wf: wf-twl-state S and
    cand:  $C \in \text{candidates-conflict } S$ 
  shows  $\text{trail } S \models_{as} CNot\ (clause\ C) \wedge C \in \text{set } (raw\_clauses\ S)$ 
proof
  def M  $\equiv$  raw-trail S
  def N  $\equiv$  raw-init-clss S
  def U  $\equiv$  raw-learned-clss S

```



```

note MNU-defs [simp] = M-def N-def U-def

have cw:
  C ∈ set (N @ U)
  set (watched C) ⊆ uminus ‘lits-of-l (trail S)
  using cand[unfolded candidates-conflict-def, simplified] by auto

obtain W UW where cw-eq: C = TWL-Clause W UW
  by (cases C, blast)

have wf-c: wf-twl-cls M C
  using wf cw(1) unfolding wf-twl-state-def by simp

have w-nw:
  distinct W
  length W < 2 ⇒ set UW ⊆ set W
   $\bigwedge L L'. L \in \text{set } W \Rightarrow -L \in \text{lits-of-l } M \Rightarrow L' \in \text{set } UW \Rightarrow L' \notin \text{set } W \Rightarrow -L' \in \text{lits-of-l } M$ 
  using wf-c unfolding cw-eq by (auto simp: image-image)

have  $\forall L \in \text{set } (\text{raw-clause } C). -L \in \text{lits-of-l } M$ 
proof (cases W)
  case Nil
  then have raw-clause C = []
    using cw(1) cw-eq w-nw(2) by (auto simp: raw-clause-def)
  then show ?thesis
    by simp
next
  case (Cons La W') note W' = this(1)
  show ?thesis
  proof
    fix L
    assume l: L ∈ set (raw-clause C)
    show  $-L \in \text{lits-of-l } M$ 
    proof (cases L ∈ set W)
      case True
      then show ?thesis
        using cw(2) cw-eq by fastforce
      next
      case False
      then show ?thesis
        using W' cw(2) cw-eq l w-nw(3) unfolding M-def raw-clause-def
        by (metis (no-types, lifting) UnE imageE list.set-intros(1)
          lits-of-mmset-of-mlit rev-subsetD set-append set-map twl-clause.sel(1)
          twl-clause.sel(2) uminus-of-uminus-id)
    qed
  qed
qed
then show trail S  $\models_{\text{as}} \text{CNot } (\text{clause } C)$ 
  unfolding CNot-def true-annots-def clause-def by auto

show C ∈ set (raw-clauses S)
  using cw by auto
qed

lemma wf-candidates-conflict-complete:
  assumes wf: wf-twl-state S and

```

```

    c-mem:  $C \in \text{set } (\text{raw-clauses } S)$  and
    unsat:  $\text{trail } S \models_{\text{as}} \text{CNot } (\text{clause } C)$ 
  shows  $C \in \text{candidates-conflict } S$ 
proof -
  def  $M \equiv \text{raw-trail } S$ 
  def  $N \equiv \text{twl.conc-init-clss } S$ 
  def  $U \equiv \text{conc-learned-clss } S$ 

  note MNU-defs [simp] =  $M\text{-def } N\text{-def } U\text{-def}$ 

  obtain  $W \ UW$  where  $\text{cw-eq}: C = \text{TWL-Clause } W \ UW$ 
  by (cases  $C$ , blast)

  have wf-c:  $\text{wf-twl-clss } M \ C$ 
  using wf c-mem unfolding wf-twl-state-def by simp

  have w-nw:
    distinct  $W$ 
    length  $W < 2 \implies \text{set } UW \subseteq \text{set } W$ 
     $\bigwedge L \ L'. L \in \text{set } W \implies -L \in \text{lits-of-l } M \implies L' \in \text{set } UW \implies L' \notin \text{set } W \implies -L' \in \text{lits-of-l } M$ 
  using wf-c unfolding cw-eq by (auto simp: image-image)

  have  $\bigwedge L. L \in \text{set } (\text{raw-clause } C) \implies -L \in \text{lits-of-l } M$ 
  unfolding M-def using unsat[unfolded CNot-def true-annots-def, simplified]
  by (auto simp: clause-def)
  then have  $\text{set } (\text{raw-clause } C) \subseteq \text{uminus ' lits-of-l } M$ 
  by (metis imageI subsetI uminus-of-uminus-id)
  then have  $\text{set } W \subseteq \text{uminus ' lits-of-l } M$ 
  using cw-eq by (auto simp: raw-clause-def)
  then have subset:  $\text{set } W \subseteq \text{uminus ' lits-of-l } M$ 
  by (simp add: w-nw(1))

  have  $W = \text{watched } C$ 
  using cw-eq twl-clause.sel(1) by simp
  then show ?thesis
  using MNU-defs c-mem subset candidates-conflict-def by blast
qed

typedef 'v wf-twl =  $\{S :: 'v \text{ twl-state. wf-twl-state } S\}$ 
morphisms rough-state-of-twl twl-of-rough-state
proof -
  have  $\text{TWL-State } ([:: ('v, 'v \text{ twl-clause}) \text{ ann-lits})$ 
     $[] [] 0 \text{ None} \in \{S :: 'v \text{ twl-state. wf-twl-state } S\}$ 
  by (auto simp: wf-twl-state-def)
  then show ?thesis by auto
qed

lemma [code abstype]:
   $\text{twl-of-rough-state } (\text{rough-state-of-twl } S) = S$ 
  by (fact CDCL-Two-Watched-Literals.wf-twl.rough-state-of-twl-inverse)

lemma wf-twl-state-rough-state-of-twl[simp]:  $\text{wf-twl-state } (\text{rough-state-of-twl } S)$ 
  using rough-state-of-twl by auto

