

Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

March 29, 2016

Contents

1	Transitions	5
1.1	More theorems about Closures	5
1.2	Full Transitions	6
1.3	Well-Foundedness and Full Transitions	8
1.4	More Well-Foundedness	8
2	Various Lemmas	9
3	More List	10
3.1	<i>upt</i>	10
3.2	Lexicographic Ordering	11
3.3	Remove	12
3.3.1	More lemmas about remove	12
3.3.2	Remove under condition	12
4	Logics	13
4.1	Definition and abstraction	13
4.2	properties of the abstraction	15
4.3	Subformulas and properties	16
4.4	Positions	18
5	Semantics over the syntax	19
6	Rewrite systems and properties	20
6.1	Lifting of rewrite rules	20
6.2	Consistency preservation	21
6.3	Full Lifting	22
7	Transformation testing	22
7.1	Definition and first properties	22
7.2	Invariant conservation	23
7.2.1	Invariant while lifting of the rewriting relation	24
7.2.2	Invariant after all rewriting	24

8	Rewrite Rules	25
8.1	Elimination of the equivalences	25
8.2	Eliminate Implication	27
8.3	Eliminate all the True and False in the formula	28
8.4	PushNeg	32
8.5	Push inside	34
8.5.1	Only one type of connective in the formula (+ not)	37
8.5.2	Push Conjunction	38
8.5.3	Push Disjunction	39
9	The full transformations	39
9.1	Abstract Property characterizing that only some connective are inside the others	39
9.1.1	Definition	39
9.2	Conjunctive Normal Form	40
9.2.1	Full CNF transformation	41
9.3	Disjunctive Normal Form	41
9.3.1	Full DNF transform	41
10	More aggressive simplifications: Removing true and false at the beginning	42
10.1	Transformation	42
10.2	More invariants	43
10.3	The new CNF and DNF transformation	43
11	Partial Clausal Logic	44
11.1	Clauses	44
11.2	Partial Interpretations	44
11.2.1	Consistency	44
11.2.2	Atoms	45
11.2.3	Totality	47
11.2.4	Interpretations	48
11.2.5	Satisfiability	50
11.2.6	Entailment for Multisets of Clauses	51
11.2.7	Tautologies	52
11.2.8	Entailment for clauses and propositions	53
11.3	Subsumptions	56
11.4	Removing Duplicates	56
11.5	Set of all Simple Clauses	56
11.6	Experiment: Expressing the Entailments as Locales	57
11.7	Entailment to be extended	58
12	Link with Multiset Version	59
12.1	Transformation to Multiset	59
12.2	Equisatisfiability of the two Version	59
13	Resolution	61
13.1	Simplification Rules	61
13.2	Unconstrained Resolution	62
13.2.1	Subsumption	62
13.3	Inference Rule	62
13.4	Lemma about the simplified state	68

13.5	Resolution and Invariants	69
13.5.1	Invariants	69
13.5.2	well-foundness if the relation	72
14	Partial Clausal Logic	77
14.1	Marked Literals	77
14.1.1	Definition	77
14.1.2	Entailment	78
14.1.3	Defined and undefined literals	80
14.2	Backtracking	81
14.3	Decomposition with respect to the First Marked Literals	82
14.3.1	Definition	82
14.3.2	Entailment of the Propagated by the Marked Literal	84
14.4	Negation of Clauses	85
14.5	Other	88
14.6	Extending Entailments to multisets	88
14.7	Abstract Clause Representation	89
15	Measure	91
16	NOT's CDCL	93
16.1	Auxiliary Lemmas and Measure	93
16.2	Initial definitions	93
16.2.1	The state	93
16.2.2	Definition of the operation	96
16.3	DPLL with backjumping	98
16.3.1	Definition	99
16.3.2	Basic properties	100
16.3.3	Termination	101
16.3.4	Normal Forms	102
16.4	CDCL	105
16.4.1	Learn and Forget	105
16.4.2	Definition of CDCL	107
16.4.3	CDCL with invariant	109
16.4.4	Termination	111
16.4.5	Restricting learn and forget	111
16.5	CDCL with restarts	117
16.5.1	Definition	117
16.5.2	Increasing restarts	118
16.6	Merging backjump and learning	122
16.7	Instantiations	128
17	DPLL as an instance of NOT	134
17.1	DPLL with simple backtrack	134
17.2	Adding restarts	137

18 DPLL	137
18.1 Rules	137
18.2 Invariants	138
18.3 Termination	140
18.4 Final States	141
18.5 Link with NOT's DPLL	141
18.5.1 Level of literals and clauses	142
18.5.2 Properties about the levels	145
19 Weidenbach's CDCL	147
19.1 The State	147
19.2 CDCL Rules	156
19.3 Invariants	162
19.3.1 Properties of the trail	162
19.3.2 Better-Suited Induction Principle	164
19.3.3 Compatibility with $op \sim$	167
19.3.4 Conservation of some Properties	168
19.3.5 Learned Clause	169
19.3.6 No alien atom in the state	170
19.3.7 No duplicates all around	171
19.3.8 Conflicts and co	172
19.3.9 Putting all the invariants together	173
19.3.10 No tautology is learned	175
19.4 CDCL Strong Completeness	175
19.5 Higher level strategy	176
19.5.1 Definition	176
19.5.2 Invariants	178
19.5.3 Literal of highest level in conflicting clauses	181
19.5.4 Literal of highest level in marked literals	182
19.5.5 Strong completeness	184
19.5.6 No conflict with only variables of level less than backtrack level	185
19.5.7 Final States are Conclusive	188
19.6 Termination	190
19.7 No Relearning of a clause	191
19.8 Decrease of a measure	194
20 Simple Implementation of the DPLL and CDCL	196
20.1 Common Rules	196
20.1.1 Propagation	196
20.1.2 Unit propagation for all clauses	197
20.1.3 Decide	197
20.2 Simple Implementation of DPLL	198
20.2.1 Combining the propagate and decide: a DPLL step	198
20.2.2 Adding invariants	199
20.2.3 Code export	202
20.3 CDCL Implementation	205
20.3.1 Types and Additional Lemmas	205
20.3.2 The Transitions	207
20.3.3 Code generation	212

21 Merging backjump rules	218
21.1 Inclusion of the states	219
21.2 More lemmas conflict-propagate and backjumping	219
21.2.1 Termination	219
21.2.2 More backjumping	220
21.3 CDCL FW	222
21.4 FW with strategy	224
21.4.1 The intermediate step	224
21.4.2 Full Transformation	226
21.4.3 Termination and full Equivalence	230
21.5 Adding Restarts	231
22 Link between Weidenbach's and NOT's CDCL	236
22.1 Inclusion of the states	236
22.2 Additional Lemmas between NOT and W states	239
22.3 More lemmas conflict-propagate and backjumping	240
22.4 CDCL FW	240
23 Incremental SAT solving	241
24 2-Watched-Literal	245
24.1 Essence of 2-WL	245
24.1.1 Datastructure and Access Functions	245
24.1.2 Invariants	247
24.1.3 Abstract 2-WL	249
24.1.4 Instanciation of the previous locale	251
24.2 Two Watched-Literals with invariant	254
24.2.1 Interpretation for <i>conflict-driven-clause-learning_W.cdcl_W</i>	255
25 Superposition	260
25.1 We can now define the rules of the calculus	264

1 Transitions

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

```
theory Wellfounded-More
imports Main
```

```
begin
```

1.1 More theorems about Closures

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

```
lemma tranclp-mono-explicit:
   $r^{++} \ a \ b \implies r \leq s \implies s^{++} \ a \ b$ 
  <proof>
```

```
lemma tranclp-mono:
  assumes mono:  $r \leq s$ 
```

shows $r^{++} \leq s^{++}$
 $\langle \text{proof} \rangle$

lemma *tranclp-idemp-rel*:
 $R^{++++} a b \longleftrightarrow R^{++} a b$
 $\langle \text{proof} \rangle$

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

lemma *trancl-idemp*: $(r^+)^+ = r^+$
 $\langle \text{proof} \rangle$

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

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

lemma *rtranclp-unfold*: $rtranclp r a b \longleftrightarrow (a = b \vee tranclp r a b)$
 $\langle \text{proof} \rangle$

lemma *tranclp-unfold-end*: $tranclp r a b \longleftrightarrow (\exists a'. rtranclp r a a' \wedge r a' b)$
 $\langle \text{proof} \rangle$

Near duplicate of $?R^{++} ?x ?y \implies \exists z. ?R ?x z \wedge ?R^{**} z ?y$:

lemma *tranclp-unfold-begin*: $tranclp r a b \longleftrightarrow (\exists a'. r a a' \wedge rtranclp r a' b)$
 $\langle \text{proof} \rangle$

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

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

lemma *tranclp-rtranclp-rtranclp[simp]*: $R^{+***} = R^{**}$
 $\langle \text{proof} \rangle$

lemma *rtranclp-exists-last-with-prop*:
assumes $R x z$
and $R^{**} z z'$ **and** $P x z$
shows $\exists y y'. R^{**} x y \wedge R y y' \wedge P y y' \wedge (\lambda a b. R a b \wedge \neg P a b)^{**} y' z'$
 $\langle \text{proof} \rangle$

lemma *rtranclp-and-rtranclp-left*: $(\lambda a b. P a b \wedge Q a b)^{**} S T \implies P^{**} S T$
 $\langle \text{proof} \rangle$

1.2 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**
 $\text{full1 } \text{transf} = (\lambda S S'. \text{tranclp } \text{transf } S S' \wedge (\forall S''. \neg \text{transf } S' S''))$

definition *full*:: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**

$$full\ transf = (\lambda S\ S'.\ rtranc\lp\ transf\ S\ S' \wedge (\forall S''.\ \neg\ transf\ S'\ S''))$$

We define output notations only for printing:

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

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

lemma $rtranc\lp\text{-}full1I$:

$$R^{**}\ a\ b \implies full1\ R\ b\ c \implies full1\ R\ a\ c$$

$\langle proof \rangle$

lemma $tranc\lp\text{-}full1I$:

$$R^{++}\ a\ b \implies full1\ R\ b\ c \implies full1\ R\ a\ c$$

$\langle proof \rangle$

lemma $rtranc\lp\text{-}fullI$:

$$R^{**}\ a\ b \implies full\ R\ b\ c \implies full\ R\ a\ c$$

$\langle proof \rangle$

lemma $tranc\lp\text{-}full\text{-}full1I$:

$$R^{++}\ a\ b \implies full\ R\ b\ c \implies full1\ R\ a\ c$$

$\langle proof \rangle$

lemma $full\text{-}fullI$:

$$R\ a\ b \implies full\ R\ b\ c \implies full1\ R\ a\ c$$

$\langle proof \rangle$

lemma $full\text{-}unfold$:

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma $full1\text{-}tranc\lp\text{-}relation\text{-}full$:

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

$\langle proof \rangle$

lemma $full\text{-}tranc\lp\text{-}relation\text{-}full$:

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

$\langle proof \rangle$

lemma $rtranc\lp\text{-}full1\text{-}eq\text{-}or\text{-}full1$:

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

$\langle proof \rangle$

lemma $tranc\lp\text{-}full1\text{-}full1$:

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

$\langle proof \rangle$

1.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:
assumes $wf:wf \ \{(x, y). R \ y \ x\}$
shows $\exists b. R^{**} \ a \ b \wedge no_step \ R \ b$
 $\langle proof \rangle$

lemma *wf-exists-normal-form-full*:
assumes $wf:wf \ \{(x, y). R \ y \ x\}$
shows $\exists b. full \ R \ a \ b$
 $\langle proof \rangle$

1.4 More Well-Foundedness

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

- link between *wf* and infinite chains: $wf \ ?r = (\nexists f. \forall i. (f \ (Suc \ i), f \ i) \in \ ?r), \llbracket wf \ ?r; \bigwedge k. (\ ?f \ (Suc \ k), \ ?f \ k) \notin \ ?r \implies \ ?thesis \rrbracket \implies \ ?thesis$

lemma *wf-if-measure-in-wf*:
 $wf \ R \implies (\bigwedge a \ b. (a, b) \in S \implies (\nu \ a, \nu \ b) \in R) \implies wf \ S$
 $\langle proof \rangle$

lemma *wfP-if-measure*: **fixes** $f :: 'a \Rightarrow nat$
shows $(\bigwedge x \ y. P \ x \implies g \ x \ y \implies f \ y < f \ x) \implies wf \ \{(y, x). P \ x \wedge g \ x \ y\}$
 $\langle proof \rangle$

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

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 $wf \ \{(y, x). P \ x \wedge g \ x \ y\}$
 $\langle proof \rangle$

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

lemma *wfP-if-measure2*: **fixes** $f :: 'a \Rightarrow nat$
shows $(\bigwedge x \ y. P \ x \ y \implies g \ x \ y \implies f \ x < f \ y) \implies wf \ \{(x, y). P \ x \ y \wedge g \ x \ y\}$
 $\langle proof \rangle$

lemma *lexord-on-finite-set-is-wf*:
assumes
 $P\text{-finite}: \bigwedge U. P \ U \longrightarrow U \in A$ **and**
 $finite: finite \ A$ **and**
 $wf: wf \ R$ **and**
 $trans: trans \ R$
shows $wf \ \{(T, S). (P \ S \wedge P \ T) \wedge (T, S) \in lexord \ R\}$
 $\langle proof \rangle$

lemma *wf-fst-wf-pair*:


```

assumes wf  $\{(M', M). R M' M\}$ 
shows wf  $\{((M', N'), (M, N)). R M' M\}$ 
 $\langle proof \rangle$ 

```

```

lemma wf-snd-wf-pair:
  assumes wf  $\{(M', M). R M' M\}$ 
  shows wf  $\{((M', N'), (M, N)). R N' N\}$ 
 $\langle proof \rangle$ 

```

```

lemma wf-if-measure-f-notation2:
  assumes wf r
  shows wf  $\{(b, h a) | b a. (f b, f (h a)) \in r\}$ 
 $\langle proof \rangle$ 

```

```

lemma wf-wf-if-measure'-notation2:
assumes wf r and H:  $(\bigwedge x y. P x \implies g x y \implies (f y, f (h x)) \in r)$ 
shows wf  $\{(y, h x) | y x. P x \wedge g x y\}$ 
 $\langle proof \rangle$ 

```

```

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

```

Sledgehammer parameters

```
sledgehammer-params[debug]
```

2 Various Lemmas

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

```

thm nat-less-induct
lemma nat-less-induct-case[case-names 0 Suc]:
  assumes
    P 0 and
     $\bigwedge n. (\forall m < Suc n. P m) \implies P (Suc n)$ 
  shows P n
 $\langle proof \rangle$ 

```

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

```

lemma if-0-1-ge-0[simp]:
   $0 < (if P then a else (0::nat)) \longleftrightarrow P \wedge 0 < a$ 
 $\langle proof \rangle$ 

```

Bounded function have not yet been defined in Isabelle.

```

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

```

```

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

```

lemma *not-bounded-nat-exists-larger*:
fixes $f :: nat \Rightarrow nat$
assumes *unbound*: *unbounded* f
shows $\exists n. f\ n > m \wedge n > n_0$
 $\langle proof \rangle$

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 $k = (0::'a)$ and $f = (\lambda i. i)$ for a counter-example).

lemma *bounded-const-product*:
fixes $k :: nat$ **and** $f :: nat \Rightarrow nat$
assumes $k > 0$
shows *bounded* $f \longleftrightarrow \text{bounded } (\lambda i. k * f\ i)$
 $\langle proof \rangle$

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

lemma *bounded-finite-linorder*:
fixes $f :: 'a \Rightarrow 'a :: \{finite, linorder\}$
shows *bounded* f
 $\langle proof \rangle$

3 More List

3.1 *upt*

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

lemma *upt-Suc-le-append*: $\neg i \leq j \implies [i..<Suc\ j] = []$
 $\langle proof \rangle$

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

declare *upt.simps*(2)[*simp del*]

lemma
assumes $i \leq n - m$
shows *take* $i\ [m..<n] = [m..<m+i]$
 $\langle proof \rangle$

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

lemma *take-upt-bound-minus*[*simp*]:
assumes $i \leq n - m$
shows *take* $i\ [m..<n] = [m..<m+i]$
 $\langle proof \rangle$

lemma *append-cons-eq-upt*:
assumes $A @ B = [m..<n]$
shows $A = [m..<m+\text{length } A]$ **and** $B = [m + \text{length } A..<n]$
 $\langle proof \rangle$

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

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

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

$\langle \text{proof} \rangle$

A more restrictive version holds:

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

(is $?P \implies ?A = ?B$)

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *upt-decomp-lt*:

assumes $H: xs @ i \# ys @ j \# zs = [m ..< n]$

shows $i < j$

$\langle \text{proof} \rangle$

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

lemma *length-list-Suc-0*:

$\text{length } W = \text{Suc } 0 \longleftrightarrow (\exists L. W = [L])$

$\langle \text{proof} \rangle$

lemma *length-list-2*: $\text{length } S = 2 \longleftrightarrow (\exists a b. S = [a, b])$

$\langle \text{proof} \rangle$

3.2 Lexicographic Ordering

lemma *lexn-Suc*:

$(x \# xs, y \# ys) \in \text{lexn } r (\text{Suc } n) \longleftrightarrow$

$(\text{length } xs = n \wedge \text{length } ys = n) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in \text{lexn } r n))$

$\langle \text{proof} \rangle$

lemma *lexn-n*:

$n > 0 \implies (x \# xs, y \# ys) \in \text{lexn } r n \longleftrightarrow$

$(\text{length } xs = n-1 \wedge \text{length } ys = n-1) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in \text{lexn } r (n-1)))$

$\langle \text{proof} \rangle$

There is some subtle point in the proof here. 1 is converted to $\text{Suc } 0$, but 2 is not: meaning that 1 is automatically simplified by default using the default simplification rule $\text{lexn } ?r \ 0 = \{\}$

$\text{lexn } ?r (\text{Suc } ?n) = \text{map-prod } (\lambda(x, xs). x \# xs) (\lambda(x, xs). x \# xs) \text{ ' } (?r <*\text{lex}*> \text{lexn } ?r \ ?n) \cap \{(xs, ys). \text{length } xs = \text{Suc } ?n \wedge \text{length } ys = \text{Suc } ?n\}$. However, the latter needs additional simplification rule (see the proof of the theorem above).

lemma *lexn2-conv*:

$([a, b], [c, d]) \in \text{lexn } r \ 2 \longleftrightarrow (a, c) \in r \vee (a = c \wedge (b, d) \in r)$
 $\langle \text{proof} \rangle$

lemma *lexn3-conv*:

$([a, b, c], [a', b', c']) \in \text{lexn } r \ 3 \longleftrightarrow$
 $(a, a') \in r \vee (a = a' \wedge (b, b') \in r) \vee (a = a' \wedge b = b' \wedge (c, c') \in r)$
 $\langle \text{proof} \rangle$

3.3 Remove

3.3.1 More lemmas about remove

lemma *remove1-nil*:

$\text{remove1 } (- \ L) \ W = [] \longleftrightarrow (W = [] \vee W = [-L])$
 $\langle \text{proof} \rangle$

lemma *remove1-mset-single-add*:

$a \neq b \implies \text{remove1-mset } a \ (\{\#b\# \} + C) = \{\#b\# \} + \text{remove1-mset } a \ C$
 $\text{remove1-mset } a \ (\{\#a\# \} + C) = C$
 $\langle \text{proof} \rangle$

3.3.2 Remove under condition

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

fun *remove1-cond where*

$\text{remove1-cond } f \ [] = [] \mid$
 $\text{remove1-cond } f \ (C' \# L) = (\text{if } f \ C' \text{ then } L \text{ else } C' \# \text{remove1-cond } f \ L)$

lemma *remove1 x xs = remove1-cond ((op =) x) xs*

$\langle \text{proof} \rangle$

lemma *mset-map-mset-remove1-cond*:

$\text{mset } (\text{map mset } (\text{remove1-cond } (\lambda L. \text{mset } L = \text{mset } a) \ C)) =$
 $\text{remove1-mset } (\text{mset } a) (\text{mset } (\text{map mset } C))$
 $\langle \text{proof} \rangle$

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

fun *removeAll-cond where*

$\text{removeAll-cond } f \ [] = [] \mid$
 $\text{removeAll-cond } f \ (C' \# L) =$
 $(\text{if } f \ C' \text{ then } \text{removeAll-cond } f \ L \text{ else } C' \# \text{removeAll-cond } f \ L)$

lemma *removeAll x xs = removeAll-cond ((op =) x) xs*

$\langle \text{proof} \rangle$

lemma *removeAll-cond* $P \text{ xs} = \text{filter } (\lambda x. \neg P \ x) \text{ xs}$
 <proof>

lemma *mset-map-mset-removeAll-cond*:
 $\text{mset } (\text{map mset } (\text{removeAll-cond } (\lambda b. \text{mset } b = \text{mset } a) \ C))$
 $= \text{removeAll-mset } (\text{mset } a) (\text{mset } (\text{map mset } C))$
 <proof>

Take from `../lib/Multiset_More.thy`, but named:

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

lemma *union-mset-list*:
 $\text{mset } \text{xs} \ \# \cup \ \text{mset } \text{ys} = \text{mset } (\text{union-mset-list } \text{xs } \text{ys})$
 <proof>

end
theory *Prop-Logic*

imports *Main*

begin

4 Logics

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

4.1 Definition and abstraction

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

datatype *'v propo* =
 $FT \mid FF \mid FVar \ 'v \mid FNot \ 'v \ propo \mid FAnd \ 'v \ propo \ 'v \ propo \mid FOr \ 'v \ propo \ 'v \ propo$
 $\mid FImp \ 'v \ propo \ 'v \ propo \mid FEq \ 'v \ propo \ 'v \ propo$

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* = $CT \mid CF \mid CVar \ 'v \mid CNot \mid CAnd \mid COr \mid CImp \mid CEq$

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  $P \ \psi$ 
 $\langle proof \rangle$ 

```

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

```

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

```

lemma connective-cases-arity[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  $P$ 
 $\langle proof \rangle$ 

```

```

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  $P$ 
 $\langle proof \rangle$ 