abbreviation candidates-conflict-twl :: 'v wf-twl  $\Rightarrow$  'v twl-clause set where
  candidates-conflict-twl  $S \equiv \text{candidates-conflict } (\text{rough-state-of-twl } S)$ 

```

abbreviation *candidates-propagate-twl* :: 'v wf-twl \Rightarrow ('v literal \times 'v twl-clause) set **where**
candidates-propagate-twl *S* \equiv *candidates-propagate* (*rough-state-of-twl* *S*)

abbreviation *raw-trail-twl* :: 'a wf-twl \Rightarrow ('a, 'a twl-clause) ann-lits **where**
raw-trail-twl *S* \equiv *raw-trail* (*rough-state-of-twl* *S*)

abbreviation *trail-twl* :: 'a wf-twl \Rightarrow ('a, 'a literal multiset) ann-lits **where**
trail-twl *S* \equiv *trail* (*rough-state-of-twl* *S*)

abbreviation *raw-clauses-twl* :: 'a wf-twl \Rightarrow 'a twl-clause list **where**
raw-clauses-twl *S* \equiv *raw-clauses* (*rough-state-of-twl* *S*)

abbreviation *raw-init-clss-twl* :: 'a wf-twl \Rightarrow 'a twl-clause list **where**
raw-init-clss-twl *S* \equiv *raw-init-clss* (*rough-state-of-twl* *S*)

abbreviation *raw-learned-clss-twl* :: 'a wf-twl \Rightarrow 'a twl-clause list **where**
raw-learned-clss-twl *S* \equiv *raw-learned-clss* (*rough-state-of-twl* *S*)

abbreviation *conc-learned-clss-twl* :: 'a wf-twl \Rightarrow 'a clauses **where**
conc-learned-clss-twl *S* \equiv *conc-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 *raw-conflicting-twl* **where**
raw-conflicting-twl *S* \equiv *raw-conflicting* (*rough-state-of-twl* *S*)

lemma *wf-candidates-twl-conflict-complete*:

assumes

c-mem: *C* \in set (*raw-clauses-twl* *S*) **and**

unsat: *trail-twl* *S* \models_{as} CNot (*clause* *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 (*raw-trail* *S*) (*raw-init-clss* *S*) (*raw-learned-clss* *S*) *k* (*raw-conflicting* *S*)

abbreviation *update-conflicting* **where**

update-conflicting *C* *S* \equiv

TWL-State (*raw-trail* *S*) (*raw-init-clss* *S*) (*raw-learned-clss* *S*) (*backtrack-lvl* *S*) *C*

Abstract 2-WL

definition *tl-trail* **where**

tl-trail *S* =

TWL-State (*tl* (*raw-trail* *S*)) (*raw-init-clss* *S*) (*raw-learned-clss* *S*) (*backtrack-lvl* *S*)
(*raw-conflicting* *S*)

locale *abstract-twl* =

fixes

watch :: 'v twl-state \Rightarrow 'v literal list \Rightarrow 'v twl-clause **and**

rewatch :: 'v literal \Rightarrow 'v twl-state \Rightarrow

'v twl-clause \Rightarrow 'v twl-clause **and**

restart-learned :: 'v twl-state \Rightarrow 'v twl-clause list

assumes

clause-watch: $\text{no-dup } (\text{raw-trail } S) \implies \text{clause } (\text{watch } S \ C) = \text{mset } C$ **and**
wf-watch: $\text{no-dup } (\text{raw-trail } S) \implies \text{wf-twl-cl } (\text{raw-trail } S) (\text{watch } S \ C)$ **and**
clause-rewatch: $\text{clause } (\text{rewatch } L' \ S \ C') = \text{clause } C'$ **and**

wf-rewatch:

$\text{no-dup } (\text{raw-trail } S) \implies \text{undefined-lit } (\text{raw-trail } S) (\text{lit-of } L) \implies$
 $\text{wf-twl-cl } (\text{raw-trail } S) \ C' \implies$
 $\text{wf-twl-cl } (L \ \# \ \text{raw-trail } S) (\text{rewatch } (\text{lit-of } L) \ S \ C')$

and

restart-learned: $\text{mset } (\text{restart-learned } S) \subseteq \# \ \text{mset } (\text{raw-learned-clss } S)$ — We need *mset* and not *set* to take care of duplicates.

begin

definition

cons-trail :: $('v, 'v \ \text{twl-clause}) \text{ ann-lit} \Rightarrow 'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-state}$

where

cons-trail $L \ S =$
 $\text{TWL-State } (L \ \# \ \text{raw-trail } S) (\text{map } (\text{rewatch } (\text{lit-of } L) \ S) (\text{raw-init-clss } S))$
 $(\text{map } (\text{rewatch } (\text{lit-of } L) \ S) (\text{raw-learned-clss } S)) (\text{backtrack-lvl } S) (\text{raw-conflicting } S)$

definition

add-init-cl :: $'v \ \text{literal list} \Rightarrow 'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-state}$

where

add-init-cl $C \ S =$
 $\text{TWL-State } (\text{raw-trail } S) (\text{watch } S \ C \ \# \ \text{raw-init-clss } S) (\text{raw-learned-clss } S) (\text{backtrack-lvl } S)$
 $(\text{raw-conflicting } S)$

definition

add-learned-cl :: $'v \ \text{literal list} \Rightarrow 'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-state}$

where

add-learned-cl $C \ S =$
 $\text{TWL-State } (\text{raw-trail } S) (\text{raw-init-clss } S) (\text{watch } S \ C \ \# \ \text{raw-learned-clss } S) (\text{backtrack-lvl } S)$
 $(\text{raw-conflicting } S)$

definition

remove-cl :: $'v \ \text{literal list} \Rightarrow 'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-state}$

where

remove-cl $C \ S =$
 $\text{TWL-State } (\text{raw-trail } S)$
 $(\text{removeAll-cond } (\lambda D. \text{clause } D = \text{mset } C) (\text{raw-init-clss } S))$
 $(\text{removeAll-cond } (\lambda D. \text{clause } D = \text{mset } C) (\text{raw-learned-clss } S))$
 $(\text{backtrack-lvl } S)$
 $(\text{raw-conflicting } S)$

definition *init-state* :: $'v \ \text{literal list list} \Rightarrow 'v \ \text{twl-state}$ **where**

init-state $N = \text{fold add-init-cl } N \ (\text{TWL-State } [] \ [] \ 0 \ \text{None})$

lemma *unchanged-fold-add-init-cl*:

$\text{raw-trail } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = M$
 $\text{raw-learned-clss } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = U$
 $\text{backtrack-lvl } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = k$
 $\text{raw-conflicting } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = C$
by (*induct* Cs *arbitrary*: N) (*auto simp*: *add-init-cl-def*)

lemma *unchanged-init-state*[*simp*]:

$\text{raw-trail } (\text{init-state } N) = []$

$raw_learned_clss \ (init_state \ N) = []$
 $backtrack_lvl \ (init_state \ N) = 0$
 $raw_conflicting \ (init_state \ N) = None$
unfolding $init_state_def$ **by** $(rule \ unchanged_fold_add_init_cls)+$

lemma $conc_init_clss[simp]$:
 $twl.conc_init_clss \ (TWL_State \ M \ N \ U \ k \ C) = raw_clss_l \ N$
by $(simp \ add: \ twl.conc_init_clss_def)$

lemma $clauses_init_fold_add_init$:
 $no_dup \ M \implies$
 $twl.conc_init_clss \ (fold \ add_init_cls \ Cs \ (TWL_State \ M \ N \ U \ k \ C)) =$
 $clauses_of_l \ Cs + raw_clss_l \ N$
by $(induct \ Cs \ arbitrary: \ N) \ (auto \ simp: \ add_init_cls_def \ clause_watch \ comp_def \ ac_simps \ clause_def[symmetric])$

lemma $init_clss_init_state[simp]$: $twl.conc_init_clss \ (init_state \ N) = clauses_of_l \ N$
unfolding $init_state_def$ **by** $(subst \ clauses_init_fold_add_init) \ simp_all$

definition $restart'$ **where**
 $restart' \ S = TWL_State \ [] \ (raw_init_clss \ S) \ (restart_learned \ S) \ 0 \ None$

end

Instanciacion of the previous locale

definition $watch_nat :: 'v \ twl_state \Rightarrow 'v \ literal \ list \Rightarrow 'v \ twl_clause$ **where**
 $watch_nat \ S \ C =$
 $(let$
 $\quad C' = remdups \ C;$
 $\quad neg_not_assigned = filter \ (\lambda L. \ -L \notin lits_of_l \ (raw_trail \ S)) \ C';$
 $\quad neg_assigned_sorted_by_trail = filter \ (\lambda L. \ L \in set \ C) \ (map \ (\lambda L. \ -lit_of \ L) \ (raw_trail \ S));$
 $\quad W = take \ 2 \ (neg_not_assigned \ @ \ neg_assigned_sorted_by_trail);$
 $\quad UW = foldr \ remove1 \ W \ C$
 $\quad in \ TWL_Clause \ W \ UW)$

lemma $list_cases2$:
fixes $l :: 'a \ list$
assumes
 $\quad l = [] \implies P$ **and**
 $\quad \bigwedge x. \ l = [x] \implies P$ **and**
 $\quad \bigwedge x \ y \ xs. \ l = x \ # \ y \ # \ xs \implies P$
shows P
by $(metis \ assms \ list.collapse)$

lemma $filter_in_list_prop_verifiedD$:
assumes $[L \leftarrow P \ . \ Q \ L] = l$
shows $\forall x \in set \ l. \ x \in set \ P \wedge Q \ x$
using $assms$ **by** $auto$

lemma $no_dup_filter_diff$:
assumes $n_d: no_dup \ M$ **and** $H: [L \leftarrow map \ (\lambda L. \ - \ lit_of \ L) \ M. \ L \in set \ C] = l$
shows $distinct \ l$
unfolding $H[symmetric]$
apply $(rule \ distinct_filter)$
using n_d **by** $(induction \ M) \ auto$

lemma *watch-nat-lists-disjointD*:

assumes

$l: [L \leftarrow \text{remdups } C. - L \notin \text{lits-of-l (raw-trail } S)] = l$ **and**

$l': [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) \text{ (raw-trail } S) . L \in \text{set } C] = l'$

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 image-image*)

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 list* **and**

$S :: 'v$ *twl-state*

defines

$xs \equiv [L \leftarrow \text{remdups } C. - L \notin \text{lits-of-l (raw-trail } S)]$ **and**

$ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) \text{ (raw-trail } S) . L \in \text{set } C]$

assumes

n-d: *no-dup (raw-trail S)* **and**

Nil-Nil: $xs = [] \implies ys = [] \implies P$ **and**

Nil-single:

$\bigwedge a. xs = [] \implies ys = [a] \implies a \in \text{set } C \implies P$ **and**

Nil-other: $\bigwedge a \ b \ ys'. xs = [] \implies ys = a \# b \# ys' \implies a \neq b \implies P$ **and**

single-Nil: $\bigwedge a. xs = [a] \implies ys = [] \implies P$ **and**

single-other: $\bigwedge a \ b \ ys'. xs = [a] \implies ys = b \# ys' \implies a \neq b \implies P$ **and**

other: $\bigwedge a \ b \ xs'. xs = a \# b \# xs' \implies a \neq b \implies P$

shows P

proof –

note *xs-def[simp]* **and** *ys-def[simp]*

have *dist*: $\bigwedge P. \text{distinct } [L \leftarrow \text{remdups } C . P \ L]$

by *auto*

then have $H: \bigwedge a \ b \ P \ xs. [L \leftarrow \text{remdups } C . P \ L] = a \# b \# xs \implies a \neq b$

by (*metis distinct-length-2-or-more*)

show *?thesis*

apply (*cases* $[L \leftarrow \text{remdups } C. - L \notin \text{lits-of-l (raw-trail } S)]$

rule: list-cases2;

cases $[L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) \text{ (raw-trail } S) . L \in \text{set } C]$ *rule: list-cases2*)

using *Nil-Nil* **apply** *simp*

using *Nil-single* **apply** (*force dest: filter-in-list-prop-verifiedD*)

using *Nil-other no-dup-filter-diff[OF n-d, of C]*

apply *fastforce*

using *single-Nil* **apply** *simp*

using *single-other xs-def ys-def* **apply** (*metis list.set-intros(1) watch-nat-lists-disjointD*)

using *single-other unfolding xs-def ys-def* **apply** (*metis list.set-intros(1)*

watch-nat-lists-disjointD)

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

$C :: 'v$ *literal list* **and**

$S :: 'v$ *twl-state*

defines

$xs \equiv [L \leftarrow \text{remdups } C . - L \notin \text{lits-of-l (raw-trail } S)]$ **and**

$ys \equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) \text{ (raw-trail } S) . L \in \text{set } C]$

assumes

n-d: *no-dup* (*raw-trail* *S*) **and**
Nil-Nil: $xs = [] \implies ys = [] \implies P$ **and**
Nil-single:
 $\bigwedge a. xs = [] \implies ys = [a] \implies a \in \text{set } C \implies P$ **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 C P*]
Nil-Nil Nil-single Nil-other single-Nil single-other other
unfolding *xs-def*[*symmetric*] *ys-def*[*symmetric*] **by** *auto*

lemma *watch-nat-lists-set-union-witness*:

fixes

C :: 'v *literal list* **and**

S :: 'v *twl-state*

defines

xs $\equiv [L \leftarrow \text{remdups } C. - L \notin \text{lits-of-l } (\text{raw-trail } S)]$ **and**

ys $\equiv [L \leftarrow \text{map } (\lambda L. - \text{lit-of } L) (\text{raw-trail } S) . L \in \text{set } C]$

assumes *n-d*: *no-dup* (*raw-trail* *S*)

shows *set C* = *set xs* \cup *set ys*

using *n-d* **unfolding** *xs-def ys-def* **by** (*auto simp: lits-of-def comp-def uminus-lit-swap*)

lemma *mset-intersection-inclusion*: $A + (B - A) = B \longleftrightarrow A \subseteq \# B$

apply (*rule iffI*)

apply (*metis mset-le-add-left*)

by (*auto simp: ac-simps multiset-eq-iff subseteq-mset-def*)

lemma *clause-watch-nat*:

assumes *no-dup* (*raw-trail* *S*)

shows *clause* (*watch-nat S C*) = *mset 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 multiset-eq-iff raw-clause-def clause-def*)

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 *foldr-remove1-W-Nil*[*simp*]: *foldr remove1 W []* = *[]*

by (*induct W*) *auto*

lemma *image-lit-of-mmset-of-mlit*[*simp*]:

lit-of ' *mmset-of-mlit* ' *A* = *lit-of* ' *A*

unfolding *comp-def*

using [*simp-trace*]] **by** (*simp add: image-image comp-def*)

lemma *distinct-filter-eq*:

assumes *distinct xs*