```

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

```

```

lemma wf-conn-induct[consumes 1, case-names CT CF CVar CNot COOr 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
  ( $\bigwedge \psi \ \psi'. c = COOr \implies P \ [\psi, \ \psi']$ ) 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  $P \ x$ 
 $\langle proof \rangle$ 

```

4.2 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 $(CVar\ x)\ l \implies conn\ (CVar\ x)\ l = FVar\ x$
 $\langle proof \rangle$

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

wf-conn $CT\ l \longleftrightarrow l = []$
wf-conn $CF\ l \longleftrightarrow l = []$
wf-conn $(CVar\ x)\ l \longleftrightarrow l = []$
wf-conn $CNot\ (\xi @ \varphi \# \xi') \longleftrightarrow \xi = [] \wedge \xi' = []$
 $\langle proof \rangle$

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 $c\ l \implies conn\ c\ l = FNot\ a \longleftrightarrow (c = CNot \wedge l = a \# [])$
 $\langle proof \rangle$

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

lemma *list-length2-decomp*: $length\ l = 2 \implies (\exists\ a\ b. l = a \# b \# [])$

$\langle proof \rangle$

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 $length\ l = 2 \longleftrightarrow wf-conn\ c\ l$
 $\langle proof \rangle$

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

fixes $l :: 'v\ propo\ list$
shows $wf-conn\ CNot\ l \longleftrightarrow length\ l = 1$
 $\langle proof \rangle$

Decomposing the Not into an element is moreover very useful.

lemma *wf-conn-Not-decomp*:

fixes $l :: 'v\ propo\ list$ **and** $a :: 'v$
assumes *corr*: $wf-conn\ CNot\ l$
shows $\exists\ a. l = [a]$
 $\langle proof \rangle$

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*:
 $\text{length } l = \text{length } l' \implies \text{wf-conn } c \ l \longleftrightarrow \text{wf-conn } c \ l'$
 $\langle \text{proof} \rangle$

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

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

lemma *conn-inj-not*:
assumes *correct*: $\text{wf-conn } c \ l$
and *conn*: $\text{conn } c \ l = \text{FNot } \psi$
shows $c = \text{CNot}$ **and** $l = [\psi]$
 $\langle \text{proof} \rangle$

lemma *conn-inj*:
fixes $c \ ca :: 'v \text{ connective}$ **and** $l \ \psi s :: 'v \text{ propo list}$
assumes *corr*: $\text{wf-conn } ca \ l$
and *corr'*: $\text{wf-conn } c \ \psi s$
and *eq*: $\text{conn } ca \ l = \text{conn } c \ \psi s$
shows $ca = c \wedge \psi s = l$
 $\langle \text{proof} \rangle$

4.3 Subformulas and properties

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

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

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

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:
shows $b: \text{FNot } \varphi \preceq \psi \implies \varphi \preceq \psi$
 $\langle \text{proof} \rangle$

lemma *subformula-in-binary-conn*:
assumes *conn*: $c \in \text{binary-connectives}$
shows $f \preceq \text{conn } c \ [f, g]$
and $g \preceq \text{conn } c \ [f, g]$
 $\langle \text{proof} \rangle$

lemma *subformula-trans*:
 $\psi \preceq \psi' \implies \varphi \preceq \psi \implies \varphi \preceq \psi'$
 $\langle \text{proof} \rangle$

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 $\varphi = \psi$
 $\langle proof \rangle$

lemma *subformula-not-incl-eq*:
assumes $\varphi \preceq conn\ c\ l$
and *wf-conn* $c\ l$
and $\forall \psi. \psi \in set\ l \longrightarrow \neg \varphi \preceq \psi$
shows $\varphi = conn\ c\ l$
 $\langle proof \rangle$

lemma *wf-subformula-conn-cases*:
 $wf-conn\ c\ l \implies \varphi \preceq conn\ c\ l \longleftrightarrow (\varphi = conn\ c\ l \vee (\exists \psi. \psi \in set\ l \wedge \varphi \preceq \psi))$
 $\langle proof \rangle$

lemma *subformula-decomp-explicit[simp]*:
 $\varphi \preceq FAnd\ \psi\ \psi' \longleftrightarrow (\varphi = FAnd\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$ (**is** $?P\ FAnd$)
 $\varphi \preceq FOr\ \psi\ \psi' \longleftrightarrow (\varphi = FOr\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
 $\varphi \preceq FEq\ \psi\ \psi' \longleftrightarrow (\varphi = FEq\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
 $\varphi \preceq FImp\ \psi\ \psi' \longleftrightarrow (\varphi = FImp\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
 $\langle proof \rangle$

lemma *wf-conn-helper-facts[iff]*:
 $wf-conn\ CNot\ [\varphi]$
 $wf-conn\ CT\ []$
 $wf-conn\ CF\ []$
 $wf-conn\ (CVar\ x)\ []$
 $wf-conn\ CAnd\ [\varphi, \psi]$
 $wf-conn\ COr\ [\varphi, \psi]$
 $wf-conn\ CImp\ [\varphi, \psi]$
 $wf-conn\ CEq\ [\varphi, \psi]$
 $\langle proof \rangle$

lemma *exists-c-conn*: $\exists\ c\ l. \varphi = conn\ c\ l \wedge wf-conn\ c\ l$
 $\langle proof \rangle$

lemma *subformula-conn-decomp[simp]*:
assumes *wf*: $wf-conn\ c\ l$
shows $\varphi \preceq conn\ c\ l \longleftrightarrow (\varphi = conn\ c\ l \vee (\exists \psi \in set\ l. \varphi \preceq \psi))$ (**is** $?A \longleftrightarrow ?B$)
 $\langle proof \rangle$

lemma *subformula-leaf-explicit[simp]*:
 $\varphi \preceq FT \longleftrightarrow \varphi = FT$
 $\varphi \preceq FF \longleftrightarrow \varphi = FF$
 $\varphi \preceq FVar\ x \longleftrightarrow \varphi = FVar\ x$
 $\langle proof \rangle$

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

primrec *vars-of-prop*:: $'v\ propo \Rightarrow 'v\ set$ **where**
 $vars-of-prop\ FT = \{\}$ |
 $vars-of-prop\ FF = \{\}$ |
 $vars-of-prop\ (FVar\ x) = \{x\}$ |
 $vars-of-prop\ (FNot\ \varphi) = vars-of-prop\ \varphi$ |
 $vars-of-prop\ (FAnd\ \varphi\ \psi) = vars-of-prop\ \varphi \cup vars-of-prop\ \psi$ |
 $vars-of-prop\ (FOr\ \varphi\ \psi) = vars-of-prop\ \varphi \cup vars-of-prop\ \psi$ |

$\text{vars-of-prop } (FImp \varphi \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi \mid$
 $\text{vars-of-prop } (FEq \varphi \psi) = \text{vars-of-prop } \varphi \cup \text{vars-of-prop } \psi$

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*: $\text{wf-conn } c \ l$ **and** *incl*: $\psi \in \text{set } l$

shows $\text{vars-of-prop } \psi \subseteq \text{vars-of-prop } (\text{conn } c \ l)$

<proof>

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:

$\varphi \preceq \psi \implies \text{vars-of-prop } \varphi \subseteq \text{vars-of-prop } \psi$

<proof>

4.4 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 $FF = \{\square\} \mid$

pos $FT = \{\square\} \mid$

pos $(FVar \ x) = \{\square\} \mid$

pos $(FAnd \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$

pos $(FOr \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$

pos $(FEq \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$

pos $(FImp \ \varphi \ \psi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\} \cup \{R \ \# \ p \mid p. p \in \text{pos } \psi\} \mid$

pos $(FNot \ \varphi) = \{\square\} \cup \{L \ \# \ p \mid p. p \in \text{pos } \varphi\}$

lemma *finite-pos*: $\text{finite } (\text{pos } \varphi)$

<proof>

lemma *finite-inj-comp-set*:

fixes $s :: 'v \text{ set}$

assumes *finite*: $\text{finite } s$

and *inj*: $\text{inj } f$

shows $\text{card } (\{f \ p \mid p. p \in s\}) = \text{card } s$

<proof>

lemma *cons-inject*:

$\text{inj } (op \ \# \ s)$

<proof>

lemma *finite-insert-nil-cons*:

$\text{finite } s \implies \text{card } (\text{insert } \square \ \{L \ \# \ p \mid p. p \in s\}) = 1 + \text{card } \{L \ \# \ p \mid p. p \in s\}$

<proof>

lemma *card-not[simp]*:

$\text{card } (\text{pos } (FNot \ \varphi)) = 1 + \text{card } (\text{pos } \varphi)$

<proof>

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** $\text{card } (?L \cup ?R) = \text{card } ?L + \text{card } ?R$)
 $\langle \text{proof} \rangle$

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

lemma *path-to-subformula*:

path-to *p* $\varphi \varphi' \Rightarrow \varphi' \preceq \varphi$

$\langle \text{proof} \rangle$

lemma *subformula-path-exists*:

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

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

$\langle \text{proof} \rangle$

fun *replace-at* :: *sign list* $\Rightarrow 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow 'v \text{ propo}$ **where**

replace-at [] - $\psi = \psi$ |

replace-at (*L* # *l*) (*FAnd* $\varphi \varphi'$) $\psi = \text{FAnd } (\text{replace-at } l \varphi \psi) \varphi'$ |

replace-at (*R* # *l*) (*FAnd* $\varphi \varphi'$) $\psi = \text{FAnd } \varphi (\text{replace-at } l \varphi' \psi)$ |

replace-at (*L* # *l*) (*FOr* $\varphi \varphi'$) $\psi = \text{FOr } (\text{replace-at } l \varphi \psi) \varphi'$ |

replace-at (*R* # *l*) (*FOr* $\varphi \varphi'$) $\psi = \text{FOr } \varphi (\text{replace-at } l \varphi' \psi)$ |

replace-at (*L* # *l*) (*FEq* $\varphi \varphi'$) $\psi = \text{FEq } (\text{replace-at } l \varphi \psi) \varphi'$ |

replace-at (*R* # *l*) (*FEq* $\varphi \varphi'$) $\psi = \text{FEq } \varphi (\text{replace-at } l \varphi' \psi)$ |

replace-at (*L* # *l*) (*FImp* $\varphi \varphi'$) $\psi = \text{FImp } (\text{replace-at } l \varphi \psi) \varphi'$ |

replace-at (*R* # *l*) (*FImp* $\varphi \varphi'$) $\psi = \text{FImp } \varphi (\text{replace-at } l \varphi' \psi)$ |

replace-at (*L* # *l*) (*FNot* φ) $\psi = \text{FNot } (\text{replace-at } l \varphi \psi)$

5 Semantics over the syntax

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

fun *eval* :: ($'v \Rightarrow \text{bool}$) $\Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ (**infix** $\models 50$) **where**

$\mathcal{A} \models \text{FT} = \text{True}$ |

$\mathcal{A} \models \text{FF} = \text{False}$ |

$\mathcal{A} \models \text{FVar } v = (\mathcal{A} \ v)$ |

$\mathcal{A} \models \text{FNot } \varphi = (\neg(\mathcal{A} \models \varphi))$ |

$\mathcal{A} \models \text{FAnd } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \wedge \mathcal{A} \models \varphi_2)$ |

$\mathcal{A} \models \text{For } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \vee \mathcal{A} \models \varphi_2) \mid$
 $\mathcal{A} \models \text{FImp } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \longrightarrow \mathcal{A} \models \varphi_2) \mid$
 $\mathcal{A} \models \text{FEq } \varphi_1 \varphi_2 = (\mathcal{A} \models \varphi_1 \longleftrightarrow \mathcal{A} \models \varphi_2)$

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

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

theorem *deduction-theorem*:

$(\varphi \models_f \psi) \longleftrightarrow (\forall A. (A \models \text{FImp } \varphi \psi))$
 $\langle \text{proof} \rangle$

A shorter proof:

lemma $\varphi \models_f \psi \longleftrightarrow (\forall A. A \models \text{FImp } \varphi \psi)$
 $\langle \text{proof} \rangle$

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

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

lemma *same-over-set-eval*:

assumes *same-over-set* $A B$ (*vars-of-prop* φ)
shows $A \models \varphi \longleftrightarrow B \models \varphi$
 $\langle \text{proof} \rangle$

end

theory *Prop-Abstract-Transformation*

imports *Main Prop-Logic Wellfounded-More*

begin

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

6 Rewrite systems and properties

6.1 Lifting of rewrite rules

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

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

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

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

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

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

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

This lemma is only a health condition:

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

shows *propo-rew-step* $r \varphi \varphi' \implies \exists \psi \psi'. \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r \psi \psi'$
 $\langle \text{proof} \rangle$

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

lemma *propo-rew-step-subformula-rec*:
fixes $\psi \psi' \varphi :: 'v \text{ propo}$
shows $\psi \preceq \varphi \implies r \psi \psi' \implies (\exists \varphi'. \psi' \preceq \varphi' \wedge \text{propo-rew-step } r \varphi \varphi')$
 $\langle \text{proof} \rangle$

lemma *propo-rew-step-subformula*:
 $(\exists \psi \psi'. \psi \preceq \varphi \wedge r \psi \psi') \longleftrightarrow (\exists \varphi'. \text{propo-rew-step } r \varphi \varphi')$
 $\langle \text{proof} \rangle$

lemma *consistency-decompose-into-list*:
assumes $wf: wf\text{-conn } c \ l$ **and** $wf': wf\text{-conn } c \ l'$
and same: $\forall n. (A \models l \ ! \ n \longleftrightarrow (A \models l' \ ! \ n))$
shows $(A \models \text{conn } c \ l) = (A \models \text{conn } c \ l')$
 $\langle \text{proof} \rangle$

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 $\exists \psi \psi' p. r \psi \psi' \wedge \text{path-to } p \varphi \psi \wedge \text{replace-at } p \varphi \psi' = \varphi'$
 $\langle \text{proof} \rangle$

6.2 Consistency preservation

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

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

lemma *propo-rew-step-preservers-val-explicit*:
 $\text{propo-rew-step } r \varphi \psi \implies \text{preserves-un-sat } r \implies \text{propo-rew-step } r \varphi \psi \implies (\forall A. A \models \varphi \longleftrightarrow A \models \psi)$
 $\langle \text{proof} \rangle$

lemma *propo-rew-step-preservers-val*:
assumes *preserves-un-sat* r
shows *preserves-un-sat* (*propo-rew-step* r)
 $\langle \text{proof} \rangle$

lemma *preserves-un-sat-OO[intro]*:
 $\text{preserves-un-sat } f \implies \text{preserves-un-sat } g \implies \text{preserves-un-sat } (f \text{ OO } g)$
 $\langle \text{proof} \rangle$

lemma *star-consistency-preservation-explicit*:
assumes $(\text{propo-rew-step } r)^{\wedge **} \varphi \psi$ **and** *preserves-un-sat* r
shows $\forall A. A \models \varphi \longleftrightarrow A \models \psi$

$\langle \text{proof} \rangle$

lemma *star-consistency-preservation*:

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

$\langle \text{proof} \rangle$

6.3 Full Lifting

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

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

7 Transformation testing

7.1 Definition and first properties

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

definition *all-subformula-st* :: $('a \text{ propo} \Rightarrow \text{bool}) \Rightarrow 'a \text{ propo} \Rightarrow \text{bool}$ **where**

$\text{all-subformula-st } \text{test-symb } \varphi \equiv \forall \psi. \psi \preceq \varphi \longrightarrow \text{test-symb } \psi$

lemma *test-symb-imp-all-subformula-st[simp]*:

$\text{test-symb } FT \implies \text{all-subformula-st } \text{test-symb } FT$

$\text{test-symb } FF \implies \text{all-subformula-st } \text{test-symb } FF$

$\text{test-symb } (FVar \ x) \implies \text{all-subformula-st } \text{test-symb } (FVar \ x)$

$\langle \text{proof} \rangle$

lemma *all-subformula-st-test-symb-true-phi*:

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

$\langle \text{proof} \rangle$

lemma *all-subformula-st-decomp-imp*:

$\text{wf-conn } c \ l \implies (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st } \text{test-symb } \varphi))$

$\implies \text{all-subformula-st } \text{test-symb } (\text{conn } c \ l)$

$\langle \text{proof} \rangle$

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

lemma *all-subformula-st-decomp-rec*:

$\text{all-subformula-st } \text{test-symb } (\text{conn } c \ l) \implies \text{wf-conn } c \ l$

$\implies (\text{test-symb } (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st } \text{test-symb } \varphi))$

$\langle \text{proof} \rangle$

lemma *all-subformula-st-decomp*:

fixes $c :: 'v \text{ connective}$ **and** $l :: 'v \text{ propo list}$
assumes *wf-conn* $c \ l$
shows *all-subformula-st test-symb* (*conn* $c \ l$)
 $\longleftrightarrow (\text{test-symb} (\text{conn } c \ l) \wedge (\forall \varphi \in \text{set } l. \text{all-subformula-st test-symb } \varphi))$
 $\langle \text{proof} \rangle$

lemma *helper-fact*: $c \in \text{binary-connectives} \longleftrightarrow (c = COr \vee c = CAnd \vee c = CEq \vee c = CImp)$
 $\langle \text{proof} \rangle$

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

fixes $\varphi \ \psi :: 'v \text{ propo}$
shows *all-subformula-st test-symb* (*FAnd* $\varphi \ \psi$)
 $\longleftrightarrow (\text{test-symb} (\text{FAnd } \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb* (*FOr* $\varphi \ \psi$)
 $\longleftrightarrow (\text{test-symb} (\text{FOr } \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb* (*FNot* φ)
 $\longleftrightarrow (\text{test-symb} (\text{FNot } \varphi) \wedge \text{all-subformula-st test-symb } \varphi)$
and *all-subformula-st test-symb* (*FEq* $\varphi \ \psi$)
 $\longleftrightarrow (\text{test-symb} (\text{FEq } \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
and *all-subformula-st test-symb* (*FImp* $\varphi \ \psi$)
 $\longleftrightarrow (\text{test-symb} (\text{FImp } \varphi \ \psi) \wedge \text{all-subformula-st test-symb } \varphi \wedge \text{all-subformula-st test-symb } \psi)$
 $\langle \text{proof} \rangle$

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

lemma *subformula-all-subformula-st*:

$\psi \preceq \varphi \implies \text{all-subformula-st test-symb } \varphi \implies \text{all-subformula-st test-symb } \psi$
 $\langle \text{proof} \rangle$

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

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

fixes $r :: 'v \text{ propo} \Rightarrow 'v \text{ propo} \Rightarrow \text{bool}$ **and** *test-symb*: $'v \text{ propo} \Rightarrow \text{bool}$ **and** $x :: 'v$
and $\varphi :: 'v \text{ propo}$
assumes *test-symb-false-nullary*: $\forall x. \text{test-symb } FF \wedge \text{test-symb } FT \wedge \text{test-symb } (FVar \ x)$
and $\forall \varphi'. \varphi' \preceq \varphi \longrightarrow (\neg \text{test-symb } \varphi') \longrightarrow (\exists \psi. r \ \varphi' \ \psi)$ **and**
 $\neg \text{all-subformula-st test-symb } \varphi$
shows $(\exists \psi \ \psi'. \psi \preceq \varphi \wedge r \ \psi \ \psi')$
 $\langle \text{proof} \rangle$

7.2 Invariant conservation

If two rewrite relation are independant (or at least independant enough), then the property characterizing the first relation *all-subformula-st test-symb* remains true. The next show the same property, with changes in the assumptions.

The assumption $\forall \varphi' \ \psi. \varphi' \preceq \Phi \longrightarrow r \ \varphi' \ \psi \longrightarrow \text{all-subformula-st test-symb } \varphi' \longrightarrow \text{all-subformula-st test-symb } \psi$ means that rewriting with r does not mess up the property we want to preserve locally.

The previous assumption is not enough to go from r to *propo-rew-step* r : we have to add the assumption that rewriting inside does not mess up the term: $\forall c \ \xi \ \varphi \ \xi' \ \varphi'. \varphi \preceq \Phi \longrightarrow \text{propo-rew-step } r \ \varphi \ \varphi' \longrightarrow \text{wf-conn } c \ (\xi @ \varphi \ \# \ \xi') \longrightarrow \text{test-symb} (\text{conn } c \ (\xi @ \varphi \ \# \ \xi')) \longrightarrow \text{test-symb } \varphi' \longrightarrow \text{test-symb} (\text{conn } c \ (\xi @ \varphi' \ \# \ \xi'))$

7.2.1 Invariant while lifting of the rewriting relation

The condition $\varphi \preceq \Phi$ (that will be used with $\Phi = \varphi$ most of the time) is here to ensure that the recursive conditions on Φ will moreover hold for the subterm we are rewriting. For example if there is no equivalence symbol in Φ , we do not have to care about equivalence symbols in the two previous assumptions.

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

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

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

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

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

The lemmas can be lifted to *propo-rew-step* r^\perp instead of *propo-rew-step*

7.2.2 Invariant after all rewriting

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

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

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

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

assumes

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

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

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

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

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

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

$\langle \text{proof} \rangle$

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**

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

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

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

$\langle \text{proof} \rangle$

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**

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

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

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

$\langle \text{proof} \rangle$

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.

8 Rewrite Rules

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

8.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

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

elim-equiv[simp]: *elim-equiv* (*FEq* φ ψ) (*FAnd* (*FImp* φ ψ) (*FImp* ψ φ))

lemma *elim-equiv-transformation-consistent*:

$A \models \text{FEq } \varphi \ \psi \longleftrightarrow A \models \text{FAnd } (\text{FImp } \varphi \ \psi) \ (\text{FImp } \psi \ \varphi)$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *elimEquiv-lifted-consistant*:

preserves-un-sat (*full* (*propo-rew-step* *elim-equiv*))

$\langle \text{proof} \rangle$

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

fun *no-equiv-symb* :: 'v *propo* \Rightarrow *bool* **where**

no-equiv-symb (*FEq* -) = *False* |

no-equiv-symb - = *True*

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

lemma *no-equiv-symb-conn-characterization*[simp]:

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

assumes *wf*: *wf-conn* *c* *l*

shows *no-equiv-symb* (*conn* *c* *l*) $\longleftrightarrow c \neq \text{CEq}$

$\langle \text{proof} \rangle$

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

lemma *no-equiv-eq*[simp]:

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

shows

$\neg \text{no-equiv } (\text{FEq } \varphi \ \psi)$

no-equiv *FT*

no-equiv *FF*

$\langle \text{proof} \rangle$

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

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

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

shows

no-equiv (*FNot* φ) $\longleftrightarrow \text{no-equiv } \varphi$

no-equiv (*FAnd* $\varphi \ \psi$) $\longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$

no-equiv (*FOr* $\varphi \ \psi$) $\longleftrightarrow (\text{no-equiv } \varphi \vee \text{no-equiv } \psi)$

no-equiv (*FImp* $\varphi \ \psi$) $\longleftrightarrow (\text{no-equiv } \varphi \wedge \text{no-equiv } \psi)$

$\langle \text{proof} \rangle$

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 φ :: 'v *propo*