```

shows  $[L \leftarrow xs. L = a] = (\text{if } a \in \text{set } xs \text{ then } [a] \text{ else } [])$ 
using assms by (induction xs) auto

lemma no-dup-distinct-map-uminus-lit-of:
  no-dup xs  $\implies$  distinct (map ( $\lambda L. - \text{lit-of } L$ ) xs)
by (induction xs) auto

lemma wf-watch-witness:
  fixes C :: 'v literal list and
    S :: 'v twl-state
  defines
    ass: neg-not-assigned  $\equiv \text{filter } (\lambda L. -L \notin \text{lits-of-l (raw-trail } S)) (\text{remdups } C)$  and
    tr: neg-assigned-sorted-by-trail  $\equiv \text{filter } (\lambda L. L \in \text{set } C) (\text{map } (\lambda L. -\text{lit-of } L) (\text{raw-trail } S))$ 
  defines
    W: W  $\equiv \text{take } 2 (\text{neg-not-assigned } @ \text{neg-assigned-sorted-by-trail})$ 
assumes
  n-d[simp]: no-dup (raw-trail S)
shows wf-twlccls (raw-trail S) (TWL-Clause W (foldr remove1 W C))
unfolding wf-twlccls.simps struct-wf-twlccls.simps
proof (intro conjI, goal-cases)
  case 1
  then show ?case using n-d W unfolding ass tr
    apply (cases rule: watch-nat-list-cases-witness[of S C, OF n-d])
    by (auto simp: distinct-mset-add-single)
next
  case 2
  then show ?case unfolding W by simp
next
  case 3
  show ?case using n-d
    proof (cases rule: watch-nat-list-cases-witness[of S C])
      case Nil-Nil
      then have set C = set []  $\cup$  set []
        using watch-nat-lists-set-union-witness n-d by metis
      then show ?thesis
        by simp
    next
      case (Nil-single a)
      moreover have  $\bigwedge x. \text{set } C = \{a\} \implies - a \in \text{lits-of-l (trail } S) \implies x \in \text{set (remove1 } a \text{ } C) \implies x = a$ 
        using notin-set-remove1 by auto
      ultimately show ?thesis
        using watch-nat-lists-set-union-witness[of S C] 3 by (auto simp: W ass tr comp-def)
    next
      case Nil-other
      then show ?thesis
        using 3 by (auto simp: W ass tr)
    next
      case (single-Nil a)
      show ?thesis
        using watch-nat-lists-set-union-witness[of S C] 3
        by (fastforce simp add: W ass tr single-Nil comp-def distinct-filter-eq no-dup-distinct-map-uminus-lit-of min-def)
    next
      case single-other
      then show ?thesis

```



```

    using 3 by (auto simp: W ass tr)
next
  case other
  then show ?thesis
    using 3 by (auto simp: W ass tr)
qed
next
case 4 note -[simp] = this
show ?case
  using n-d apply (cases rule: watch-nat-list-cases-witness[of S C])
  apply (auto dest: filter-in-list-prop-verifiedD
    simp: W ass tr lits-of-def filter-empty-conv)[4]
  using watch-nat-lists-set-union-witness[of S C]
  by (force dest: filter-in-list-prop-verifiedD simp: W ass tr lits-of-def)+
next
case 5
from n-d show ?case
proof (cases rule: watch-nat-list-cases-witness[of S 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] tr by (fastforce simp: W ass)
next
  case Nil-other
  then show ?thesis
    unfolding watched-only-lazy-updates.simps Ball-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 - -  $\lambda L. L \in \text{set } C$ ])
    by (auto dest: filter-in-list-prop-verifiedD
      simp: uminus-lit-swap lits-of-def o-def W ass tr dest: in-diffD)
next
  case single-Nil
  then show ?thesis
    using watch-nat-lists-set-union-witness[of S C] tr by (fastforce simp: W ass)
next
  case single-other
  then show ?thesis
    unfolding watched-only-lazy-updates.simps Ball-def
    apply (clarify)
    apply (subst index-uminus-index-map-uminus,
      simp add: index-uminus-index-map-uminus lits-of-def image-image 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 - -  $\lambda L. L \in \text{set } C$ ])
    by (auto dest: filter-in-list-prop-verifiedD
      simp: W ass tr uminus-lit-swap lits-of-def o-def dest: in-diffD)
next
  case other

```

```

then show ?thesis
  unfolding watched-only-lazy-updates.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 - -  $\lambda L. L \in \text{set } C$ ])
  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 (raw-trail S)  $\implies$  wf-twl-cls (raw-trail S) (watch-nat S C)
  using wf-watch-witness[of S C] watch-nat-def by metis

definition
  rewatch-nat ::
    'v literal  $\Rightarrow$  'v twl-state  $\Rightarrow$  'v twl-clause  $\Rightarrow$  'v twl-clause
where
  rewatch-nat L S C =
    (if  $\neg L \in \text{set } (\text{watched } C)$  then
      case filter ( $\lambda L'. L' \notin \text{set } (\text{watched } C) \wedge \neg L' \in \text{insert } L (\text{lits-of-l } (\text{trail } S))$ )
        (unwatched C) of
        []  $\Rightarrow$  C
      | L' # -  $\Rightarrow$ 
        TWL-Clause (L' # remove1 ( $\neg L$ ) (watched C)) ( $\neg L$  # remove1 L' (unwatched C))
    else
      C)

lemma clause-rewatch-nat:
  fixes UW :: 'v literal list and
    S :: 'v twl-state and
    L :: 'v literal and C :: 'v twl-clause
  shows clause (rewatch-nat L S C) = clause C
  using List.set-remove1-subset[of  $\neg L$  watched C]
  apply (cases C)
  by (auto simp: raw-clause-def rewatch-nat-def ac-simps multiset-eq-iff clause-def
    split: list.split
    dest: filter-in-list-prop-verifiedD)

lemma filter-sorted-list-of-multiset-Nil:
   $[x \leftarrow \text{sorted-list-of-multiset } M. p \ x] = [] \iff (\forall x \in \# M. \neg p \ x)$ 
  by auto (metis empty-iff filter-set list.set(1) 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:
   $a - \{\#b\} = \{\#\} \iff a = \{\#b\} \vee a = \{\#\}$ 
  by (metis Multiset.diff-cancel add.right-neutral diff-single-eq-union
    diff-single-trivial zero-diff)

```