assumes *no-equiv*: $\neg \text{no-equiv } \varphi$
shows $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elim-equiv } \psi \psi'$
 $\langle \text{proof} \rangle$

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

lemma *no-equiv-full-propo-rew-step-elim-equiv*:
 $\text{full } (\text{propo-rew-step elim-equiv}) \varphi \psi \implies \text{no-equiv } \psi$
 $\langle \text{proof} \rangle$

8.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* :: '*v propo* \Rightarrow '*v propo* \Rightarrow *bool* **where**
 $[\text{simp}]: \text{elim-imp } (F\text{Imp } \varphi \psi) (F\text{Or } (F\text{Not } \varphi) \psi)$

lemma *elim-imp-transformation-consistent*:
 $A \models F\text{Imp } \varphi \psi \longleftrightarrow A \models F\text{Or } (F\text{Not } \varphi) \psi$
 $\langle \text{proof} \rangle$

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

lemma *elim-imp-consistent*: *preserves-un-sat elim-imp*
 $\langle \text{proof} \rangle$

lemma *elim-imp-lifted-consistent*:
 $\text{preserves-un-sat } (\text{full } (\text{propo-rew-step elim-imp}))$
 $\langle \text{proof} \rangle$

fun *no-imp-symb* **where**
 $\text{no-imp-symb } (F\text{Imp } -) = \text{False} \mid$
 $\text{no-imp-symb } - = \text{True}$

lemma *no-imp-symb-conn-characterization*:
 $\text{wf-conn } c \ l \implies \text{no-imp-symb } (\text{conn } c \ l) \longleftrightarrow c \neq C\text{Imp}$
 $\langle \text{proof} \rangle$

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

lemma *no-imp-Imp* $[\text{simp}]$:
 $\neg \text{no-imp } (F\text{Imp } \varphi \psi)$
 $\text{no-imp } FT$
 $\text{no-imp } FF$
 $\langle \text{proof} \rangle$

lemma *all-subformula-st-decomp-explicit-imp* $[\text{simp}]$:
fixes $\varphi \psi :: 'v \text{ propo}$
shows
 $\text{no-imp } (F\text{Not } \varphi) \longleftrightarrow \text{no-imp } \varphi$
 $\text{no-imp } (F\text{And } \varphi \psi) \longleftrightarrow (\text{no-imp } \varphi \wedge \text{no-imp } \psi)$
 $\text{no-imp } (F\text{Or } \varphi \psi) \longleftrightarrow (\text{no-imp } \varphi \wedge \text{no-imp } \psi)$
 $\langle \text{proof} \rangle$

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:

elim-imp $\varphi \psi \implies \text{no-equiv } \varphi \implies \text{no-equiv } \psi$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

lemma *no-no-imp-elim-imp-step-exists*:

fixes $\varphi :: 'v \text{ propo}$
assumes *no-equiv*: $\neg \text{no-imp } \varphi$
shows $\exists \psi \psi'. \psi \preceq \varphi \wedge \text{elim-imp } \psi \psi'$
 $\langle \text{proof} \rangle$

lemma *no-imp-full-propo-rew-step-elim-imp*: *full* (*propo-rew-step elim-imp*) $\varphi \psi \implies \text{no-imp } \psi$
 $\langle \text{proof} \rangle$

8.3 Eliminate all the True and False in the formula

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

inductive *elimTB* **where**

ElimTB1: *elimTB* (*FAnd* φ *FT*) φ |
ElimTB1': *elimTB* (*FAnd* *FT* φ) φ |

ElimTB2: *elimTB* (*FAnd* φ *FF*) *FF* |
ElimTB2': *elimTB* (*FAnd* *FF* φ) *FF* |

ElimTB3: *elimTB* (*FOr* φ *FT*) *FT* |
ElimTB3': *elimTB* (*FOr* *FT* φ) *FT* |

ElimTB4: *elimTB* (*FOr* φ *FF*) φ |
ElimTB4': *elimTB* (*FOr* *FF* φ) φ |

ElimTB5: *elimTB* (*FNot* *FT*) *FF* |
ElimTB6: *elimTB* (*FNot* *FF*) *FT*

lemma *elimTB-consistent*: *preserves-un-sat elimTB*
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *wf-conn-no-T-F-symb-iff-explicit*[simp]:
 $no-T-F-symb (FAnd \varphi \psi) \longleftrightarrow (\forall \chi \in set [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
 $no-T-F-symb (FOr \varphi \psi) \longleftrightarrow (\forall \chi \in set [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
 $no-T-F-symb (FEq \varphi \psi) \longleftrightarrow (\forall \chi \in set [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
 $no-T-F-symb (FImp \varphi \psi) \longleftrightarrow (\forall \chi \in set [\varphi, \psi]. \chi \neq FF \wedge \chi \neq FT)$
 $\langle proof \rangle$

lemma *no-T-F-symb-false*[simp]:
fixes $c :: 'v$ *connective*
shows
 $\neg no-T-F-symb (FT :: 'v propo)$
 $\neg no-T-F-symb (FF :: 'v propo)$
 $\langle proof \rangle$

lemma *no-T-F-symb-bool*[simp]:
fixes $x :: 'v$
shows $no-T-F-symb (FVar x)$
 $\langle proof \rangle$

lemma *no-T-F-symb-fnot-imp*:
 $\neg no-T-F-symb (FNot \varphi) \implies \varphi = FT \vee \varphi = FF$
 $\langle proof \rangle$

lemma *no-T-F-symb-fnot*[simp]:
 $no-T-F-symb (FNot \varphi) \longleftrightarrow \neg(\varphi = FT \vee \varphi = FF)$
 $\langle proof \rangle$

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 $no-T-F-symb-except-toplevel (FVar x)$
 $\langle proof \rangle$

lemma *no-T-F-symb-except-toplevel-not-decom*:
 $\varphi \neq FT \implies \varphi \neq FF \implies no-T-F-symb-except-toplevel (FNot \varphi)$
 $\langle proof \rangle$

lemma *no-T-F-symb-except-toplevel-bin-decom*:
fixes $\varphi \psi :: 'v propo$
assumes $\varphi \neq FT$ **and** $\varphi \neq FF$ **and** $\psi \neq FT$ **and** $\psi \neq FF$
and $c :: binary-connectives$
shows $no-T-F-symb-except-toplevel (conn c [\varphi, \psi])$
 $\langle proof \rangle$

lemma *no-T-F-symb-except-toplevel-if-is-a-true-false*:
fixes $l :: 'v propo list$ **and** $c :: 'v connective$
assumes $corr :: wf-conn c l$

and $FT \in \text{set } l \vee FF \in \text{set } l$
shows $\neg \text{no-}T\text{-}F\text{-symb-except-toplevel } (\text{conn } c \ l)$
 $\langle \text{proof} \rangle$

lemma *no-}T\text{-}F\text{-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
 $\neg \text{no-}T\text{-}F\text{-symb-except-toplevel } (F\text{And } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-symb-except-toplevel } (F\text{Or } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-symb-except-toplevel } (F\text{Imp } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-symb-except-toplevel } (F\text{Eq } \varphi \ \psi)$
 $\langle \text{proof} \rangle$

lemma *no-}T\text{-}F\text{-symb-except-top-level-false-not[simp]:*
fixes $\varphi \ \psi :: 'v \ \text{propo}$
assumes $\varphi = FT \vee \varphi = FF$
shows
 $\neg \text{no-}T\text{-}F\text{-symb-except-toplevel } (F\text{Not } \varphi)$
 $\langle \text{proof} \rangle$

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

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

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

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

lemma *no-}T\text{-}F\text{-except-top-level-false:*
fixes $l :: 'v \ \text{propo list}$ **and** $c :: 'v \ \text{connective}$
assumes $\text{wf-conn } c \ l$
and $FT \in \text{set } l \vee FF \in \text{set } l$
shows $\neg \text{no-}T\text{-}F\text{-except-top-level } (\text{conn } c \ l)$
 $\langle \text{proof} \rangle$

lemma *no-}T\text{-}F\text{-except-top-level-false-example[simp]:*
fixes $\varphi \ \psi :: 'v \ \text{propo}$
assumes $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
shows
 $\neg \text{no-}T\text{-}F\text{-except-top-level } (F\text{And } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-except-top-level } (F\text{Or } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-except-top-level } (F\text{Eq } \varphi \ \psi)$
 $\neg \text{no-}T\text{-}F\text{-except-top-level } (F\text{Imp } \varphi \ \psi)$
 $\langle \text{proof} \rangle$

lemma *no-}T\text{-}F\text{-symb-except-toplevel-no-}T\text{-}F\text{-symb:*
 $\text{no-}T\text{-}F\text{-symb-except-toplevel } \varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies \text{no-}T\text{-}F\text{-symb } \varphi$
 $\langle \text{proof} \rangle$

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

lemma *no-}T\text{-}F\text{-symb-except-toplevel-all-subformula-st-no-}T\text{-}F\text{-symb:*

no-T-F-except-top-level $\varphi \implies \varphi \neq FF \implies \varphi \neq FT \implies \text{no-T-F } \varphi$
 $\langle \text{proof} \rangle$

lemma *no-T-F-no-T-F-except-top-level*:
no-T-F $\varphi \implies \text{no-T-F-except-top-level } \varphi$
 $\langle \text{proof} \rangle$

lemma *no-T-F-except-top-level-simp[simp]*: *no-T-F-except-top-level* FF *no-T-F-except-top-level* FT
 $\langle \text{proof} \rangle$

lemma *no-T-F-no-T-F-except-top-level'[simp]*:
no-T-F-except-top-level $\varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee \text{no-T-F } \varphi)$
 $\langle \text{proof} \rangle$

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

lemma *no-T-F-bin-decomp-expanded[simp]*:
assumes $c: c = CAnd \vee c = COr \vee c = CEq \vee c = CImp$
shows *no-T-F* (*conn* c $[\varphi, \psi]$) $\longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
 $\langle \text{proof} \rangle$

lemma *no-T-F-comp-expanded-explicit[simp]*:
fixes $\varphi \psi :: 'v \text{ propo}$
shows
no-T-F (*FAnd* $\varphi \psi$) $\longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
no-T-F (*FOr* $\varphi \psi$) $\longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
no-T-F (*FEq* $\varphi \psi$) $\longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
no-T-F (*FImp* $\varphi \psi$) $\longleftrightarrow (\text{no-T-F } \varphi \wedge \text{no-T-F } \psi)$
 $\langle \text{proof} \rangle$

lemma *no-T-F-comp-not[simp]*:
fixes $\varphi \psi :: 'v \text{ propo}$
shows *no-T-F* (*FNot* φ) $\longleftrightarrow \text{no-T-F } \varphi$
 $\langle \text{proof} \rangle$

lemma *no-T-F-decomp*:
fixes $\varphi \psi :: 'v \text{ propo}$
assumes $\varphi: \text{no-T-F } (FAnd \varphi \psi) \vee \text{no-T-F } (FOr \varphi \psi) \vee \text{no-T-F } (FEq \varphi \psi) \vee \text{no-T-F } (FImp \varphi \psi)$
shows *no-T-F* ψ **and** *no-T-F* φ
 $\langle \text{proof} \rangle$

lemma *no-T-F-decomp-not*:
fixes $\varphi :: 'v \text{ propo}$
assumes $\varphi: \text{no-T-F } (FNot \varphi)$
shows *no-T-F* φ
 $\langle \text{proof} \rangle$

lemma *no-T-F-symb-except-toplevel-step-exists*:
fixes $\varphi \psi :: 'v \text{ propo}$
assumes *no-equiv* φ **and** *no-imp* φ
shows $\psi \preceq \varphi \implies \neg \text{no-T-F-symb-except-toplevel } \psi \implies \exists \psi'. \text{elimTB } \psi \psi'$
 $\langle \text{proof} \rangle$

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

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

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

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

$\langle \text{proof} \rangle$

lemma *elimTB-inv*:

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

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

and *no-equiv* φ **and** *no-imp* φ

shows *no-equiv* ψ **and** *no-imp* ψ

$\langle \text{proof} \rangle$

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

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

assumes *no-equiv* φ **and** *no-imp* φ **and** *full* (*propo-rew-step elimTB*) $\varphi \psi$

shows *no-T-F-except-top-level* ψ

$\langle \text{proof} \rangle$

8.4 PushNeg

Push the negation inside the formula, until the litteral.

inductive *pushNeg* **where**

PushNeg1[*simp*]: *pushNeg* (*FNot* (*FAnd* $\varphi \psi$)) (*FOr* (*FNot* φ) (*FNot* ψ)) |

PushNeg2[*simp*]: *pushNeg* (*FNot* (*FOr* $\varphi \psi$)) (*FAnd* (*FNot* φ) (*FNot* ψ)) |

PushNeg3[*simp*]: *pushNeg* (*FNot* (*FNot* φ)) φ

lemma *pushNeg-transformation-consistent*:

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

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

$A \models \text{FNot } (\text{FNot } \varphi) \longleftrightarrow A \models \varphi$

$\langle \text{proof} \rangle$

lemma *pushNeg-explicit*: *pushNeg* $\varphi \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *pushNeg-lifted-consistant*:

preserves-un-sat (*full* (*propo-rew-step pushNeg*))

$\langle \text{proof} \rangle$

fun *simple* **where**

simple *FT* = *True* |

simple *FF* = *True* |

simple (*FVar* $-$) = *True* |

simple $-$ = *False*

lemma *simple-decomp*:

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

$\langle \text{proof} \rangle$

lemma *subformula-conn-decomp-simple*:

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

assumes s : *simple* ψ

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

$\langle \text{proof} \rangle$

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

fixes $\varphi :: 'v \text{ propo}$ **and** $x :: 'v$

shows

$\varphi \preceq \text{FNot } \text{FT} \longleftrightarrow (\varphi = \text{FNot } \text{FT} \vee \varphi = \text{FT})$

$\varphi \preceq \text{FNot } \text{FF} \longleftrightarrow (\varphi = \text{FNot } \text{FF} \vee \varphi = \text{FF})$

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

$\langle \text{proof} \rangle$

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

simple-not-symb ($\text{FNot } \varphi$) = (*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]*:

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

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

$\langle \text{proof} \rangle$

lemma *simple-not-step-exists*:

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

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

no-T-F-symb ($\text{FOr } (\text{FNot } \varphi') (\text{FNot } \psi')$)

no-T-F-symb ($\text{FAnd } (\text{FNot } \varphi') (\text{FNot } \psi')$)

no-T-F-symb ($\text{FNot } (\text{FNot } \varphi')$)

$\langle \text{proof} \rangle$

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

propo-rew-step pushNeg $\varphi \psi \implies \text{no-T-F-except-top-level } \varphi \implies \text{no-T-F-symb } \varphi \implies \text{no-T-F-symb } \psi$
 $\langle \text{proof} \rangle$

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

propo-rew-step pushNeg $\varphi \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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

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

8.5 Push inside

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

for *c c' :: 'v connective* **where**

push-conn-inside-l[simp]: c = CAnd \vee c = COr $\implies c' = CAnd \vee c' = COr$

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

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

push-conn-inside-r[simp]: c = CAnd \vee c = COr $\implies c' = CAnd \vee c' = COr$

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

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

lemma *push-conn-inside-explicit: push-conn-inside c c' $\varphi \psi \implies \forall A. A \models \varphi \longleftrightarrow A \models \psi$*

$\langle \text{proof} \rangle$

lemma *push-conn-inside-consistent: preserves-un-sat (push-conn-inside c c')*

$\langle \text{proof} \rangle$

lemma *propo-rew-step-push-conn-inside[simp]*:

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

$\neg \text{not-}c\text{-in-}c'\text{-symb } c \ c' \ FF$
 $\neg \text{not-}c\text{-in-}c'\text{-symb } c \ c' \ FT$
 $\neg \text{not-}c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ x)$
 $\neg \text{not-}c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ FF)$
 $\neg \text{not-}c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ FT)$
 $\neg \text{not-}c\text{-in-}c'\text{-symb } c \ c' \ (FNot \ (FVar \ x))$
 $\langle \text{proof} \rangle$

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

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

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

$c\text{-in-}c'\text{-only } c \ c' \ FF$
 $c\text{-in-}c'\text{-only } c \ c' \ FT$
 $c\text{-in-}c'\text{-only } c \ c' \ (FVar \ x)$
 $c\text{-in-}c'\text{-only } c \ c' \ (FNot \ FF)$
 $c\text{-in-}c'\text{-only } c \ c' \ (FNot \ FT)$
 $c\text{-in-}c'\text{-only } c \ c' \ (FNot \ (FVar \ x))$
 $\langle \text{proof} \rangle$

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

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

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

$\text{wf-conn } c \ [\varphi, \psi] \implies c\text{-in-}c'\text{-symb } c \ c' \ (\text{conn } c \ [\varphi, \psi]) \longleftrightarrow c\text{-in-}c'\text{-symb } c \ c' \ (\text{conn } c \ [\psi, \varphi])$
 $\langle \text{proof} \rangle$

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

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

lemma $\text{not-}c\text{-in-}c'\text{-simp}[simp]$:

fixes $\varphi1 \ \varphi2 \ \psi :: 'v \ \text{propo}$ **and** $x :: 'v$
shows
 $c\text{-in-}c'\text{-symb } c \ c' \ FT$
 $c\text{-in-}c'\text{-symb } c \ c' \ FF$
 $c\text{-in-}c'\text{-symb } c \ c' \ (FVar \ x)$
 $\text{wf-conn } c \ [\text{conn } c' \ [\varphi1, \varphi2], \psi] \implies \text{wf-conn } c' \ [\varphi1, \varphi2]$
 $\implies \neg c\text{-in-}c'\text{-only } c \ c' \ (\text{conn } c \ [\text{conn } c' \ [\varphi1, \varphi2], \psi])$
 $\langle \text{proof} \rangle$

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

fixes $c\ c' :: 'v\ \text{connective}$ **and** $\psi :: 'v\ \text{propo}$
shows *c-in-c'-symb* $c\ c'$ (*FNot* ψ)

<proof>

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

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

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

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

<proof>

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

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

assumes *noTB*: $\neg\ c\text{-in-c'-only}\ c\ c'\ \varphi$

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>

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>

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

propo-rew-step (*push-conn-inside* $c\ c'$) $\varphi\ \psi \implies \text{simple}\ \varphi \implies \text{simple}\ \psi$

<proof>

lemma *simple-propo-rew-step-inv-push-conn-inside-simple-not*:

fixes $c\ c' :: 'v\ \text{connective}$ **and** $\varphi\ \psi :: 'v\ \text{propo}$

shows *propo-rew-step* (*push-conn-inside* $c\ c'$) $\varphi\ \psi \implies \text{simple-not}\ \varphi \implies \text{simple-not}\ \psi$

<proof>

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* $c\ (\xi @ \varphi' \# \xi')$)

<proof>

lemma *push-conn-inside-not-true-false*:

push-conn-inside $c\ c'\ \varphi\ \psi \implies \psi \neq FT \wedge \psi \neq FF$

<proof>

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>

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* $c \ c' \ \psi$
 $\langle \text{proof} \rangle$

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

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

lemma *only-c-inside-symb-simp*[*simp*]:
only-c-inside-symb $c \ FF$ *only-c-inside-symb* $c \ FT$ *only-c-inside-symb* $c \ (FVar \ x)$ $\langle \text{proof} \rangle$

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

lemma *only-c-inside-symb-decomp*:
only-c-inside-symb $c \ \psi \longleftrightarrow (\text{simple } \psi$
 $\vee (\exists \ \varphi'. \ \psi = FNot \ \varphi' \wedge \text{simple } \varphi')$
 $\vee (\exists \ l. \ \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l))$
 $\langle \text{proof} \rangle$

lemma *only-c-inside-symb-decomp-not*[*simp*]:
fixes $c :: 'v \text{ connective}$
assumes $c \neq CNot$
shows *only-c-inside-symb* $c \ (FNot \ \psi) \longleftrightarrow \text{simple } \psi$
 $\langle \text{proof} \rangle$

lemma *only-c-inside-decomp-not*[*simp*]:
assumes $c \neq CNot$
shows *only-c-inside* $c \ (FNot \ \psi) \longleftrightarrow \text{simple } \psi$
 $\langle \text{proof} \rangle$

lemma *only-c-inside-decomp*:
only-c-inside $c \ \varphi \longleftrightarrow$
 $(\forall \psi. \ \psi \preceq \varphi \longrightarrow (\text{simple } \psi \vee (\exists \ \varphi'. \ \psi = FNot \ \varphi' \wedge \text{simple } \varphi')$
 $\vee (\exists \ l. \ \psi = \text{conn } c \ l \wedge \text{wf-conn } c \ l)))$
 $\langle \text{proof} \rangle$

lemma *only-c-inside-c-c'-false*:
fixes $c \ c' :: 'v \text{ connective}$ **and** $l :: 'v \text{ propo list}$ **and** $\varphi :: 'v \text{ propo}$
assumes cc' : $c \neq c'$ **and** c : $c = CAnd \vee c = COr$ **and** c' : $c' = CAnd \vee c' = COr$

and only: *only-c-inside* $c \varphi$ **and** *incl:* $\text{conn } c' l \preceq \varphi$ **and** *wf:* $\text{wf-conn } c' l$
shows *False*
 $\langle \text{proof} \rangle$

lemma *only-c-inside-implies-c-in-c'-symb:*
assumes $\delta: c \neq c'$ **and** $c: c = \text{CAnd} \vee c = \text{COr}$ **and** $c': c' = \text{CAnd} \vee c' = \text{COr}$
shows *only-c-inside* $c \varphi \implies \text{c-in-c'-symb } c c' \varphi$
 $\langle \text{proof} \rangle$

lemma *c-in-c'-symb-decomp-level1:*
fixes $l :: 'v \text{ propo list}$ **and** $c c' ca :: 'v \text{ connective}$
shows $\text{wf-conn } ca l \implies ca \neq c \implies \text{c-in-c'-symb } c c' (\text{conn } ca l)$
 $\langle \text{proof} \rangle$

lemma *only-c-inside-implies-c-in-c'-only:*
assumes $\delta: c \neq c'$ **and** $c: c = \text{CAnd} \vee c = \text{COr}$ **and** $c': c' = \text{CAnd} \vee c' = \text{COr}$
shows *only-c-inside* $c \varphi \implies \text{c-in-c'-only } c c' \varphi$
 $\langle \text{proof} \rangle$

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

8.5.2 Push Conjunction

definition *pushConj* **where** $\text{pushConj} = \text{push-conn-inside } \text{CAnd } \text{COr}$

lemma *pushConj-consistent:* *preserves-un-sat* pushConj
 $\langle \text{proof} \rangle$

definition *and-in-or-symb* **where** $\text{and-in-or-symb} = \text{c-in-c'-symb } \text{CAnd } \text{COr}$

definition *and-in-or-only* **where**
 $\text{and-in-or-only} = \text{all-subformula-st } (\text{c-in-c'-symb } \text{CAnd } \text{COr})$

lemma *pushConj-inv:*
fixes $\varphi \psi :: 'v \text{ propo}$
assumes *full* $(\text{propo-rew-step } \text{pushConj}) \varphi \psi$
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
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

8.5.3 Push Disjunction

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

lemma *pushDisj-consistent*: *preserves-un-sat pushDisj*
 $\langle \text{proof} \rangle$

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]*:
 $\sim \text{or-in-and-only } (FOr (FAnd \psi1 \psi2) \varphi')$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

9 The full transformations

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

9.1.1 Definition

The normal is a super group of groups

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

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

grouped-by c ($FNot\ FT$)
grouped-by c ($FNot\ FF$)
grouped-by c ($FNot\ (FVar\ x)$)
 $\langle proof \rangle$

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

only-c-inside-symb c ($conn\ c' [\varphi 1, \varphi 2]$) $\implies c' = CAnd \vee c' = COr \implies wf\text{-}conn\ c' [\varphi 1, \varphi 2]$
 $\implies c' = c$
 $\langle proof \rangle$

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

only-c-inside c ($conn\ c' [\varphi 1, \varphi 2]$) $\implies c' = CAnd \vee c' = COr \implies wf\text{-}conn\ c' [\varphi 1, \varphi 2] \implies c = c'$
 $\langle proof \rangle$

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

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

lemma *grouped-by-false*:

grouped-by c ($conn\ c' [\varphi, \psi]$) $\implies c \neq c' \implies wf\text{-}conn\ c' [\varphi, \psi] \implies False$
 $\langle proof \rangle$

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

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

super-grouped-by $c\ c'\ FT$
super-grouped-by $c\ c'\ FF$
super-grouped-by $c\ c' (FVar\ x)$
super-grouped-by $c\ c' (FNot\ FT)$
super-grouped-by $c\ c' (FNot\ FF)$