```

lemma size-mset-le-2-cases:
  assumes size  $W \leq 2$ 
  shows  $W = \{\#\} \vee (\exists a. W = \{\#a\#\}) \vee (\exists a\ b. W = \{\#a,b\#\})$ 
proof -
  have size  $W = 0 \vee \text{size } W = 1 \vee \text{size } W = 2$ 
    using assms by linarith
  then show ?thesis
    using assms by (fastforce elim!: size-mset-SucE simp: Num.numeral-2-eq-2)
qed

lemma filter-sorted-list-of-multiset-eqD:
  assumes  $[x \leftarrow \text{sorted-list-of-multiset } A. p\ x] = x \# xs$  (is ?comp = -)
  shows  $x \in \# A$ 
proof -
  have  $x \in \text{set } ?comp$ 
    using assms by simp
  then have  $x \in \text{set } (\text{sorted-list-of-multiset } A)$ 
    by simp
  then show  $x \in \# A$ 
    by simp
qed

lemma clause-rewatch-witness':
  assumes
    wf: wf-twl-cls (raw-trail S) C and
    undef: undefined-lit (raw-trail S) (lit-of L)
  shows wf-twl-cls (L  $\#$  raw-trail S) (rewatch-nat (lit-of L) S C)
proof (cases - lit-of L  $\in$  set (watched C))
  case False
  then show ?thesis
    apply (cases C)
    using wf undef unfolding rewatch-nat-def
    by (auto simp: uminus-lit-swap Decided-Propagated-in-iff-in-lits-of-l comp-def)
next
  case falsified: True

  let ?unwatched-nonfalsified =
     $[L' \leftarrow \text{unwatched } C. L' \notin \text{set } (\text{watched } C) \wedge - L' \notin \text{insert } (\text{lit-of } L) (\text{lits-of-l } (\text{trail } S))]$ 
  obtain W UW where C: C = TWL-Clause W UW
    by (cases C)

  show ?thesis
proof (cases ?unwatched-nonfalsified)
  case Nil
  show ?thesis
    using falsified Nil
    apply (simp only: wf-twl-cl.simps if-True list.cases C rewatch-nat-def struct-wf-twl-cl.simps)
    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  $\bigwedge p l. \text{filter } p (\text{unwatched } C) \neq [] \vee l \notin \text{set } UW \vee \neg p l$ 
    unfolding C by (metis (no-types) filter-empty-conv twl-clause.sel(2))
  then show ?case
    using 4(2) C by auto
next
  case 5
  then show ?case
    using wf by (fastforce simp add: C comp-def uminus-lit-swap)
qed
next
case (Cons L' Ls)
show ?thesis
  unfolding rewatch-nat-def
  using falsified Cons
  apply (simp only: wf-twl-cls.simps if-True list.cases C struct-wf-twl-cls.simps)
  apply (intro conjI)
  proof goal-cases
    case 1
    have distinct (watched (TWL-Clause W UW))
      using wf unfolding C by auto
    moreover have  $L' \notin \text{set } (\text{remove1 } (-\text{lit-of } L) (\text{watched } (TWL-Clause W UW)))$ 
      using 1(2) not-gr0 by (fastforce dest: filter-in-list-prop-verifiedD in-diffD)
    ultimately show ?case
      by (auto simp: distinct-mset-single-add)
  next
    case 2
    have f2:  $[l \leftarrow \text{unwatched } (TWL-Clause W UW) . l \notin \text{set } (\text{watched } (TWL-Clause W UW)) \wedge -l \notin \text{insert } (\text{lit-of } L) (\text{lits-of-l } (\text{trail } S))] \neq []$ 
      using 2(2) by simp
    then have  $\neg \text{set } UW \subseteq \text{set } W$ 
      using 2 by (auto simp add: filter-empty-conv)
    then show ?case
      using wf C 2(1) by (auto simp: length-remove1)
  next
    case 3
    have  $W: \text{length } W \leq \text{Suc } 0 \longleftrightarrow \text{length } W = 0 \vee \text{length } W = \text{Suc } 0$ 
      by linarith
    show ?case
      using wf C 3 by (auto simp: length-remove1 W length-list-Suc-0 dest!: subset-singletonD)
  next
    case 4
    have H:  $\forall L \in \text{set } W. -L \in \text{lits-of-l } (\text{trail } S) \longrightarrow (\forall L' \in \text{set } UW. L' \notin \text{set } W \longrightarrow -L' \in \text{lits-of-l } (\text{trail } S))$ 
      using wf by (auto simp: C)
    have  $W: \text{length } W \leq 2$  and  $W-UW: \text{length } W < 2 \longrightarrow \text{set } UW \subseteq \text{set } W$ 
      using wf by (auto simp: C)
    have distinct: distinct W
      using wf by (auto simp: C)
    show ?case
      using 4
      unfolding C watched-only-lazy-updates.simps Ball-def twl-clause.sel
        watched-wf-twl-cls.simps

```

```

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 distinct apply (fastforce)
  using distinct apply (fastforce)
  using distinct apply (fastforce)
  apply (force dest: filter-in-list-prop-verifiedD)
  using H by (auto simp: uminus-lit-swap)
next
case 5
have H:  $\forall x. x \in \text{set } W \longrightarrow - x \in \text{lits-of-l } (\text{trail } S) \longrightarrow (\forall x. x \in \text{set } UW \longrightarrow x \notin \text{set } W \longrightarrow - x \in \text{lits-of-l } (\text{trail } S))$ 
  using wf by (auto simp: C)
show ?case
  unfolding C watched-only-lazy-updates.simps Ball-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 = this
        have f9:  $L' \in \text{set } [l \leftarrow \text{unwatched } C. l \notin \text{set } (\text{watched } (TWL\text{-Clause } W UW)) \wedge - l \notin \text{lits-of-l } (L \# \text{raw-trail } S)]$ 
          using 1(2) 5 C by auto
        moreover then have f11:  $- xW \in \text{lits-of-l } (\text{trail } S)$ 
          using 1(3) LxW by (auto simp: uminus-lit-swap)
        moreover then have  $xW \notin \text{set } W$ 
          using f9 1(2) H by (auto simp: C)
        ultimately have False
          using 1 by auto
        then show ?thesis
          by fast
    qed
  qed
qed
qed
qed
qed

```

interpretation *twl*: *abstract-twl watch-nat rewatch-nat raw-learned-clss*
apply *unfold-locales*
apply (*rule clause-watch-nat; simp add: image-image comp-def*)
apply (*rule wf-watch-nat; simp add: image-image comp-def*)
apply (*rule clause-rewatch-nat*)
apply (*rule clause-rewatch-witness'; simp add: image-image comp-def*)
apply (*simp*)
done

interpretation *twl2*: *abstract-twl watch-nat rewatch-nat $\lambda\cdot$. []*
apply *unfold-locales*

```

apply (rule clause-watch-nat; simp add: image-image comp-def)
apply (rule wf-watch-nat; simp add: image-image comp-def)
apply (rule clause-rewatch-nat)
apply (rule clause-rewatch-witness'; simp add: image-image comp-def)
apply (simp)
done

```

end

7.3.2 Two Watched-Literals with invariant

```

theory CDCL-Two-Watched-Literals-Invariant
imports CDCL-Two-Watched-Literals DPLL-CDCL-W-Implementation
begin

```

Interpretation for *conflict-driven-clause-learning_W.cdcl_W-restart*

We define here the 2-WL with the invariant of well-foundedness and show the role of the candidates by defining an equivalent CDCL procedure using the candidates given by the data-structure.

```

context abstract-twl
begin

```

Direct Interpretation **lemma** *mset-map-removeAll-cond:*

```

mset (map clause
  (removeAll-cond (λD. clause D = clause C) N))
= mset (removeAll (clause C) (map clause N))
by (induction N) auto

```

lemma *raw-clss-l-raw-init-clss-conc-init-clss[simp]:*

```

raw-clss-l (raw-init-clss S) = twl.conc-init-clss S
by (cases S) auto

```

lemma *mset-raw-init-clss-init-state:*

```

raw-clss-l (raw-init-clss (init-state (map raw-clause N))) = raw-clss-l N
unfolding raw-clss-l-raw-init-clss-conc-init-clss
by (metis (full-types) clause-def comp-def init-clss-init-state map-eq-conv map-map)

```

fun *reduce-trail-to* **where**

```

reduce-trail-to M1 S =
  (case S of
    (TWL-State M N U k C) ⇒ TWL-State (drop (length M - length M1) M) N U k C)

```

abbreviation *resolve-conflicting* **where**

```

resolve-conflicting L D S ≡
  update-conflicting
  (Some (union-mset-list (remove1 (-L) (the (raw-conflicting S))) (remove1 L (raw-clause D))))
  S

```

interpretation *rough-cdcl: abs-state_W-ops*

```

clause
raw-clss-l
λL C. L ∈ set C op #

```

```

mset

```

```

raw-clause  $\lambda C. \text{TWL-Clause } [] \ C$ 
trail  $\lambda S. \text{hd } (\text{raw-trail } S)$ 
( $\lambda S. \text{raw-init-clss } S @ \text{raw-learned-clss } S$ ) backtrack-lvl raw-conflicting
conc-learned-clss

cons-trail tl-trail  $\lambda S. \text{update-conflicting None } (\text{add-learned-cls } (\text{the } (\text{raw-conflicting } S)) \ S)$ 
 $\lambda C. \text{remove-cls } (\text{raw-clause } C)$ 
update-backtrack-lvl
 $\lambda C. \text{update-conflicting } (\text{Some } C) \text{ reduce-trail-to resolve-conflicting}$ 
 $\lambda N. \text{init-state } (\text{map raw-clause } N) \text{ restart'}$ 
rewrites
rough-cdcl.mmset-of-mlit = mmset-of-mlit
proof goal-cases
case 1
show  $H: ?case$  by unfold-locales

case 2
show  $?case$ 
  apply (rule ext)
  apply (rename-tac  $x$ )
  apply (case-tac  $x$ )
  apply (simp-all add: abs-stateW-ops.mmset-of-mlit.simps[OF  $H$ ] raw-clause-def clause-def)
done
qed

interpretation rough-cdcl: abs-stateW
  clause
  raw-clss-l
   $\lambda L \ C. L \in \text{set } C \text{ op } \#$ 

  mset

raw-clause  $\lambda C. \text{TWL-Clause } [] \ C$ 
trail  $\lambda S. \text{hd } (\text{raw-trail } S)$ 
( $\lambda S. \text{raw-init-clss } S @ \text{raw-learned-clss } S$ ) backtrack-lvl raw-conflicting
conc-learned-clss