super-grouped-by $c\ c' (FNot\ (FVar\ x))$
 $\langle proof \rangle$

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

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

9.2 Conjunctive Normal Form

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

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

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

definition *is-cnf* **where**

is-cnf $\varphi \equiv \text{is-conj-with-TF } \varphi \wedge \text{no-T-F-except-top-level } \varphi$

9.2.1 Full CNF transformation

The full1 CNF transformation consists simply in chaining all the transformation defined before.

definition *cnf-rew* **where** *cnf-rew* =

(full (prop-rew-step elim-equiv)) OO
 (full (prop-rew-step elim-imp)) OO
 (full (prop-rew-step elimTB)) OO
 (full (prop-rew-step pushNeg)) OO
 (full (prop-rew-step pushDisj))

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

$\langle \text{proof} \rangle$

lemma *cnf-rew-is-cnf: cnf-rew $\varphi \varphi' \implies \text{is-cnf } \varphi'$*

$\langle \text{proof} \rangle$

9.3 Disjunctive Normal Form

definition *is-disj-with-TF* **where** *is-disj-with-TF* $\equiv \text{super-grouped-by } C\text{And } C\text{Or}$

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

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

$\langle \text{proof} \rangle$

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

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

9.3.1 Full DNF transform

The full1 DNF transformation consists simply in chaining all the transformation defined before.

definition *dnf-rew* **where** *dnf-rew* \equiv

(full (prop-rew-step elim-equiv)) OO
 (full (prop-rew-step elim-imp)) OO
 (full (prop-rew-step elimTB)) OO
 (full (prop-rew-step pushNeg)) OO
 (full (prop-rew-step pushConj))

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

$\langle \text{proof} \rangle$

theorem *dnf-transformation-correction:*

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

$\langle \text{proof} \rangle$

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

10.1 Transformation

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

inductive *elimTBFull* **where**

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

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

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

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

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

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

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

The transformation is still consistent.

lemma *elimTBFull-consistent*: *preserves-un-sat elimTBFull*
 <proof>

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

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

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

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

<proof>

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

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

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

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

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

<proof>

lemma *elimTBFull-full-propo-rew-step*:
fixes $\varphi \psi :: 'v \text{ propo}$
assumes *full (propo-rew-step elimTBFull)* $\varphi \psi$
shows *no-T-F-except-top-level* ψ
 $\langle \text{proof} \rangle$

10.2 More invariants

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

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

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

lemma *propo-rew-step-ElimImp-no-T-F*: *propo-rew-step elim-imp* $\varphi \psi \implies \text{no-T-F } \varphi \implies \text{no-T-F } \psi$
 $\langle \text{proof} \rangle$

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

10.3 The new CNF and DNF transformation

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

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

(*full (propo-rew-step elimTBFull)*) *OO*
(*full (propo-rew-step elim-equiv)*) *OO*
(*full (propo-rew-step elim-imp)*) *OO*
(*full (propo-rew-step pushNeg)*) *OO*
(*full (propo-rew-step pushConj)*)

lemma *dnf-rew'-consistent: preserves-un-sat dnf-rew'*
 $\langle \text{proof} \rangle$

theorem *cnf-transformation-correction*:
dnf-rew' $\varphi \varphi' \implies \text{is-dnf } \varphi'$
 $\langle \text{proof} \rangle$

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

definition *cnf-rew'* :: *'a propo \Rightarrow 'a propo \Rightarrow bool* **where**
cnf-rew' =
(*full (propo-rew-step elimTBFull)*) *OO*

$(full \ (propo\text{-}rew\text{-}step \ elim\text{-}equiv)) \ OO$
 $(full \ (propo\text{-}rew\text{-}step \ elim\text{-}imp)) \ OO$
 $(full \ (propo\text{-}rew\text{-}step \ pushNeg)) \ OO$
 $(full \ (propo\text{-}rew\text{-}step \ pushDisj))$

lemma *cnf-rew'-consistent: preserves-un-sat cnf-rew'*
 $\langle proof \rangle$

theorem *cnf'-transformation-correction:*
 $cnf\text{-}rew' \ \varphi \ \varphi' \implies is\text{-}cnf \ \varphi'$
 $\langle proof \rangle$

end

11 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.

11.1 Clauses

Clauses are (finite) multisets of literals.

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

11.2 Partial Interpretations

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

definition *true-lit :: 'a interp \Rightarrow 'a literal \Rightarrow bool* (**infix** \models_l 50) **where**
 $I \models_l L \longleftrightarrow L \in I$

declare *true-lit-def[simp]*

11.2.1 Consistency

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

lemma *consistent-interp-empty[simp]:*
 $consistent\text{-}interp \ \{\}$ $\langle proof \rangle$

lemma *consistent-interp-single[simp]:*
 $consistent\text{-}interp \ \{L\}$ $\langle proof \rangle$

lemma *consistent-interp-subset:*
assumes
 $A \subseteq B$ **and**

consistent-interp B
shows *consistent-interp A*
 ⟨proof⟩

lemma *consistent-interp-change-insert*:
 $a \notin A \implies -a \notin A \implies \text{consistent-interp } (\text{insert } (-a) A) \longleftrightarrow \text{consistent-interp } (\text{insert } a A)$
 ⟨proof⟩

lemma *consistent-interp-insert-pos[simp]*:
 $a \notin A \implies \text{consistent-interp } (\text{insert } a A) \longleftrightarrow \text{consistent-interp } A \wedge -a \notin A$
 ⟨proof⟩

lemma *consistent-interp-insert-not-in*:
 $\text{consistent-interp } A \implies a \notin A \implies -a \notin A \implies \text{consistent-interp } (\text{insert } a A)$
 ⟨proof⟩

11.2.2 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**
 $\text{atms-of-ms } \psi s = \bigcup (\text{atms-of } ' \psi s)$

lemma *atms-of-mmltiset[simp]*:
 $\text{atms-of } (\text{mset } a) = \text{atm-of } ' \text{ set } a$
 ⟨proof⟩

lemma *atms-of-ms-mset-unfold*:
 $\text{atms-of-ms } (\text{mset } ' b) = (\bigcup x \in b. \text{atm-of } ' \text{ set } x)$
 ⟨proof⟩

definition *atms-of-s* :: 'a literal set \Rightarrow 'a set **where**
 $\text{atms-of-s } C = \text{atm-of } ' C$

lemma *atms-of-ms-empty-set[simp]*:
 $\text{atms-of-ms } \{\} = \{\}$
 ⟨proof⟩

lemma *atms-of-ms-mempty[simp]*:
 $\text{atms-of-ms } \{\{\#\}\} = \{\}$
 ⟨proof⟩

lemma *atms-of-ms-mono*:
 $A \subseteq B \implies \text{atms-of-ms } A \subseteq \text{atms-of-ms } B$
 ⟨proof⟩

lemma *atms-of-ms-finite[simp]*:
 $\text{finite } \psi s \implies \text{finite } (\text{atms-of-ms } \psi s)$
 ⟨proof⟩

lemma *atms-of-ms-union[simp]*:
 $\text{atms-of-ms } (\psi s \cup \chi s) = \text{atms-of-ms } \psi s \cup \text{atms-of-ms } \chi s$
 ⟨proof⟩

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

atms-of-ms (*insert* ψs χs) = *atms-of* $\psi s \cup \text{atms-of-ms } \chi s$
 ⟨proof⟩

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

lemma *atms-of-atms-of-ms-mono*[*simp*]:
 $A \in \psi \implies \text{atms-of } A \subseteq \text{atms-of-ms } \psi$
 ⟨proof⟩

lemma *atms-of-ms-single-set-mset-atms-of*[*simp*]:
atms-of-ms (*single* ‘ *set-mset* B) = *atms-of* B
 ⟨proof⟩

lemma *atms-of-ms-remove-incl*:
shows *atms-of-ms* (*Set.remove* a ψ) \subseteq *atms-of-ms* ψ
 ⟨proof⟩

lemma *atms-of-ms-remove-subset*:
atms-of-ms ($\varphi - \psi$) \subseteq *atms-of-ms* φ
 ⟨proof⟩

lemma *finite-atms-of-ms-remove-subset*[*simp*]:
 $\text{finite } (\text{atms-of-ms } A) \implies \text{finite } (\text{atms-of-ms } (A - C))$
 ⟨proof⟩

lemma *atms-of-ms-empty-iff*:
atms-of-ms $A = \{\}$ $\longleftrightarrow A = \{\{\#\}\} \vee A = \{\}$
 ⟨proof⟩

lemma *in-implies-atm-of-on-atms-of-ms*:
assumes $L \in \# C$ **and** $C \in N$
shows *atm-of* $L \in \text{atms-of-ms } N$
 ⟨proof⟩

lemma *in-plus-implies-atm-of-on-atms-of-ms*:
assumes $C + \{\#L\} \in N$
shows *atm-of* $L \in \text{atms-of-ms } N$
 ⟨proof⟩

lemma *in-m-in-literals*:
assumes $\{\#A\} + D \in \psi s$
shows *atm-of* $A \in \text{atms-of-ms } \psi s$
 ⟨proof⟩

lemma *atms-of-s-union*[*simp*]:
atms-of-s ($Ia \cup Ib$) = *atms-of-s* $Ia \cup \text{atms-of-s } Ib$
 ⟨proof⟩

lemma *atms-of-s-single*[*simp*]:
atms-of-s $\{L\} = \{\text{atm-of } L\}$
 ⟨proof⟩

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

11.2.3 Totality

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

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

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

total-over-m $I\ \psi s = \text{total-over-set } I\ (\text{atms-of-ms } \psi s)$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

lemma *total-over-m-subset*:

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

lemma *total-over-m-extension*:

fixes $I :: 'v\ \text{literal set}$ **and** $A :: 'v\ \text{clauses}$

assumes *total*: *total-over-m* *I* *A*
shows $\exists I'. \text{total-over-m } (I \cup I') (A \cup B)$
 $\wedge (\forall x \in I'. \text{atm-of } x \in \text{atms-of-ms } B \wedge \text{atm-of } x \notin \text{atms-of-ms } A)$
 $\langle \text{proof} \rangle$

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

lemma *total-over-set-atms-of-m[simp]*:
total-over-set *Ia* (*atms-of-s* *Ia*)
 $\langle \text{proof} \rangle$

lemma *total-over-set-literal-defined*:
assumes $\{\#A\# \} + D \in \psi s$
and *total-over-set* *I* (*atms-of-ms* ψs)
shows $A \in I \vee -A \in I$
 $\langle \text{proof} \rangle$

lemma *tot-over-m-remove*:
assumes *total-over-m* $(I \cup \{L\}) \{\psi\}$
and $L: \neg L \in \# \psi - L \notin \# \psi$
shows *total-over-m* *I* $\{\psi\}$
 $\langle \text{proof} \rangle$

lemma *total-union*:
assumes *total-over-m* *I* ψ
shows *total-over-m* $(I \cup I') \psi$
 $\langle \text{proof} \rangle$

lemma *total-union-2*:
assumes *total-over-m* *I* ψ
and *total-over-m* *I'* ψ'
shows *total-over-m* $(I \cup I') (\psi \cup \psi')$
 $\langle \text{proof} \rangle$

11.2.4 Interpretations

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

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

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

lemma *true-cls-union[iff]*: $I \models C + D \longleftrightarrow I \models C \vee I \models D$
 $\langle \text{proof} \rangle$

lemma *true-cls-mono-set-mset*: *set-mset* $C \subseteq \text{set-mset } D \Longrightarrow I \models C \Longrightarrow I \models D$

$\langle proof \rangle$

lemma *true-cls-mono-leD[dest]*: $A \subseteq\# B \implies I \models A \implies I \models B$

$\langle proof \rangle$

lemma

assumes $I \models \psi$

shows

true-cls-union-increase[simp]: $I \cup I' \models \psi$ **and**

true-cls-union-increase'[simp]: $I' \cup I \models \psi$

$\langle proof \rangle$

lemma *true-cls-mono-set-mset-l*:

assumes $A \models \psi$

and $A \subseteq B$

shows $B \models \psi$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *true-cls-not-in-remove*:

assumes $L \notin\# \chi$ **and** $I \cup \{L\} \models \chi$

shows $I \models \chi$

$\langle proof \rangle$

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]*: $I \models_s \{\}$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *true-clss-empty-entails-empty[iff]*: $\{\} \models_s N \longleftrightarrow N = \{\}$

$\langle proof \rangle$

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

$M \models A \implies insert\ L\ M \models A$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *true-clss-insert[iff]*: $I \models_s insert\ C\ DD \longleftrightarrow I \models C \wedge I \models_s DD$

$\langle proof \rangle$

lemma *true-clss-mono*: $DD \subseteq CC \implies I \models_s CC \implies I \models_s DD$

$\langle proof \rangle$

lemma *true-clss-union-increase[simp]*:

assumes $I \models_s \psi$
shows $I \cup I' \models_s \psi$
 $\langle \text{proof} \rangle$

lemma *true-clss-union-increase'*[simp]:

assumes $I' \models_s \psi$
shows $I \cup I' \models_s \psi$
 $\langle \text{proof} \rangle$

lemma *true-clss-commute-l*:

$(I \cup I' \models_s \psi) \longleftrightarrow (I' \cup I \models_s \psi)$
 $\langle \text{proof} \rangle$

lemma *model-remove*[simp]: $I \models_s N \implies I \models_s \text{Set.remove } a \ N$

$\langle \text{proof} \rangle$

lemma *model-remove-minus*[simp]: $I \models_s N \implies I \models_s N - A$

$\langle \text{proof} \rangle$

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

assumes $\forall x \in I'. \text{atm-of } x \notin \text{atms-of-ms } A$
and $\text{atms-of } L \subseteq \text{atms-of-ms } A$
and $I \cup I' \models L$
shows $I \models L$
 $\langle \text{proof} \rangle$

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 $I \models_s L$
 $\langle \text{proof} \rangle$

11.2.5 Satisfiability

definition *satisfiable* :: 'a clause set \Rightarrow bool **where**

satisfiable $CC \equiv \exists I. (I \models_s CC \wedge \text{consistent-interp } I \wedge \text{total-over-m } I \ CC)$

lemma *satisfiable-single*[simp]:

satisfiable $\{\{\#L\#\}\}$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

lemma *satisfiable-def-min*:

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

11.2.6 Entailment for Multisets of Clauses

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

lemma *true-cls-mset-empty[simp]*: $I \models_m \{\#\}$
 $\langle proof \rangle$

lemma *true-cls-mset-singleton[iff]*: $I \models_m \{\# C \#\} \longleftrightarrow I \models C$
 $\langle proof \rangle$

lemma *true-cls-mset-union[iff]*: $I \models_m CC + DD \longleftrightarrow I \models_m CC \wedge I \models_m DD$
 $\langle proof \rangle$

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

lemma *true-cls-mset-mono*: $\text{set-mset } DD \subseteq \text{set-mset } CC \Longrightarrow I \models_m CC \Longrightarrow I \models_m DD$
 $\langle proof \rangle$

lemma *true-clss-set-mset[iff]*: $I \models_s \text{set-mset } CC \longleftrightarrow I \models_m CC$
 $\langle proof \rangle$

lemma *true-cls-mset-increasing-r[simp]*:
 $I \models_m CC \Longrightarrow I \cup J \models_m CC$
 $\langle proof \rangle$

theorem *true-cls-remove-unused*:
assumes $I \models \psi$
shows $\{v \in I. \text{atm-of } v \in \text{atms-of } \psi\} \models \psi$
 $\langle proof \rangle$

theorem *true-clss-remove-unused*:
assumes $I \models_s \psi$
shows $\{v \in I. \text{atm-of } v \in \text{atms-of-ms } \psi\} \models_s \psi$
 $\langle proof \rangle$

A simple application of the previous theorem:

lemma *true-clss-union-decrease*:
assumes $I \cup I' \models \psi$
and $H: \forall v \in I'. \text{atm-of } v \notin \text{atms-of } \psi$
shows $I \models \psi$
 $\langle proof \rangle$

lemma *multiset-not-empty*:
assumes $M \neq \{\#\}$
and $x \in \# M$
shows $\exists A. x = \text{Pos } A \vee x = \text{Neg } A$
 $\langle proof \rangle$

lemma *atms-of-ms-empty*:
fixes $\psi :: 'v \text{ clauses}$
assumes $\text{atms-of-ms } \psi = \{\}$
shows $\psi = \{\} \vee \psi = \{\{\#\}\}$
 $\langle proof \rangle$

lemma *consistent-interp-disjoint*:
assumes *consI*: *consistent-interp I*
and *disj*: *atms-of-s A* \cap *atms-of-s I* = {}
and *consA*: *consistent-interp A*
shows *consistent-interp (A \cup I)*
 \langle *proof* \rangle

lemma *total-remove-unused*:
assumes *total-over-m I ψ*
shows *total-over-m {v \in I. atm-of v \in atms-of-ms ψ } ψ*
 \langle *proof* \rangle

lemma *true-cls-remove-hd-if-notin-vars*:
assumes *insert a M' \models D*
and *atm-of a \notin atms-of D*
shows *M' \models D*
 \langle *proof* \rangle

lemma *total-over-set-atm-of*:
fixes *I :: 'v interp* **and** *K :: 'v set*
shows *total-over-set I K \longleftrightarrow ($\forall l \in K. l \in$ (atm-of ' I))*
 \langle *proof* \rangle

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

lemma *tautology-Pos-Neg[intro]*:
assumes *Pos p $\in \#$ A* **and** *Neg p $\in \#$ A*
shows *tautology A*
 \langle *proof* \rangle

lemma *tautology-minus[simp]*:
assumes *L $\in \#$ A* **and** *$\neg L \in \#$ A*
shows *tautology A*
 \langle *proof* \rangle

lemma *tautology-exists-Pos-Neg*:
assumes *tautology ψ*
shows $\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi$
 \langle *proof* \rangle

lemma *tautology-decomp*:
 $\text{tautology } \psi \longleftrightarrow (\exists p. \text{Pos } p \in \# \psi \wedge \text{Neg } p \in \# \psi)$
 \langle *proof* \rangle

lemma *tautology-false[simp]*: $\neg \text{tautology } \{\#\}$
 \langle *proof* \rangle

lemma *tautology-add-single*:
 $\text{tautology } (\{\#a\} + L) \longleftrightarrow \text{tautology } L \vee \neg a \in \# L$
 \langle *proof* \rangle

lemma *minus-interp-tautology*:
assumes $\{-L \mid L. L \in \# \chi\} \models \chi$
shows *tautology* χ
 $\langle proof \rangle$

lemma *remove-literal-in-model-tautology*:
assumes $I \cup \{Pos\ P\} \models \varphi$
and $I \cup \{Neg\ P\} \models \varphi$
shows $I \models \varphi \vee \text{tautology } \varphi$
 $\langle proof \rangle$

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' $\langle proof \rangle$

11.2.8 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]*:
 $A \models_f A$
 $\langle proof \rangle$

lemma *true-cls-cls-insert-l[simp]*:
 $a \models_f C \implies \text{insert } a \ A \models_p C$
 $\langle proof \rangle$

lemma *true-cls-clss-empty[iff]*:
 $N \models_{fs} \{\}$
 $\langle proof \rangle$

lemma *true-prop-true-clause[iff]*:
 $\{\varphi\} \models_p \psi \longleftrightarrow \varphi \models_f \psi$
 $\langle proof \rangle$

lemma *true-clss-clss-true-clss-cls[iff]*:
 $N \models_{ps} \{\psi\} \longleftrightarrow N \models_p \psi$
 $\langle proof \rangle$

lemma *true-clss-clss-true-cls-clss[iff]*:
 $\{\chi\} \models_{ps} \psi \longleftrightarrow \chi \models_{fs} \psi$
 $\langle proof \rangle$

lemma *true-clss-clss-empty[simp]*:

$N \models_{ps} \{\}$
 $\langle proof \rangle$

lemma *true-clss-clss-subset*:

$A \subseteq B \implies A \models_p CC \implies B \models_p CC$
 $\langle proof \rangle$

lemma *true-clss-clss-mono-l[simp]*:

$A \models_p CC \implies A \cup B \models_p CC$
 $\langle proof \rangle$

lemma *true-clss-clss-mono-l2[simp]*:

$B \models_p CC \implies A \cup B \models_p CC$
 $\langle proof \rangle$

lemma *true-clss-clss-mono-r[simp]*:

$A \models_p CC \implies A \models_p CC + CC'$
 $\langle proof \rangle$

lemma *true-clss-clss-mono-r'[simp]*:

$A \models_p CC' \implies A \models_p CC + CC'$
 $\langle proof \rangle$

lemma *true-clss-clss-union-l[simp]*:

$A \models_{ps} CC \implies A \cup B \models_{ps} CC$
 $\langle proof \rangle$

lemma *true-clss-clss-union-l-r[simp]*:

$B \models_{ps} CC \implies A \cup B \models_{ps} CC$
 $\langle proof \rangle$

lemma *true-clss-clss-in[simp]*:

$CC \in A \implies A \models_p CC$
 $\langle proof \rangle$

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

$A \models_p C \implies insert\ a\ A \models_p C$
 $\langle proof \rangle$

lemma *true-clss-clss-insert-l[iff]*:

$A \models_{ps} C \implies insert\ a\ A \models_{ps} C$
 $\langle proof \rangle$

lemma *true-clss-clss-union-and[iff]*:

$A \models_{ps} C \cup D \longleftrightarrow (A \models_{ps} C \wedge A \models_{ps} D)$
 $\langle proof \rangle$

lemma *true-clss-clss-insert[iff]*:

$A \models_{ps} insert\ L\ Ls \longleftrightarrow (A \models_p L \wedge A \models_{ps} Ls)$
 $\langle proof \rangle$

lemma *true-clss-clss-subset*:

$A \subseteq B \implies A \models_{ps} CC \implies B \models_{ps} CC$
 $\langle proof \rangle$

lemma *union-trus-clss-clss[simp]*: $A \cup B \models_{ps} B$
 $\langle proof \rangle$

lemma *true-clss-clss-remove[simp]*:
 $A \models_{ps} B \implies A \models_{ps} B - C$
 $\langle proof \rangle$

lemma *true-clss-clss-subsetE*:
 $N \models_{ps} B \implies A \subseteq B \implies N \models_{ps} A$
 $\langle proof \rangle$

lemma *true-clss-clss-in-imp-true-clss-clss*:
assumes $N \models_{ps} U$
and $A \in U$
shows $N \models_p A$
 $\langle proof \rangle$

lemma *all-in-true-clss-clss*: $\forall x \in B. x \in A \implies A \models_{ps} B$
 $\langle proof \rangle$

lemma *true-clss-clss-left-right*:
assumes $A \models_{ps} B$
and $A \cup B \models_{ps} M$
shows $A \models_{ps} M \cup B$
 $\langle proof \rangle$

lemma *true-clss-clss-generalise-true-clss-clss*:
 $A \cup C \models_{ps} D \implies B \models_{ps} C \implies A \cup B \models_{ps} D$
 $\langle proof \rangle$

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 $N \models_p D + C$
 $\langle proof \rangle$

lemma *true-clss-clss-union-mset[iff]*: $I \models C \# \cup D \longleftrightarrow I \models C \vee I \models D$
 $\langle proof \rangle$

lemma *true-clss-clss-union-mset-true-clss-clss-or-not-true-clss-clss-or*:
assumes
 $D: N \models_p D + \{\# - L\#\}$ **and**
 $C: N \models_p C + \{\#L\#\}$
shows $N \models_p D \# \cup C$
 $\langle proof \rangle$

lemma *satisfiable-carac[iff]*:
 $(\exists I. \text{consistent-interp } I \wedge I \models_s \varphi) \longleftrightarrow \text{satisfiable } \varphi \text{ (is } (\exists I. ?Q I) \longleftrightarrow ?S)$
 $\langle proof \rangle$

lemma *satisfiable-carac'[simp]*: $\text{consistent-interp } I \implies I \models_s \varphi \implies \text{satisfiable } \varphi$
 $\langle proof \rangle$

11.3 Subsumptions

lemma *subsumption-total-over-m*:

assumes $A \subseteq\# B$

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

$\langle \text{proof} \rangle$

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

$\text{atms-of } (D - \text{replicate-mset } (\text{count } D \ L) \ L) - \text{replicate-mset } (\text{count } D \ (-L)) \ (-L)$

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

$\langle \text{proof} \rangle$

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 $(\forall I. \text{total-over-m } I \{C\} \longrightarrow I \models C \longrightarrow I \models \varphi) \vee \text{tautology } \varphi$

$\langle \text{proof} \rangle$

11.4 Removing Duplicates

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

$\text{tautology } (\text{remdups-mset } C) \longleftrightarrow \text{tautology } C$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *true-clss-clss-remdups-mset[iff]*: $A \models_p \text{remdups-mset } C \longleftrightarrow A \models_p C$

$\langle \text{proof} \rangle$

11.5 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* :: $'v \text{ set} \Rightarrow 'v \text{ clause set}$ **where**

$\text{simple-clss } \text{atms} = \{C. \text{atms-of } C \subseteq \text{atms} \wedge \neg \text{tautology } C \wedge \text{distinct-mset } C\}$

lemma *simple-clss-empty[simp]*:

$\text{simple-clss } \{\} = \{\{\#\}\}$

$\langle \text{proof} \rangle$

lemma *simple-clss-insert*:

assumes $l \notin \text{atms}$


```

shows simple-clss (insert l atms) =
  (op + {#Pos l#}) ‘ (simple-clss atms)
  ∪ (op + {#Neg l#}) ‘ (simple-clss atms)
  ∪ simple-clss atms(is ?I = ?U)
⟨proof⟩

```

```

lemma simple-clss-finite:
  fixes atms :: 'v set
  assumes finite atms
  shows finite (simple-clss atms)
⟨proof⟩

```

```

lemma simple-clssE:
  assumes
    x ∈ simple-clss atms
  shows atms-of x ⊆ atms ∧ ¬tautology x ∧ distinct-mset x
⟨proof⟩

```

```

lemma cls-in-simple-clss:
  shows {#} ∈ simple-clss s
⟨proof⟩

```

```

lemma simple-clss-card:
  fixes atms :: 'v set
  assumes finite atms
  shows card (simple-clss atms) ≤ (3::nat) ^ (card atms)
⟨proof⟩

```

```

lemma simple-clss-mono:
  assumes incl: atms ⊆ atms'
  shows simple-clss atms ⊆ simple-clss atms'
⟨proof⟩

```

```

lemma distinct-mset-not-tautology-implies-in-simple-clss:
  assumes distinct-mset χ and ¬tautology χ
  shows χ ∈ simple-clss (atms-of χ)
⟨proof⟩

```

```

lemma simplified-in-simple-clss:
  assumes distinct-mset-set ψ and ∀ χ ∈ ψ. ¬tautology χ
  shows ψ ⊆ simple-clss (atms-of-ms ψ)
⟨proof⟩

```

11.6 Experiment: Expressing the Entailments as Locales