cons-trail tl-trail  $\lambda S. \text{update-conflicting None } (\text{add-learned-cls } (\text{the } (\text{raw-conflicting } S)) \ S)$ 
 $\lambda C. \text{remove-cls } (\text{raw-clause } C)$ 
update-backtrack-lvl
 $\lambda C. \text{update-conflicting } (\text{Some } C) \text{ reduce-trail-to resolve-conflicting}$ 
 $\lambda N. \text{init-state } (\text{map raw-clause } N) \text{ restart'}$ 
proof goal-cases
case 1
have stupid-locales: abs-stateW-ops clause raw-clss-l ( $\lambda L \ C. L \in \text{set } C$ ) op  $\#$ 
  mset raw-clause (TWL-Clause  $[]$ )
  by unfold-locales
have [simp]: abs-stateW-ops.mmset-of-mlit clause = mmset-of-mlit
  apply (rule ext, rename-tac  $L$ , case-tac  $L$ )
  by (auto simp: abs-stateW-ops.mmset-of-mlit.simps[OF stupid-locales] clause-def
    raw-clause-def)
have [simp]:  $\bigwedge S. \text{raw-clss-l } (\text{restart-learned } S) \subseteq \# \text{conc-learned-clss } S$ 
  using image-mset-subseteq-mono[OF restart-learned] unfolding mset-map
  by blast
have [simp]:  $\bigwedge st \ C. \text{mset } (\text{removeAll } (\text{clause } C) (\text{map clause } (\text{raw-init-clss } st))) =$ 
  removeAll-mset (clause  $C$ ) (rough-cdcl.conc-init-clss  $st$ )

```

unfolding *rough-cdcl.conc-init-clss-def* **by** (*metis mset-removeAll rough-cdcl.conc-init-clss-def twl2.raw-clss-l-raw-init-clss-conc-init-clss*)

have $H: \bigwedge M2\ M1\ x1. M2 @ M1 = \text{map } \text{mset-of-mlit } x1 \implies$
 $\text{map } \text{mset-of-mlit } (\text{drop } (\text{length } x1 - \text{length } M1) x1) = M1$
by (*metis add-diff-cancel-right' append-eq-conv-conj drop-map length-append length-map*)
show $H: ?\text{case}$
apply *unfold-locales*
apply (*case-tac raw-trail S; case-tac hd (raw-trail S)*)
by (*auto simp add: add-init-clss-def add-learned-clss-def clause-rewatch clause-watch*
cons-trail-def remove-clss-def restart'-def tl-trail-def map-tl comp-def
ac-simps mset-map-removeAll-cond mset-raw-init-clss-init-state rough-cdcl.state-def
clause-def[symmetric] H union-mset-list[symmetric] split: twl-state.splits)
qed

interpretation *rough-cdcl: abs-conflict-driven-clause-learning_w*

clause

raw-clss-l

$\lambda L\ C. L \in \text{set } C \text{ op } \#$

mset

raw-clause $\lambda C. \text{TWL-Clause } []\ C$

trail $\lambda S. \text{hd } (\text{raw-trail } S)$

$(\lambda S. \text{raw-init-clss } S @ \text{raw-learned-clss } S) \text{ backtrack-lvl raw-conflicting}$

conc-learned-clss

cons-trail *tl-trail* $\lambda S. \text{update-conflicting } \text{None } (\text{add-learned-clss } (\text{the } (\text{raw-conflicting } S))\ S)$

$\lambda C. \text{remove-clss } (\text{raw-clause } C)$

update-backtrack-lvl

$\lambda C. \text{update-conflicting } (\text{Some } C) \text{ reduce-trail-to resolve-conflicting}$

$\lambda N. \text{init-state } (\text{map raw-clause } N) \text{ restart}'$

by *unfold-locales*

declare *local.rough-cdcl.mset-ccls-ccls-of-clss[simp del]*

Opaque Type with Invariant **declare** *rough-cdcl.state-simp[simp del]*

definition *cons-trail-twl* $:: ('v, 'v\ \text{twl-clause}) \text{ann-lit} \Rightarrow 'v\ \text{wf-twl} \Rightarrow 'v\ \text{wf-twl}$

where

cons-trail-twl $L\ S \equiv \text{twl-of-rough-state } (\text{cons-trail } L\ (\text{rough-state-of-twl } S))$

lemma *wf-twl-state-cons-trail:*

assumes

undef: undefined-lit (raw-trail S) (lit-of L) and

wf: wf-twl-state S

shows *wf-twl-state (cons-trail L S)*

using *undef wf wf-rewatch[of S]* **unfolding** *wf-twl-state-def Ball-def*

by (*auto simp: cons-trail-def defined-lit-map comp-def image-def*

rough-cdcl.conc-clauses-init-learned)

lemma *rough-state-of-twl-cons-trail:*

undefined-lit (raw-trail-twl S) (lit-of L) \implies

rough-state-of-twl (cons-trail-twl L S) = cons-trail L (rough-state-of-twl S)

using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-cons-trail*

unfolding *cons-trail-twl-def* **by** *blast*

abbreviation *add-init-cls-twl* **where**

add-init-cls-twl $C\ S \equiv \text{twl-of-rough-state } (\text{add-init-cls } C\ (\text{rough-state-of-twl } S))$

lemma *wf-twl-add-init-cls*: $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{add-init-cls } L\ S)$

unfolding *wf-twl-state-def* **by** (*auto simp*: *wf-watch add-init-cls-def comp-def*
rough-cdcl.conc-clauses-init-learned
split: *if-split-asm*)

lemma *rough-state-of-twl-add-init-cls*:

rough-state-of-twl (*add-init-cls-twl* $L\ S$) = *add-init-cls* $L\ (\text{rough-state-of-twl } S)$

using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-init-cls* **by** *blast*

abbreviation *add-learned-cls-twl* **where**

add-learned-cls-twl $C\ S \equiv \text{twl-of-rough-state } (\text{add-learned-cls } C\ (\text{rough-state-of-twl } S))$

lemma *wf-twl-add-learned-cls*: $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{add-learned-cls } L\ S)$

unfolding *wf-twl-state-def* **by** (*auto simp*: *wf-watch add-learned-cls-def*
rough-cdcl.conc-clauses-init-learned
split: *if-split-asm*)

lemma *rough-state-of-twl-add-learned-cls*:

rough-state-of-twl (*add-learned-cls-twl* $L\ S$) = *add-learned-cls* $L\ (\text{rough-state-of-twl } S)$

using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-add-learned-cls* **by** *blast*

abbreviation *remove-cls-twl* **where**

remove-cls-twl $C\ S \equiv \text{twl-of-rough-state } (\text{remove-cls } C\ (\text{rough-state-of-twl } S))$

lemma *set-removeAll-condD*: $x \in \text{set } (\text{removeAll-cond } f\ xs) \implies x \in \text{set } xs$

by (*induction xs*) (*auto split*: *if-split-asm*)

lemma *wf-twl-remove-cls*: $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{remove-cls } L\ S)$

unfolding *wf-twl-state-def* **by** (*auto simp*: *wf-watch remove-cls-def*
comp-def split: *if-split-asm dest*: *set-removeAll-condD*)

lemma *rough-state-of-twl-remove-cls*:

rough-state-of-twl (*remove-cls-twl* $L\ S$) = *remove-cls* $L\ (\text{rough-state-of-twl } S)$

using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-remove-cls* **by** *blast*

abbreviation *init-state-twl* **where**

init-state-twl $N \equiv \text{twl-of-rough-state } (\text{init-state } N)$

lemma *wf-twl-state-wf-twl-state-fold-add-init-cls*:

assumes *wf-twl-state* S

shows *wf-twl-state* (*fold add-init-cls* $N\ S$)

using *assms apply* (*induction N arbitrary*: S)

apply (*auto simp*: *wf-twl-state-def*) \square

by (*simp add*: *wf-twl-add-init-cls*)

lemma *wf-twl-state-epsilon-state*[*simp*]:

wf-twl-state (*TWL-State* $\square\ \square\ \square\ 0\ \text{None}$)

by (*auto simp*: *wf-twl-state-def*)

lemma *wf-twl-init-state*: $\text{wf-twl-state } (\text{init-state } N)$

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 (*auto 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 comp-def)

lemma *rough-state-of-twl-tl-trail*:
rough-state-of-twl (tl-trail-twl S) = tl-trail (rough-state-of-twl S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-tl-trail* **by** *blast*

abbreviation *update-backtrack-lvl-twl* **where**
update-backtrack-lvl-twl k S \equiv *twl-of-rough-state (update-backtrack-lvl k (rough-state-of-twl S))*

lemma *wf-twl-state-update-backtrack-lvl*:
wf-twl-state S \implies wf-twl-state (update-backtrack-lvl k S)
unfolding *wf-twl-state-def* **by** (*auto simp: comp-def*)

lemma *rough-state-of-twl-update-backtrack-lvl*:
rough-state-of-twl (update-backtrack-lvl-twl k S) = update-backtrack-lvl k
(rough-state-of-twl S)
using *rough-state-of-twl*
twl-of-rough-state-inverse[of update-backtrack-lvl k (rough-state-of-twl S)]
wf-twl-state-update-backtrack-lvl[of rough-state-of-twl S k] **by** *fast*

abbreviation *update-conflicting-twl* **where**
update-conflicting-twl k S \equiv *twl-of-rough-state (update-conflicting k (rough-state-of-twl S))*

lemma *wf-twl-state-update-conflicting*:
wf-twl-state S \implies wf-twl-state (update-conflicting k S)
unfolding *wf-twl-state-def* **by** (*auto simp: comp-def*)

lemma *rough-state-of-twl-update-add-learned-cls*:
rough-state-of-twl (update-conflicting-twl None (add-learned-cls-twl C S)) =
update-conflicting None (add-learned-cls C (rough-state-of-twl S))
(is rough-state-of-twl ?upd = update-conflicting None ?le)
using *rough-state-of-twl[of ?upd] twl-of-rough-state-inverse*
wf-twl-add-learned-cls[of rough-state-of-twl S C]
wf-twl-state-update-conflicting[of ?le None]
by *fastforce*

abbreviation *reduce-trail-to-twl* **where**
reduce-trail-to-twl M1 S \equiv *twl-of-rough-state (reduce-trail-to M1 (rough-state-of-twl S))*

abbreviation *resolve-conflicting-twl* **where**
resolve-conflicting-twl L D S \equiv
twl-of-rough-state (resolve-conflicting L D (rough-state-of-twl S))

lemma *rough-state-of-twl-update-conflicting*:
rough-state-of-twl (update-conflicting-twl k S) = update-conflicting k
(rough-state-of-twl S)
using *rough-state-of-twl twl-of-rough-state-inverse wf-twl-state-update-conflicting* **by** *fast*

abbreviation *raw-clauses-twl* **where**

raw-clauses-twl $S \equiv \text{raw-clauses } (\text{rough-state-of-twl } S)$

abbreviation *restart-twl* **where**

restart-twl $S \equiv \text{twl-of-rough-state } (\text{restart}' (\text{rough-state-of-twl } S))$

lemma *mset-union-mset-setD*:

$\text{mset } A \subseteq\# \text{ mset } B \implies \text{set } A \subseteq \text{set } B$

by *auto*

lemma *wf-wf-restart'*: $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{restart}' S)$

unfolding *restart'-def* *wf-twl-state-def* **apply** *standard*

apply *clarify*

apply (*rename-tac* x)

apply (*subgoal-tac* *wf-twl-cls* (*raw-trail* S) x)

apply (*case-tac* x)

using *restart-learned* **by** (*auto simp: comp-def dest: mset-union-mset-setD*)

lemma *rough-state-of-twl-restart-twl*:

$\text{rough-state-of-twl } (\text{restart-twl } S) = \text{restart}' (\text{rough-state-of-twl } S)$

by (*simp add: twl-of-rough-state-inverse wf-wf-restart'*)

lemma *undefined-lit-trail-twl-raw-trail*[*iff*]:

$\text{undefined-lit } (\text{trail-twl } S) L \longleftrightarrow \text{undefined-lit } (\text{raw-trail-twl } S) L$

by (*auto simp: defined-lit-map image-image*)

lemma *wf-twl-reduce-trail-to*:

assumes $\text{trail } S = M2 @ M1$ **and** *wf*: $\text{wf-twl-state } S$

shows $\text{wf-twl-state } (\text{reduce-trail-to } M1 S)$

proof –

obtain $M N U k C$ **where** $S: S = \text{TWL-State } M N U k C$

by (*cases* S)

have *n-d*: *no-dup* M

using *wf* **by** (*auto simp: S comp-def wf-twl-state-def*)

have M : $M = \text{take } (\text{length } M - \text{length } M1) M @ \text{drop } (\text{length } M - \text{length } M1) M$

by *auto*

have [*simp*]: *no-dup* ($\text{drop } (\text{length } M - \text{length } M1) M$)

using *n-d* **by** (*metis distinct-drop drop-map*)

have $\bigwedge C. C \in \text{set } (\text{raw-clauses } S) \implies \text{wf-twl-cls } (\text{raw-trail } S) C$

using *wf* **by** (*auto simp: S comp-def wf-twl-state-def*)

then show *?thesis*

unfolding *wf-twl-state-def* S

using *wf-twl-cls-append*[*of take (length M – length M1) M drop (length M – length M1) M, unfolded M[symmetric]*]

by (*simp-all add: n-d*)

qed

lemma *trail-twl-twl-rough-state-reduce-trail-to*:

assumes $\text{trail-twl } st = M2 @ M1$

shows $\text{trail-twl } (\text{twl-of-rough-state } (\text{reduce-trail-to } M1 (\text{rough-state-of-twl } st))) = M1$

proof –

have $\text{wf-twl-state } (\text{reduce-trail-to } M1 (\text{rough-state-of-twl } st))$

using *wf-twl-reduce-trail-to* *assms* **by** *fastforce*

moreover

have $\text{length } (\text{trail-twl } st) - \text{length } M1 = \text{length } M2$

unfolding *assms* **by** *auto*
then have *trail* (*reduce-trail-to* *M1* (*rough-state-of-twl st*)) = *M1*
apply (*cases rough-state-of-twl st*)
using *assms* **by** (*auto simp: drop-map[symmetric]*)
ultimately show *?thesis*
using *twl-of-rough-state-inverse*[*of reduce-trail-to M1 (rough-state-of-twl st)*]
rough-state-of-twl[*of st*]
by (*auto simp add: assms*)
qed

lemma *twl-of-rough-state-reduce-trail-to:*

assumes *trail-twl st = M2 @ M1 and*

S: rough-cdcl.state (rough-state-of-twl st) = (M, S)

shows

rough-cdcl.state

(*rough-state-of-twl* (*twl-of-rough-state* (*reduce-trail-to M1 (rough-state-of-twl st)*))) =
(*M1, S*) (**is** *?st*) **and**

raw-init-clss-twl (*twl-of-rough-state* (*reduce-trail-to M1 (rough-state-of-twl st)*))

= *raw-init-clss-twl st* (**is** *?A*) **and**

raw-learned-clss-twl (*twl-of-rough-state* (*reduce-trail-to M1 (rough-state-of-twl st)*))

= *raw-learned-clss-twl st* (**is** *?B*) **and**

backtrack-lvl-twl (*twl-of-rough-state* (*reduce-trail-to M1 (rough-state-of-twl st)*))

= *backtrack-lvl-twl st* (**is** *?C*) **and**

rough-cdcl.conc-conflicting (*rough-state-of-twl* (*twl-of-rough-state*
(*reduce-trail-to M1 (rough-state-of-twl st)*)))

= *rough-cdcl.conc-conflicting* (*rough-state-of-twl st*) (**is** *?D*)

proof –

have *wf-twl-state* (*reduce-trail-to M1 (rough-state-of-twl st)*)

using *wf-twl-reduce-trail-to assms* **by** *fastforce*

moreover

have *length* (*trail-twl st*) – *length M1* = *length M2*

unfolding *assms* **by** *auto*

then have

raw-init-clss (*reduce-trail-to M1 (rough-state-of-twl st)*) = *raw-init-clss-twl st*

raw-learned-clss (*reduce-trail-to M1 (rough-state-of-twl st)*) = *raw-learned-clss-twl st*

backtrack-lvl (*reduce-trail-to M1 (rough-state-of-twl st)*) = *backtrack-lvl-twl st*

rough-cdcl.conc-conflicting (*reduce-trail-to M1 (rough-state-of-twl st)*) =

rough-cdcl.conc-conflicting (*rough-state-of-twl st*)

using *assms* **by** (*cases rough-state-of-twl st, auto simp: drop-map[symmetric]*) +

ultimately show *?A ?B ?C ?D*

using *twl-of-rough-state-inverse*[*of reduce-trail-to M1 (rough-state-of-twl st)*]

rough-state-of-twl[*of st*]

by (*auto simp add: assms*)

moreover have *trail-twl* (*twl-of-rough-state* (*reduce-trail-to M1 (rough-state-of-twl st)*)) = *M1*

using *trail-twl-twl-rough-state-reduce-trail-to*[*OF assms(1)*].

ultimately show *?st* **using** *S* **unfolding** *rough-cdcl.state-def*

by (*metis* (*no-types*) *prod.inject twl2.raw-clss-l-raw-init-clss-conc-init-clss*)

qed

lemma *add-learned-clss-rough-state-of-twl-simp:*

assumes *raw-conflicting-twl st = Some z*

shows

trail (*add-learned-clss z (rough-state-of-twl st)*) = *trail-twl st*

rough-cdcl.conc-init-clss (*add-learned-clss z (rough-state-of-twl st)*) =

rough-cdcl.conc-init-clss (*rough-state-of-twl st*)

```

    conc-learned-clss (local.add-learned-cls z (rough-state-of-twl st)) =
      {#mset z#} + conc-learned-clss (rough-state-of-twl st)
    backtrack-lvl (add-learned-cls z (rough-state-of-twl st)) = backtrack-lvl-twl st
using assms wf-twl-state-rough-state-of-twl[of st]
unfolding wf-twl-state-def apply
(auto simp: wf-watch add-learned-cls-def comp-def local.clause-watch
  ac-simps
  split: if-split-asm)
done

```

sublocale wf-twl: abs-state_W-ops

```

  clause
  raw-clss-l
  λL C. L ∈ set C op #

```

mset

```

λC. raw-clause C λC. TWL-Clause [] C
trail-twl λS. hd (raw-trail-twl S)
raw-clauses-twl
backtrack-lvl-twl
raw-conflicting-twl
conc-learned-clss-twl

```

```

cons-trail-twl
tl-trail-twl
λS. update-conflicting-twl None (add-learned-cls-twl (the (raw-conflicting-twl S)) S)
λC. remove-cls-twl (raw-clause C)
update-backtrack-lvl-twl
λC. update-conflicting-twl (Some C)
reduce-trail-to-twl
resolve-conflicting-twl
λN. init-state-twl (map raw-clause N)
restart-twl
by unfold-locales

```

lemma wf-twl-conc-init-clss-restart-twl[simp]:

```

  wf-twl.conc-init-clss (restart-twl S) = wf-twl.conc-init-clss S
using rough-state-of-twl-restart-twl wf-twl.conc-init-clss-def by auto

```

sublocale wf-twl: abs-state_W

```

  clause
  raw-clss-l
  λL C. L ∈ set C op #

```

mset

```

λC. raw-clause C λC. TWL-Clause [] C
trail-twl λS. hd (raw-trail-twl S)
raw-clauses-twl
backtrack-lvl-twl
raw-conflicting-twl
conc-learned-clss-twl

```

```

cons-trail-twl
tl-trail-twl

```

```

λS. update-conflicting-twl None (add-learned-clss-twl (the (raw-conflicting-twl S)) S)
λC. remove-clss-twl (raw-clause C)
update-backtrack-lvl-twl
λC. update-conflicting-twl (Some C)
reduce-trail-to-twl
resolve-conflicting-twl
λN. init-state-twl (map raw-clause N)
restart-twl
proof goal-cases
  case 1
  have stupid-locales: abs-stateW-ops clause raw-clss-l (λL C. L ∈ set C) op #
    mset raw-clause (TWL-Clause [])
  by unfold-locales
  have ugly[simp]: abs-stateW-ops.mmset-of-mlit clause = mmset-of-mlit
    apply (rule ext, rename-tac L, case-tac L)
  by (auto simp: abs-stateW-ops.mmset-of-mlit.simps[OF stupid-locales] clause-def
    raw-clause-def)
  have [simp]: ∧S. raw-clss-l (restart-learned S) ⊆# conc-learned-clss S
    using image-mset-subseteq-mono[OF restart-learned] unfolding mset-map
    by blast
  interpret abs-stateW-ops clause
    raw-clss-l
    λL C. L ∈ set C op #
    mset

  λC. raw-clause C λC. TWL-Clause [] C
  by unfold-locales
  have abs: ∧S. abs-stateW-ops.state raw-clss-l mset trail-twl raw-clauses-twl
    backtrack-lvl-twl raw-conflicting-twl conc-learned-clss-twl S =
    rough-cdcl.state (rough-state-of-twl S)
    using wf-twl.conc-init-clss-def unfolding abs-stateW-ops.state-def[OF stupid-locales] by auto
  show ?case
    apply unfold-locales
      using rough-cdcl.hd-raw-conc-trail unfolding ugly apply blast
      apply (auto simp add: rough-state-of-twl-cons-trail rough-cdcl.state-def abs; fail)[]
      apply (auto simp add: rough-state-of-twl-tl-trail rough-cdcl.state-def abs; fail)[]
      apply (auto simp add: rough-state-of-twl-remove-clss
        rough-state-of-twl-update-backtrack-lvl rough-cdcl.state-def abs; fail)[]
      apply (auto simp add: rough-state-of-twl-update-add-learned-clss rough-cdcl.state-def
        add-learned-clss-rough-state-of-twl-simp
        abs; fail)[]
      apply (auto simp add: rough-state-of-twl-update-backtrack-lvl
        rough-state-of-twl-update-conflicting rough-cdcl.state-def abs; fail)[]
      apply (auto simp add: rough-state-of-twl-update-add-learned-clss rough-cdcl.state-def
        add-learned-clss-rough-state-of-twl-simp
        rough-state-of-twl-update-conflicting abs; fail)[]
      apply (auto simp add: rough-state-of-twl-update-add-learned-clss rough-cdcl.state-def
        rough-state-of-twl-update-conflicting abs; fail)[]
      using twl-of-rough-state-reduce-trail-to(1) unfolding abs
      using rough-cdcl.conc-init-clss-restart-state rough-cdcl.conc-learned-clss-restart-state
      apply (simp add: rough-cdcl.resolve-conflicting twl2.rough-state-of-twl-update-conflicting;
        fail)
    using twl-of-rough-state-reduce-trail-to(1) unfolding abs
    using rough-cdcl.conc-init-clss-restart-state rough-cdcl.conc-learned-clss-restart-state
    by (auto simp: rough-state-of-twl-restart-twl abs
      rough-cdcl.state-def rough-state-of-twl-init-state comp-def)[8]

```

qed

sublocale *wf-twl*: *abs-conflict-driven-clause-learning_W*

clause

raw-clss-l

$\lambda L C. L \in \text{set } C \text{ op } \#$

mset

$\lambda C. \text{raw-clause } C \ \lambda C. \text{TWL-Clause } [] \ C$

trail-twl $\lambda S. \text{hd } (\text{raw-trail-twl } S)$

raw-clauses-twl

backtrack-lvl-twl

raw-conflicting-twl

conc-learned-clss-twl

cons-trail-twl

tl-trail-twl

$\lambda S. \text{update-conflicting-twl } \text{None } (\text{add-learned-clss-twl } (\text{the } (\text{raw-conflicting-twl } S)) \ S)$

$\lambda C. \text{remove-clss-twl } (\text{raw-clause } C)$

update-backtrack-lvl-twl

$\lambda C. \text{update-conflicting-twl } (\text{Some } C)$

reduce-trail-to-twl

resolve-conflicting-twl

$\lambda N. \text{init-state-twl } (\text{map raw-clause } N)$

restart-twl

by *unfold-locales*

declare *local.rough-cdcl.mset-clss-ccls-of-clss*[*simp del*]

abbreviation *state-eq-twl* (**infix** $\sim \text{TWL } 51$) **where**

state-eq-twl $S \ S' \equiv \text{rough-cdcl.state-eq } (\text{rough-state-of-twl } S) \ (\text{rough-state-of-twl } S')$

notation *wf-twl.state-eq* (**infix** ~ 51)

To avoid ambiguities:

no-notation *state-eq-twl* (**infix** ~ 51)

Alternative Definition of CDCL using the candidates of 2-WL *inductive propagate-twl*

$:: 'v \text{wf-twl} \Rightarrow 'v \text{wf-twl} \Rightarrow \text{bool}$ **where**

propagate-twl-rule: $(L, C) \in \text{candidates-propagate-twl } S \Rightarrow$

$S' \sim \text{cons-trail-twl } (\text{Propagated } L \ C) \ S \Rightarrow$

$\text{raw-conflicting-twl } S = \text{None} \Rightarrow$

propagate-twl $S \ S'$

lemma *cdcl_W-restart-all-struct-inv-clause-distinct-mset*:

cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv (*wf-twl.state* S) \Rightarrow

$C \in \text{set } (\text{CDCL-Two-Watched-Literals.raw-clauses-twl } S) \Rightarrow \text{distinct } (\text{raw-clause } C)$

unfolding *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv-def* *cdcl_W-restart-mset.distinct-cdcl_W-restart-state-def*

distinct-mset-set-def *wf-twl.conc-clauses-init-learned*

by (*metis* (*no-types*, *lifting*) *distinct-mset-distinct in-clss-mset-clss union-iff*

wf-twl.conc-clauses-init-learned *wf-twl.init-clss-state-conc-init-clss*

wf-twl.learned-clss-state-conc-learned-clss *wf-twl.mset-clss-ccls-of-clss*)

inductive-cases *propagate-twlE*: *propagate-twl* $S \ T$

lemma *propagate-twl-iff-propagate*:

assumes *inv*: *cdcl_W-restart-mset.cdcl_W-restart-all-struct-inv* (*wf-twl.state* S)

shows $wf\text{-}twl.\text{propagate-abs } S \ T \longleftrightarrow \text{propagate-twl } S \ T$ (is $?P \longleftrightarrow ?T$)

proof

assume $?P$

then obtain $L \ E$ where

raw-conflicting-twl $S = \text{None}$ and

CL-Clauses: $E \in \text{set } (\text{raw-clauses-twl } S)$ and

LE: $L \in \# \text{ clause } E$ and

tr-CNot: $\text{trail-twl } S \models_{as} \text{CNot } (\text{remove1-mset } L \ (\text{clause } E))$ and

undef-lot[simp]: $\text{undefined-lit } (\text{trail-twl } S) \ L$ and

$T \sim \text{cons-trail-twl } (\text{Propagated } L \ E) \ S$

by (blast elim: $wf\text{-}twl.\text{propagate-abs}E$)

have distinct (raw-clause E)

using inv CL-Clauses $cdcl_W\text{-restart-all-struct-inv-clause-distinct-mset}$ by blast

then have X : $\text{remove1-mset } L \ (\text{mset } (\text{raw-clause } E)) = \text{mset-set } (\text{set } (\text{raw-clause } E) - \{L\})$

by (auto simp: multiset-eq-iff raw-clause-def count-mset distinct-filter-eq-if)

have $(L, E) \in \text{candidates-propagate-twl } S$

apply (rule $wf\text{-}candidates\text{-propagate-complete}$)

using rough-state-of-twl apply (auto; fail)[]

using CL-Clauses apply (auto; fail)[]

using LE apply (simp add: clause-def; fail)

using tr-CNot X apply (simp add: clause-def; fail)

using undef-lot apply blast

done

show $?T$

apply (rule $\text{propagate-twl-rule}$)

apply (rule $\langle (L, E) \in \text{candidates-propagate-twl } S \rangle$)

using $\langle T \sim \text{cons-trail-twl } (\text{Propagated } L \ E) \ S \rangle$

apply (auto simp: $\langle \text{raw-conflicting-twl } S = \text{None} \rangle \text{cdcl}_W\text{-restart-mset.state-eq-def}$)

done

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: $\text{raw-conflicting-twl } S = \text{None}$

by (auto elim: $\text{propagate-twl}E$)

have

$C'S$: $C \in \text{set } (\text{raw-clauses-twl } S)$ and

L: $\text{set } (\text{watched } C) - \text{uminus ' lits-of-l } (\text{trail-twl } S) = \{L\}$ and

undef: $\text{undefined-lit } (\text{trail-twl } S) \ L$

using LC unfolding $\text{candidates-propagate-def}$ by auto

have dist: distinct (raw-clause C)

using inv $C'S$ $cdcl_W\text{-restart-all-struct-inv-clause-distinct-mset}$ by blast

then have C-L-L: $\text{mset-set } (\text{set } (\text{raw-clause } C) - \{L\}) = \text{clause } C - \{\#L\# \}$

by (metis distinct-mset-distinct distinct-mset-minus distinct-mset-set-mset-ident mset-remove1 set-mset-mset set-remove1-eq clause-def)

show $?P$

apply (rule $wf\text{-}twl.\text{propagate-abs-rule}[of \ S \ C \ L]$)

using confl apply (auto; fail)[]

using $C'S$ apply (simp; fail)

using L unfolding $\text{candidates-propagate-def}$

apply (auto simp: raw-clause-def clause-def; fail)[]

using $wf\text{-}candidates\text{-propagate-sound}[OF - LC]$ rough-state-of-twl dist

apply (simp add: distinct-mset-remove1-All true-annots-true-cls clause-def; fail)

using undef apply (simp; fail)

using T unfolding $cdcl_W\text{-restart-mset.state-eq-def}$ by auto
qed

no-notation $twl.state\text{-}eq\text{-}twl$ (infix $\sim TWL$ 51)

inductive $conflict\text{-}twl$ where

$conflict\text{-}twl\text{-}rule$:

$C \in candidates\text{-}conflict\text{-}twl\ S \implies$
 $S' \sim update\text{-}conflicting\text{-}twl\ (Some\ (raw\text{-}clause\ C))\ S \implies$
 $raw\text{-}conflicting\text{-}twl\ S = None \implies$
 $conflict\text{-}twl\ S\ S'$

inductive-cases $conflict\text{-}twlE$: $conflict\text{-}twl\ S\ T$

lemma $conflict\text{-}twl\text{-}iff\text{-}conflict$:

shows $wf\text{-}twl.conflict\text{-}abs\ S\ T \longleftrightarrow conflict\text{-}twl\ S\ T$ (is $?C \longleftrightarrow ?T$)

proof

assume $?C$

then obtain D where

S : $raw\text{-}conflicting\text{-}twl\ S = None$ and
 D : $D \in set\ (raw\text{-}clauses\text{-}twl\ S)$ and
 MD : $trail\text{-}twl\ S \models_{as} CNot\ (clause\ D)$ and
 T : $T \sim update\text{-}conflicting\text{-}twl\ (Some\ (raw\text{-}clause\ D))\ S$
by ($elim\ wf\text{-}twl.conflict\text{-}absE$)

have $D \in candidates\text{-}conflict\text{-}twl\ S$

apply (rule $wf\text{-}candidates\text{-}conflict\text{-}complete$)

apply simp

using D apply (auto; fail)[]

using $MD\ S$ by auto

moreover have $T \sim twl\text{-}of\text{-}rough\text{-}state\ (update\text{-}conflicting\ (Some\ (raw\text{-}clause\ D)))$
($rough\text{-}state\text{-}of\text{-}twl\ S$)

using T unfolding $cdcl_W\text{-restart-mset.state-eq-def}$ by auto

ultimately show $?T$

using S by (auto intro: $conflict\text{-}twl\text{-}rule$)

next

assume $?T$

then obtain C where

C : $C \in candidates\text{-}conflict\text{-}twl\ S$ and
 T : $T \sim update\text{-}conflicting\text{-}twl\ (Some\ (raw\text{-}clause\ C))\ S$ and
 $confl$: $raw\text{-}conflicting\text{-}twl\ S = None$
by (auto elim: $conflict\text{-}twlE$)

have

$C \in set\ (raw\text{-}clauses\text{-}twl\ S)$

using C unfolding $candidates\text{-}conflict\text{-}def$ by auto

moreover have $trail\text{-}twl\ S \models_{as} CNot\ (clause\ C)$

using $wf\text{-}candidates\text{-}conflict\text{-}sound[OF\ -\ C]$ by auto

ultimately show $?C$ apply –

apply (rule $wf\text{-}twl.conflict\text{-}abs\text{-}rule[OF\ -\ C]$)

using $confl\ T$ unfolding $cdcl_W\text{-restart-mset.state-eq-def}$ by (auto simp del: $map\text{-}map$)

qed

We have shown that we we can use $conflict\text{-}twl$ and $propagate\text{-}twl$ in a CDCL calculus.

end

end