```

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

```

```

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

```

```

lemma entails-empty[simp]:
  I |=es {}

```

$\langle \text{proof} \rangle$

lemma *entails-single*[*iff*]:

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *entails-insert-mono*: $DD \subseteq CC \implies I \models_{es} CC \implies I \models_{es} DD$

$\langle \text{proof} \rangle$

lemma *entails-union-increase*[*simp*]:

assumes $I \models_{es} \psi$

shows $I \cup I' \models_{es} \psi$

$\langle \text{proof} \rangle$

lemma *true-clss-commute-l*:

$(I \cup I' \models_{es} \psi) \longleftrightarrow (I' \cup I \models_{es} \psi)$

$\langle \text{proof} \rangle$

lemma *entails-remove*[*simp*]: $I \models_{es} N \implies I \models_{es} \text{Set.remove } a \ N$

$\langle \text{proof} \rangle$

lemma *entails-remove-minus*[*simp*]: $I \models_{es} N \implies I \models_{es} N - A$

$\langle \text{proof} \rangle$

end

interpretation *true-cl*s: *entail true-cl*s

$\langle \text{proof} \rangle$

11.7 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

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

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

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

$\langle \text{proof} \rangle$

lemma *true-clss-ext-decrease-right-remove-r*:
assumes $I \models_{\text{sext}} N$
shows $I \models_{\text{sext}} N - \{C\}$
 $\langle \text{proof} \rangle$

lemma *consistent-true-clss-ext-satisfiable*:
assumes *consistent-interp* I **and** $I \models_{\text{sext}} A$
shows *satisfiable* A
 $\langle \text{proof} \rangle$

lemma *not-consistent-true-clss-ext*:
assumes $\neg \text{consistent-interp } I$
shows $I \models_{\text{sext}} A$
 $\langle \text{proof} \rangle$

end

theory *Prop-Logic-Multiset*

imports *../lib/Multiset-More Prop-Normalisation Partial-Clausal-Logic*

begin

12 Link with Multiset Version

12.1 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) = $\{\# \text{ Pos } v \#\}$ |
mset-of-conj (*FNot* (*FVar* v)) = $\{\# \text{ Neg } v \#\}$ |
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* $\varphi \cup \text{mset-of-formula } \psi$ |
mset-of-formula (*FOr* φ ψ) = $\{\text{mset-of-conj } (\text{FOr } \varphi \ \psi)\}$ |
mset-of-formula (*FVar* ψ) = $\{\text{mset-of-conj } (\text{FVar } \psi)\}$ |
mset-of-formula (*FNot* ψ) = $\{\text{mset-of-conj } (\text{FNot } \psi)\}$ |
mset-of-formula *FF* = $\{\{\#\}\}$ |
mset-of-formula *FT* = $\{\}$

12.2 Equisatisfiability of the two Version

lemma *is-conj-with-TF-FNot*:
 $\text{is-conj-with-TF } (\text{FNot } \varphi) \longleftrightarrow (\exists v. \varphi = \text{FVar } v \vee \varphi = \text{FF} \vee \varphi = \text{FT})$
 $\langle \text{proof} \rangle$

lemma *grouped-by-COr-FNot*:
 $\text{grouped-by } \text{COr } (\text{FNot } \varphi) \longleftrightarrow (\exists v. \varphi = \text{FVar } v \vee \varphi = \text{FF} \vee \varphi = \text{FT})$
 $\langle \text{proof} \rangle$

lemma
shows *no-T-F-FF[simp]*: $\neg \text{no-T-F } \text{FF}$ **and**
 no-T-F-FT[simp] : $\neg \text{no-T-F } \text{FT}$
 $\langle \text{proof} \rangle$

lemma *grouped-by-CAnd-FAnd*:
 $\text{grouped-by } \text{CAnd } (\text{FAnd } \varphi_1 \ \varphi_2) \longleftrightarrow \text{grouped-by } \text{CAnd } \varphi_1 \wedge \text{grouped-by } \text{CAnd } \varphi_2$

$\langle \text{proof} \rangle$

lemma *grouped-by-COr-FOr*:

grouped-by COr (FOr $\varphi 1$ $\varphi 2$) \longleftrightarrow grouped-by COr $\varphi 1 \wedge$ grouped-by COr $\varphi 2$

$\langle \text{proof} \rangle$

lemma *grouped-by-COr-FAnd[simp]*: \neg grouped-by COr (FAnd $\varphi 1$ $\varphi 2$)

$\langle \text{proof} \rangle$

lemma *grouped-by-COr-FEq[simp]*: \neg grouped-by COr (FEq $\varphi 1$ $\varphi 2$)

$\langle \text{proof} \rangle$

lemma [simp]: \neg grouped-by COr (FImp φ ψ)

$\langle \text{proof} \rangle$

lemma [simp]: \neg is-conj-with-TF (FImp φ ψ)

$\langle \text{proof} \rangle$

lemma [simp]: \neg grouped-by COr (FEq φ ψ)

$\langle \text{proof} \rangle$

lemma [simp]: \neg is-conj-with-TF (FEq φ ψ)

$\langle \text{proof} \rangle$

lemma *is-conj-with-TF-Fand*:

is-conj-with-TF (FAnd $\varphi 1$ $\varphi 2$) \implies is-conj-with-TF $\varphi 1 \wedge$ is-conj-with-TF $\varphi 2$

$\langle \text{proof} \rangle$

lemma *is-conj-with-TF-FOr*:

is-conj-with-TF (FOr $\varphi 1$ $\varphi 2$) \implies grouped-by COr $\varphi 1 \wedge$ grouped-by COr $\varphi 2$

$\langle \text{proof} \rangle$

lemma *grouped-by-COr-mset-of-formula*:

grouped-by COr $\varphi \implies$ mset-of-formula $\varphi = (\text{if } \varphi = \text{FT then } \{\} \text{ else } \{\text{mset-of-conj } \varphi\})$

$\langle \text{proof} \rangle$

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 $\text{eval } A \varphi \longleftrightarrow \text{Partial-Clausal-Logic.true-clss } (\{\text{Pos } v \mid v. A \ v\} \cup \{\text{Neg } v \mid v. \neg A \ v\})$

$(\text{mset-of-formula } \varphi)$

$\langle \text{proof} \rangle$

end

theory *Prop-Resolution*

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

begin

13 Resolution

13.1 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 $I \models_s N' \longrightarrow I \models_s N$

<proof>

lemma *simplify-preserves-un-sat*:

fixes *N N'* :: 'v clauses

assumes *simplify N N'*

and *total-over-m I N*

shows $I \models_s N \longrightarrow I \models_s N'$

<proof>

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

fixes *N N'* :: 'v clauses

assumes *simplify N N'*

and *total-over-m I N'*

shows $I \models_s N \longrightarrow I \models_s N'$

<proof>

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

fixes *N N'* :: 'v clauses

assumes *simplify N N'*

and *total-over-m I N*

shows $I \models_s N \longleftrightarrow I \models_s N'$

<proof>

lemma *simplify-preserves-finite*:

assumes *simplify $\psi\ \psi'$*

shows *finite $\psi \longleftrightarrow$ finite ψ'*

<proof>

lemma *rtrancpl-simplify-preserves-finite*:

assumes *rtrancpl simplify $\psi\ \psi'$*

shows *finite $\psi \longleftrightarrow$ finite ψ'*

<proof>

lemma *simplify-atms-of-ms*:

assumes *simplify $\psi\ \psi'$*

shows *atms-of-ms $\psi' \subseteq$ atms-of-ms ψ*

<proof>

lemma *rtrancpl-simplify-atms-of-ms*:

assumes *rtrancpl simplify* $\psi \ \psi'$
shows *atms-of-ms* $\psi' \subseteq \text{atms-of-ms } \psi$
 $\langle \text{proof} \rangle$

lemma *factoring-imp-simplify*:
assumes $\{\#L\# \} + \{\#L\# \} + C \in N$
shows $\exists N'. \text{ simplify } N \ N'$
 $\langle \text{proof} \rangle$

13.2 Unconstrained Resolution

type-synonym *'v uncon-state* = *'v clauses*

inductive *uncon-res* :: *'v uncon-state* \Rightarrow *'v uncon-state* \Rightarrow *bool* **where**

resolution:

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

$\Longrightarrow \text{uncon-res } (N) (N \cup \{C + D\}) \mid$

factoring: $\{\#L\# \} + \{\#L\# \} + C \in N \Longrightarrow \text{uncon-res } N (N \cup \{C + \{\#L\# \}\})$

lemma *uncon-res-increasing*:
assumes *uncon-res* $S \ S'$ **and** $\psi \in S$
shows $\psi \in S'$
 $\langle \text{proof} \rangle$

lemma *rtrancpl-uncon-inference-increasing*:
assumes *rtrancpl uncon-res* $S \ S'$ **and** $\psi \in S$
shows $\psi \in S'$
 $\langle \text{proof} \rangle$

13.2.1 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 $\chi \ \chi$
 $\langle \text{proof} \rangle$

lemma *subsumes-subsumption*:
assumes *subsumes* $D \ \chi$
and $C \subset\# D$ **and** $\neg \text{tautology } \chi$
shows *subsumes* $C \ \chi$ $\langle \text{proof} \rangle$

lemma *subsumes-tautology*:
assumes *subsumes* $(C + \{\#Pos \ P\# \} + \{\#Neg \ P\# \}) \ \chi$
shows *tautology* χ
 $\langle \text{proof} \rangle$

13.3 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*)
 $\implies \text{inference } S$ (*fst* $S \cup \{\text{clause}\}$, *already-used*)

abbreviation *already-used-inv*

:: 'a literal multiset set \times ('a literal multiset \times 'a literal multiset) set \Rightarrow bool **where**
already-used-inv state \equiv
 $(\forall (A, B) \in \text{snd state}. \exists p. \text{Pos } p \in \# A \wedge \text{Neg } p \in \# B \wedge$
 $((\exists \chi \in \text{fst state}. \text{subsumes } \chi ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\})))$
 $\vee \text{tautology } ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))))$

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

assumes *inference-clause* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* (*fst* $S \cup \{\text{fst } S'\}$, *snd* S')
 $\langle \text{proof} \rangle$

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

assumes *inference* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
 $\langle \text{proof} \rangle$

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

assumes *rtranclp inference* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
 $\langle \text{proof} \rangle$

lemma *subsumes-condensation:*

assumes *subsumes* $(C + \{\#L\#\} + \{\#L\#\})\ D$
shows *subsumes* $(C + \{\#L\#\})\ D$
 $\langle \text{proof} \rangle$

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

assumes *simplify* $N\ N'$
and *already-used-inv* $(N, \text{already-used})$
shows *already-used-inv* $(N', \text{already-used})$
 $\langle \text{proof} \rangle$

lemma

factoring-satisfiable: $I \models \{\#L\#\} + \{\#L\#\} + C \longleftrightarrow I \models \{\#L\#\} + C$ **and**

resolution-satisfiable:

consistent-interp $I \implies I \models \{\#Pos\ p\#\} + C \implies I \models \{\#Neg\ p\#\} + D \implies I \models C + D$ **and**

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

$\langle \text{proof} \rangle$

lemma *inference-increasing*:

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

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

$\langle \text{proof} \rangle$

lemma *rtrancplp-inference-increasing*:

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

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

$\langle \text{proof} \rangle$

lemma *inference-clause-already-used-increasing*:

assumes *inference-clause* $S S'$

shows $\text{snd } S \subseteq \text{snd } S'$

$\langle \text{proof} \rangle$

lemma *inference-already-used-increasing*:

assumes *inference* $S S'$

shows $\text{snd } S \subseteq \text{snd } S'$

$\langle \text{proof} \rangle$

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 $I \models_s \text{fst } T \longleftrightarrow I \models_s \text{fst } T \cup \{\text{fst } T'\}$

$\langle \text{proof} \rangle$

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 $I \models_s \text{fst } T \longleftrightarrow I \models_s \text{fst } T'$

$\langle \text{proof} \rangle$

lemma *inference-clause-preserves-atms-of-ms*:

assumes *inference-clause* $S S'$

shows $\text{atms-of-ms } (\text{fst } (\text{fst } S \cup \{\text{fst } S'\}, \text{snd } S')) \subseteq \text{atms-of-ms } (\text{fst } S)$

$\langle \text{proof} \rangle$

lemma *inference-preserves-atms-of-ms*:

fixes $N N' :: 'v \text{ clauses}$

assumes *inference* $T T'$

shows $\text{atms-of-ms } (\text{fst } T') \subseteq \text{atms-of-ms } (\text{fst } T)$

$\langle \text{proof} \rangle$

lemma *inference-preserves-total*:

fixes $N N' :: 'v \text{ clauses}$

assumes *inference* $(N, \text{already-used}) (N', \text{already-used}')$

shows $\text{total-over-m } I N \implies \text{total-over-m } I N'$

$\langle \text{proof} \rangle$

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference T T'*
shows *total-over-m I (fst T) \implies total-over-m I (fst T')*
 $\langle proof \rangle$

lemma *rtranclp-inference-preserves-un-sat*:
assumes *rtranclp inference N N'*
and *total-over-m I (fst N)*
and *consistent: consistent-interp I*
shows *I \models_s fst N \longleftrightarrow I \models_s fst N'*
 $\langle proof \rangle$

lemma *inference-preserves-finite*:
assumes *inference ψ ψ' and finite (fst ψ)*
shows *finite (fst ψ')*
 $\langle proof \rangle$

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause ψ ψ' and finite (snd ψ)*
shows *finite (snd ψ')*
 $\langle proof \rangle$

lemma *inference-preserves-finite-snd*:
assumes *inference ψ ψ' and finite (snd ψ)*
shows *finite (snd ψ')*
 $\langle proof \rangle$

lemma *rtranclp-inference-preserves-finite*:
assumes *rtranclp inference ψ ψ' and finite (fst ψ)*
shows *finite (fst ψ')*
 $\langle proof \rangle$

lemma *consistent-interp-insert*:
assumes *consistent-interp I*
and *atm-of P \notin atm-of ' I*
shows *consistent-interp (insert P I)*
 $\langle proof \rangle$

lemma *simplify-clause-preserves-sat*:
assumes *simp: simplify ψ ψ'*
and *satisfiable ψ'*
shows *satisfiable ψ*
 $\langle proof \rangle$

lemma *simplify-preserves-unsat*:
assumes *inference ψ ψ'*
shows *satisfiable (fst ψ') \longrightarrow satisfiable (fst ψ)*
 $\langle proof \rangle$

lemma *inference-preserves-unsat*:

assumes *inference*** *S S'*
shows *satisfiable* (*fst S'*) \longrightarrow *satisfiable* (*fst S*)
 $\langle \text{proof} \rangle$

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*]:
 $(\bigwedge xs :: 'v \text{ sem-tree. } (\bigwedge ys :: 'v \text{ sem-tree. } \text{sem-tree-size } ys < \text{sem-tree-size } xs \implies P \text{ } ys) \implies P \text{ } xs)$
 $\implies P \text{ } xs$
 $\langle \text{proof} \rangle$

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

lemma *simplify-preserve-partial-leaf*:
simplify N N' \implies partial-interps Leaf I N \implies partial-interps Leaf I N'
 $\langle \text{proof} \rangle$

lemma *simplify-preserve-partial-tree*:
assumes *simplify N N'*
and *partial-interps t I N*
shows *partial-interps t I N'*
 $\langle \text{proof} \rangle$

lemma *inference-preserve-partial-tree*:
assumes *inference S S'*
and *partial-interps t I (fst S)*
shows *partial-interps t I (fst S')*
 $\langle \text{proof} \rangle$

lemma *rtranclp-inference-preserve-partial-tree*:
assumes *rtranclp inference N N'*
and *partial-interps t I (fst N)*
shows *partial-interps t I (fst N')*
 $\langle \text{proof} \rangle$

function *build-sem-tree* :: *'v :: linorder set* \Rightarrow *'v clauses* \Rightarrow *'v sem-tree* **where**
build-sem-tree atms ψ =
(if atms = {} \vee \neg finite atms
then Leaf
else Node (Min atms) (build-sem-tree (Set.remove (Min atms) atms) ψ)
(build-sem-tree (Set.remove (Min atms) atms) ψ)
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

declare *build-sem-tree.induct*[*case-names tree*]

lemma *unsatisfiable-empty*[*simp*]:

$\neg \text{unsatisfiable } \{\}$

$\langle \text{proof} \rangle$

lemma *partial-interps-build-sem-tree-atms-general*:

fixes $\psi :: 'v :: \text{linorder clauses}$ **and** $p :: 'v \text{ literal list}$

assumes *unsat*: *unsatisfiable* ψ **and** *finite* ψ **and** *consistent-interp* I
and *finite* *atms*

and *atms-of-ms* $\psi = \text{atms} \cup \text{atms-of-s } I$ **and** $\text{atms} \cap \text{atms-of-s } I = \{\}$

shows *partial-interps* (*build-sem-tree* *atms* ψ) $I \psi$

$\langle \text{proof} \rangle$

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) $\{\} \psi$

$\langle \text{proof} \rangle$

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 $\exists \psi' \chi'. \text{inference}^{**} \psi \psi' \wedge \chi' \in \text{fst } \psi' \wedge (\forall L. L \in \# \chi \longleftrightarrow L \in \# \chi')$
 $\wedge \text{count } \chi' L = 1$
 $\wedge (\forall \varphi. \varphi \in \text{fst } \psi \longrightarrow \varphi \in \text{fst } \psi')$
 $\wedge (I \models \chi \longleftrightarrow I \models \chi')$
 $\wedge (\forall I'. \text{total-over-m } I' \{\chi\} \longrightarrow \text{total-over-m } I' \{\chi'\})$

$\langle \text{proof} \rangle$

lemma *can-decrease-tree-size*:

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)

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

$\langle \text{proof} \rangle$

lemma *inference-completeness-inv*:

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

assumes

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

finite: *finite* (*fst* ψ) **and**

a-u-v: *already-used-inv* ψ

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

$\langle \text{proof} \rangle$

lemma *inference-completeness*:

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

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

and *finite*: *finite* (*fst* ψ)

and $\text{snd } \psi = \{\}$
shows $\exists \psi'. (\text{rtranclp inference } \psi \ \psi' \wedge \{\#\} \in \text{fst } \psi')$
 $\langle \text{proof} \rangle$

lemma *inference-soundness*:
fixes $\psi :: 'v :: \text{linorder state}$
assumes $\text{rtranclp inference } \psi \ \psi' \text{ and } \{\#\} \in \text{fst } \psi'$
shows $\text{unsatisfiable } (\text{fst } \psi)$
 $\langle \text{proof} \rangle$

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

13.4 Lemma about the simplified state

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

lemma *simplified-count*:
assumes $\text{simp: simplified } \psi \text{ and } \chi: \chi \in \psi$
shows $\text{count } \chi \ L \leq 1$
 $\langle \text{proof} \rangle$

lemma *simplified-no-both*:
assumes $\text{simp: simplified } \psi \text{ and } \chi: \chi \in \psi$
shows $\neg (L \in \# \ \chi \wedge \neg L \in \# \ \chi)$
 $\langle \text{proof} \rangle$

lemma *simplified-not-tautology*:
assumes $\text{simplified } \{\psi\}$
shows $\sim \text{tautology } \psi$
 $\langle \text{proof} \rangle$

lemma *simplified-remove*:
assumes $\text{simplified } \{\psi\}$
shows $\text{simplified } \{\psi - \{\#l\# \}\}$
 $\langle \text{proof} \rangle$

lemma *in-simplified-simplified*:
assumes $\text{simp: simplified } \psi \text{ and } \text{incl: } \psi' \subseteq \psi$
shows $\text{simplified } \psi'$
 $\langle \text{proof} \rangle$

lemma *simplified-in*:
assumes $\text{simplified } \psi$
and $N \in \psi$
shows $\text{simplified } \{N\}$
 $\langle \text{proof} \rangle$

lemma *subsumes-imp-formula*:
assumes $\psi \leq \# \ \varphi$
shows $\{\psi\} \models_p \varphi$

⟨proof⟩

lemma *simplified-imp-distinct-mset-tauto*:

assumes *simp*: *simplified* ψ'

shows *distinct-mset-set* ψ' **and** $\forall \chi \in \psi'. \neg \text{tautology } \chi$

⟨proof⟩

lemma *simplified-no-more-full1-simplified*:

assumes *simplified* ψ

shows $\neg \text{full1 simplify } \psi \ \psi'$

⟨proof⟩

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

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

13.5.1 Invariants

lemma *resolution-finite*:

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

shows *finite* (*fst* ψ')

⟨proof⟩

lemma *rtrancpl-resolution-finite*:

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

shows *finite* (*fst* ψ')

⟨proof⟩

lemma *resolution-finite-snd*:

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

shows *finite* (*snd* ψ')

⟨proof⟩

lemma *rtrancpl-resolution-finite-snd*:

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

shows *finite* (*snd* ψ')

⟨proof⟩

lemma *resolution-always-simplified*:

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

shows *simplified* (*fst* ψ')

⟨proof⟩

lemma *trancpl-resolution-always-simplified*:

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

shows *simplified* (*fst* ψ')

⟨proof⟩

lemma *resolution-atms-of*:

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

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

⟨proof⟩

lemma *rtrancpl-resolution-atms-of:*

assumes *resolution*** $\psi \ \psi'$ **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') \subseteq *atms-of-ms* (*fst* ψ)
 \langle *proof* \rangle

lemma *resolution-include:*

assumes *res: resolution* $\psi \ \psi'$ **and** *finite: finite* (*fst* ψ)
shows *fst* $\psi' \subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
 \langle *proof* \rangle

lemma *rtrancpl-resolution-include:*

assumes *res: trancpl resolution* $\psi \ \psi'$ **and** *finite: finite* (*fst* ψ)
shows *fst* $\psi' \subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
 \langle *proof* \rangle

abbreviation *already-used-all-simple*

$:: ('a \text{ literal multiset} \times 'a \text{ literal multiset}) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$ **where**
already-used-all-simple *already-used* *vars* \equiv
 $(\forall (A, B) \in \text{already-used. simplified } \{A\} \wedge \text{simplified } \{B\} \wedge \text{atms-of } A \subseteq \text{vars} \wedge \text{atms-of } B \subseteq \text{vars})$

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

assumes *vars* \subseteq *vars'*
shows *already-used-all-simple* *a vars* \implies *already-used-all-simple* *a vars'*
 \langle *proof* \rangle

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

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

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*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *rtrancpl-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*
 \langle *proof* \rangle

lemma *inference-clause-simplified-already-used-subset*:
assumes *inference-clause* $S S'$
and *simplified* (*fst* S)
shows $\text{snd } S \subset \text{snd } S'$
 $\langle \text{proof} \rangle$

lemma *inference-simplified-already-used-subset*:
assumes *inference* $S S'$
and *simplified* (*fst* S)
shows $\text{snd } S \subset \text{snd } S'$
 $\langle \text{proof} \rangle$

lemma *resolution-simplified-already-used-subset*:
assumes *resolution* $S S'$
and *simplified* (*fst* S)
shows $\text{snd } S \subset \text{snd } S'$
 $\langle \text{proof} \rangle$

lemma *trancpl-resolution-simplified-already-used-subset*:
assumes *trancpl resolution* $S S'$
and *simplified* (*fst* S)
shows $\text{snd } S \subset \text{snd } S'$
 $\langle \text{proof} \rangle$

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

lemma *already-used-all-simple-in-already-used-top*:
assumes *already-used-all-simple* $s \text{ vars}$ **and** *finite vars*
shows $s \subseteq \text{already-used-top vars}$
 $\langle \text{proof} \rangle$

lemma *already-used-top-finite*:
assumes *finite vars*
shows *finite* (*already-used-top vars*)
 $\langle \text{proof} \rangle$

lemma *already-used-top-increasing*:
assumes $\text{var} \subseteq \text{var}'$ **and** *finite var'*
shows *already-used-top var* \subseteq *already-used-top var'*
 $\langle \text{proof} \rangle$

lemma *already-used-all-simple-finite*:
fixes $s :: ('a \text{ literal multiset} \times 'a \text{ literal multiset}) \text{ set}$ **and** $\text{vars} :: 'a \text{ set}$
assumes *already-used-all-simple* $s \text{ vars}$ **and** *finite vars*
shows *finite* s
 $\langle \text{proof} \rangle$

abbreviation *card-simple vars* $\psi \equiv \text{card } (\text{already-used-top vars} - \psi)$

lemma *resolution-card-simple-decreasing*:
assumes *res: resolution* $\psi \psi'$
and *a-u-s: already-used-all-simple* (*snd* ψ) *vars*
and *finite-v: finite vars*
and *finite-fst: finite* (*fst* ψ)

and *finite-snd*: *finite* (*snd* ψ)
and *simp*: *simplified* (*fst* ψ)
and *atms-of-ms* (*fst* ψ) \subseteq *vars*
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
 <proof>

lemma *trancp-resolution-card-simple-decreasing*:
assumes *trancp resolution* ψ ψ' **and** *finite-fst*: *finite* (*fst* ψ)
and *already-used-all-simple* (*snd* ψ) *vars*
and *atms-of-ms* (*fst* ψ) \subseteq *vars*
and *finite-v*: *finite vars*
and *finite-snd*: *finite* (*snd* ψ)
and *simplified* (*fst* ψ)
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
 <proof>

lemma *trancp-resolution-card-simple-decreasing-2*:
assumes *trancp resolution* ψ ψ'
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>

13.5.2 well-foundness if the relation

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

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

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

lemma *rtrancp-simplify-already-used-inv*:
assumes *simplify*** $S \ S'$
and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
 <proof>

lemma *full1-simplify-already-used-inv*:
assumes *full1 simplify* $S \ S'$

and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
 $\langle \text{proof} \rangle$

lemma *full-simplify-already-used-inv*:
assumes *full simplify* $S S'$
and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
 $\langle \text{proof} \rangle$

lemma *resolution-already-used-inv*:
assumes *resolution* $S S'$
and *already-used-inv* S
shows *already-used-inv* S'
 $\langle \text{proof} \rangle$

lemma *rtranclp-resolution-already-used-inv*:
assumes *resolution*** $S S'$
and *already-used-inv* S
shows *already-used-inv* S'
 $\langle \text{proof} \rangle$

lemma *rtanclp-simplify-preserves-unsat*:
assumes *simplify*** $\psi \psi'$
shows *satisfiable* $\psi' \longrightarrow \text{satisfiable } \psi$
 $\langle \text{proof} \rangle$

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* $\psi \psi'$
shows *satisfiable* $\psi' \longrightarrow \text{satisfiable } \psi$
 $\langle \text{proof} \rangle$

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* $\psi \psi'$
shows *satisfiable* $\psi' \longrightarrow \text{satisfiable } \psi$
 $\langle \text{proof} \rangle$

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

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

lemma *rtranclp-simplify-preserve-partial-tree*:
assumes *simplify*** $N N'$
and *partial-interps* $t I N$
shows *partial-interps* $t I N'$
 $\langle \text{proof} \rangle$

lemma *full1-simplify-preserve-partial-tree*:
assumes *full1 simplify* $N N'$
and *partial-interps* $t I N$

shows *partial-interps* t I N'
 $\langle \text{proof} \rangle$

lemma *full-simplify-preserve-partial-tree*:
assumes *full simplify* N N'
and *partial-interps* t I N
shows *partial-interps* t I N'
 $\langle \text{proof} \rangle$

lemma *resolution-preserve-partial-tree*:
assumes *resolution* S S'
and *partial-interps* t I (*fst* S)
shows *partial-interps* t I (*fst* S')
 $\langle \text{proof} \rangle$

lemma *rtrancp-resolution-preserve-partial-tree*:
assumes *resolution*** S S'
and *partial-interps* t I (*fst* S)
shows *partial-interps* t I (*fst* S')
 $\langle \text{proof} \rangle$
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 $P \ n$
 $\langle \text{proof} \rangle$

lemma *wf-always-more-step-False*:
assumes *wf* R
shows $(\forall x. \exists z. (z, x) \in R) \implies \text{False}$
 $\langle \text{proof} \rangle$

lemma *finite-finite-mset-element-of-mset*[*simp*]:
assumes *finite* N
shows *finite* $\{f \ \varphi \ L \mid \varphi \ L. \ \varphi \in N \wedge L \in \# \ \varphi \wedge P \ \varphi \ L\}$
 $\langle \text{proof} \rangle$

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

syntax
 $\text{-comprehension1'-mset} :: 'a \Rightarrow 'b \Rightarrow 'b \text{ multiset} \Rightarrow 'a \text{ multiset}$
 $((\{\# \text{-} / . \text{-} : \text{setof -}\#\}))$

translations
 $\{\# e. x. \text{setof } M \#\} == \text{CONST set-mset } (\text{CONST image-mset } (\%x. e) \ M)$
value $\{\# a. a : \text{setof } \{\# 1, 1, 2 :: \text{int}\} \#\} = \{1, 2\}$

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

interpretation *sum-count-ge-2*:

folding $(\lambda\varphi. op + (msetsum \{ \#count \varphi L \mid L \in \# \varphi. 2 \leq count \varphi L \# \})) 0$

rewrites

folding.F $(\lambda\varphi. op + (msetsum \{ \#count \varphi L \mid L \in \# \varphi. 2 \leq count \varphi L \# \})) 0 = sum-count-ge-2$

<proof>

lemma *finite-incl-le-setsum*:

finite $(B :: 'a \text{ multiset set}) \implies A \subseteq B \implies \Xi A \leq \Xi B$

<proof>

lemma *simplify-finite-measure-decrease*:

simplify $N N' \implies \text{finite } N \implies card N' + \Xi N' < card N + \Xi N$

<proof>

lemma *simplify-terminates*:

wf $\{(N', N). \text{finite } N \wedge \text{simplify } N N'\}$

<proof>

lemma *wf-terminates*:

assumes *wf* *r*

shows $\exists N'. (N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r)$

<proof>

lemma *rtranclp-simplify-terminates*:

assumes *fin*: *finite* *N*

shows $\exists N'. \text{simplify}^{**} N N' \wedge \text{simplified } N'$

<proof>

lemma *finite-simplified-full1-simp*:

assumes *finite* *N*

shows $\text{simplified } N \vee (\exists N'. \text{full1 simplify } N N')$

<proof>

lemma *finite-simplified-full-simp*:

assumes *finite* *N*

shows $\exists N'. \text{full simplify } N N'$

<proof>

lemma *can-decrease-tree-size-resolution*:

fixes $\psi :: 'v \text{ state}$ **and** $\text{tree} :: 'v \text{ sem-tree}$

assumes *finite* $(fst \psi)$ **and** *already-used-inv* ψ

and *partial-interps* *tree* *I* $(fst \psi)$

and *simplified* $(fst \psi)$

shows $\exists (\text{tree}' :: 'v \text{ sem-tree}) \psi'. \text{resolution}^{**} \psi \psi' \wedge \text{partial-interps } \text{tree}' I (fst \psi')$

$\wedge (\text{sem-tree-size } \text{tree}' < \text{sem-tree-size } \text{tree} \vee \text{sem-tree-size } \text{tree} = 0)$

<proof>

lemma *resolution-completeness-inv*:

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

assumes

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

finite: *finite* $(fst \psi)$ **and**

a-u-v: *already-used-inv* ψ

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

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

assumes *resolution* $S S'$
and *already-used-inv* S
shows *already-used-inv* S'
 $\langle \text{proof} \rangle$

lemma *rtrancpl-resolution-preserves-already-used-inv*:

assumes *resolution*^{**} $S S'$
and *already-used-inv* S
shows *already-used-inv* S'
 $\langle \text{proof} \rangle$

lemma *resolution-completeness*:

fixes $\psi :: 'v :: \text{linorder state}$
assumes *unsat*: $\neg \text{satisfiable } (\text{fst } \psi)$
and *finite*: *finite* $(\text{fst } \psi)$
and *snd* $\psi = \{\}$
shows $\exists \psi'. (\text{resolution}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')$
 $\langle \text{proof} \rangle$

lemma *rtrancpl-preserves-sat*:

assumes *simplify*^{**} $S S'$
and *satisfiable* S
shows *satisfiable* S'
 $\langle \text{proof} \rangle$

lemma *resolution-preserves-sat*:

assumes *resolution* $S S'$
and *satisfiable* $(\text{fst } S)$
shows *satisfiable* $(\text{fst } S')$
 $\langle \text{proof} \rangle$

lemma *rtrancpl-resolution-preserves-sat*:

assumes *resolution*^{**} $S S'$
and *satisfiable* $(\text{fst } S)$
shows *satisfiable* $(\text{fst } S')$
 $\langle \text{proof} \rangle$

lemma *resolution-soundness*:

fixes $\psi :: 'v :: \text{linorder state}$
assumes *resolution*^{**} $\psi \psi'$ **and** $\{\#\} \in \text{fst } \psi'$
shows *unsatisfiable* $(\text{fst } \psi)$
 $\langle \text{proof} \rangle$

lemma *resolution-soundness-and-completeness*:

fixes $\psi :: 'v :: \text{linorder state}$
assumes *finite*: *finite* $(\text{fst } \psi)$
and *snd*: *snd* $\psi = \{\}$
shows $(\exists \psi'. (\text{resolution}^{**} \psi \psi' \wedge \{\#\} \in \text{fst } \psi')) \longleftrightarrow \text{unsatisfiable } (\text{fst } \psi)$
 $\langle \text{proof} \rangle$

lemma *simplified-falsity*:

```

assumes simp: simplified  $\psi$ 
and  $\{\#\} \in \psi$ 
shows  $\psi = \{\{\#\}\}$ 
 $\langle proof \rangle$ 

```

lemma *simplify-falsity-in-preserved*:

```

assumes simplify  $\chi s \chi s'$ 
and  $\{\#\} \in \chi s$ 
shows  $\{\#\} \in \chi s'$ 
 $\langle proof \rangle$ 

```

lemma *rtrancp-simplify-falsity-in-preserved*:

```

assumes simplify**  $\chi s \chi s'$ 
and  $\{\#\} \in \chi s$ 
shows  $\{\#\} \in \chi s'$ 
 $\langle proof \rangle$ 

```

lemma *resolution-falsity-get-falsity-alone*:

```

assumes finite (fst  $\psi$ )
shows  $(\exists \psi'. (resolution^{**} \psi \psi' \wedge \{\#\} \in fst \psi')) \longleftrightarrow (\exists a-u-v. resolution^{**} \psi (\{\{\#\}\}, a-u-v))$ 
  (is  $?A \longleftrightarrow ?B$ )
 $\langle proof \rangle$ 

```

lemma *resolution-soundness-and-completeness'*:

```

fixes  $\psi :: 'v :: linorder \text{ state}$ 
assumes
  finite: finite (fst  $\psi$ ) and
  snd: snd  $\psi = \{\}$ 
shows  $(\exists a-u-v. (resolution^{**} \psi (\{\{\#\}\}, a-u-v))) \longleftrightarrow unsatisfiable (fst \psi)$ 
 $\langle proof \rangle$ 

```

end

14 Partial Clausal Logic

We here define marked 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

14.1 Marked Literals

14.1.1 Definition

```

datatype ('v, 'vl, 'mark) marked-lit =
  is-marked: Marked (lit-of: 'v literal) (level-of: 'vl) |
  is-proped: Propagated (lit-of: 'v literal) (mark-of: 'mark)

```

lemma *marked-lit-list-induct*[*case-names nil marked proped*]:

```

assumes  $P []$  and
 $\bigwedge L l xs. P xs \implies P (\text{Marked } L l \# xs)$  and
 $\bigwedge L m xs. P xs \implies P (\text{Propagated } L m \# xs)$ 

```

shows $P \text{ } xs$
 $\langle proof \rangle$

lemma *is-marked-ex-Marked*:
 $is\text{-}marked \ L \implies \exists K \text{ } lvl. \ L = \text{Marked } K \text{ } lvl$
 $\langle proof \rangle$

type-synonym $('v, 'l, 'm) \text{ } marked\text{-}lits = ('v, 'l, 'm) \text{ } marked\text{-}lit \text{ } list$

definition $lits\text{-}of :: ('a, 'b, 'c) \text{ } marked\text{-}lit \text{ } set \Rightarrow 'a \text{ } literal \text{ } set$ **where**
 $lits\text{-}of \ Ls = lit\text{-}of \text{ } ' Ls$

abbreviation $lits\text{-}of\text{-}l :: ('a, 'b, 'c) \text{ } marked\text{-}lit \text{ } list \Rightarrow 'a \text{ } literal \text{ } set$ **where**
 $lits\text{-}of\text{-}l \ Ls \equiv lits\text{-}of \ (set \ Ls)$

lemma *lits-of-l-empty[simp]*:
 $lits\text{-}of \ \{\} = \{\}$
 $\langle proof \rangle$

lemma *lits-of-insert[simp]*:
 $lits\text{-}of \ (insert \ L \ Ls) = insert \ (lit\text{-}of \ L) \ (lits\text{-}of \ Ls)$
 $\langle proof \rangle$

lemma *lits-of-l-Un[simp]*:
 $lits\text{-}of \ (l \cup l') = lits\text{-}of \ l \cup lits\text{-}of \ l'$
 $\langle proof \rangle$

lemma *finite-lits-of-def[simp]*:
 $finite \ (lits\text{-}of\text{-}l \ L)$
 $\langle proof \rangle$

abbreviation *unmark* **where**
 $unmark \equiv (\lambda a. \ \{\#lit\text{-}of \ a\#\})$

abbreviation *unmark-s* **where**
 $unmark\text{-}s \ M \equiv unmark \text{ } ' M$

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]*:
 $atms\text{-}of\text{-}ms \ (unmark\text{-}l \ M') = atm\text{-}of \text{ } ' lits\text{-}of\text{-}l \ M'$
 $\langle proof \rangle$

lemma *lits-of-l-empty-is-empty[iff]*:
 $lits\text{-}of\text{-}l \ M = \{\} \longleftrightarrow M = []$
 $\langle proof \rangle$

14.1.2 Entailment

definition $true\text{-}annot :: ('a, 'l, 'm) \text{ } marked\text{-}lits \Rightarrow 'a \text{ } clause \Rightarrow bool$ (**infix** \models_a 49) **where**
 $I \models_a C \longleftrightarrow (lits\text{-}of\text{-}l \ I) \models C$

definition $true\text{-}annots :: ('a, 'l, 'm) \text{ } marked\text{-}lits \Rightarrow 'a \text{ } clauses \Rightarrow bool$ (**infix** \models_{as} 49) **where**
 $I \models_{as} CC \longleftrightarrow (\forall C \in CC. \ I \models_a C)$

lemma *true-annot-empty-model*[simp]:

$\neg[] \models_a \psi$
 $\langle \text{proof} \rangle$

lemma *true-annot-empty*[simp]:

$\neg I \models_a \{\#\}$
 $\langle \text{proof} \rangle$

lemma *empty-true-annots-def*[iff]:

$[] \models_{as} \psi \longleftrightarrow \psi = \{\}$
 $\langle \text{proof} \rangle$

lemma *true-annots-empty*[simp]:

$I \models_{as} \{\}$
 $\langle \text{proof} \rangle$

lemma *true-annots-single-true-annot*[iff]:

$I \models_{as} \{C\} \longleftrightarrow I \models_a C$
 $\langle \text{proof} \rangle$

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

$M \models_a A \implies L \# M \models_a A$
 $\langle \text{proof} \rangle$

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

$M \models_{as} A \implies L \# M \models_{as} A$
 $\langle \text{proof} \rangle$

lemma *true-annots-union*[iff]:

$M \models_{as} A \cup B \longleftrightarrow (M \models_{as} A \wedge M \models_{as} B)$
 $\langle \text{proof} \rangle$

lemma *true-annots-insert*[iff]:

$M \models_{as} \text{insert } a \ A \longleftrightarrow (M \models_a a \wedge M \models_{as} A)$
 $\langle \text{proof} \rangle$

Link between \models_{as} and \models_s :

lemma *true-annots-true-clss*:

$I \models_{as} CC \longleftrightarrow \text{lits-of-l } I \models_s CC$
 $\langle \text{proof} \rangle$

lemma *in-lit-of-true-annot*:

$a \in \text{lits-of-l } M \longleftrightarrow M \models_a \{\#a\#\}$
 $\langle \text{proof} \rangle$

lemma *true-annot-lit-of-notin-skip*:

$L \# M \models_a A \implies \text{lit-of } L \notin \# A \implies M \models_a A$
 $\langle \text{proof} \rangle$

lemma *true-clss-singleton-lit-of-implies-incl*:

$I \models_s \text{unmark-l } MLs \implies \text{lits-of-l } MLs \subseteq I$
 $\langle \text{proof} \rangle$

lemma *true-annot-true-clss-clss*:

$MLs \models_a \psi \implies \text{set } (\text{map } \text{unmark } MLs) \models_p \psi$
 $\langle \text{proof} \rangle$

lemma *true-annots-true-clss-clss*:

$MLs \models_{as} \psi \implies \text{set } (\text{map } \text{unmark } MLs) \models_{ps} \psi$
 $\langle \text{proof} \rangle$

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

$\text{map } (\lambda M. \text{Marked } M \ a) \ M \models_{as} N \iff \text{set } M \models_s N$
 $\langle \text{proof} \rangle$

lemma *true-annot-singleton[iff]*: $M \models_a \{\#L\# \} \iff L \in \text{lits-of-l } M$

$\langle \text{proof} \rangle$

lemma *true-annots-true-clss-clss*:

$A \models_{as} \Psi \implies \text{unmark-l } A \models_{ps} \Psi$
 $\langle \text{proof} \rangle$

lemma *true-annot-commute*:

$M \ @ \ M' \models_a D \iff M' \ @ \ M \models_a D$
 $\langle \text{proof} \rangle$

lemma *true-annots-commute*:

$M \ @ \ M' \models_{as} D \iff M' \ @ \ M \models_{as} D$
 $\langle \text{proof} \rangle$

lemma *true-annot-mono[dest]*:

$\text{set } I \subseteq \text{set } I' \implies I \models_a N \implies I' \models_a N$
 $\langle \text{proof} \rangle$

lemma *true-annots-mono*:

$\text{set } I \subseteq \text{set } I' \implies I \models_{as} N \implies I' \models_{as} N$
 $\langle \text{proof} \rangle$

14.1.3 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 marked literals (aka a trail in most cases).

Remark that *undefined* already exists and is a completely different Isabelle function.

definition *defined-lit* :: $('a, 'l, 'm) \text{ marked-lits} \Rightarrow 'a \text{ literal} \Rightarrow \text{bool}$

where

$\text{defined-lit } I \ L \iff (\exists l. \text{Marked } L \ l \in \text{set } I) \vee (\exists P. \text{Propagated } L \ P \in \text{set } I)$
 $\vee (\exists l. \text{Marked } (-L) \ l \in \text{set } I) \vee (\exists P. \text{Propagated } (-L) \ P \in \text{set } I)$

abbreviation *undefined-lit* :: $('a, 'l, 'm) \text{ marked-lit list} \Rightarrow 'a \text{ literal} \Rightarrow \text{bool}$

where $\text{undefined-lit } I \ L \equiv \neg \text{defined-lit } I \ L$

lemma *defined-lit-rev[simp]*:

$\text{defined-lit } (\text{rev } M) \ L \iff \text{defined-lit } M \ L$
 $\langle \text{proof} \rangle$

lemma *atm-imp-marked-or-proped*:

assumes $x \in \text{set } I$

shows

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

lemma *literal-is-lit-of-marked*:

assumes $L = \text{lit-of } x$
shows $(\exists l. x = \text{Marked } L \ l) \vee (\exists l'. x = \text{Propagated } L \ l')$
 $\langle \text{proof} \rangle$

lemma *true-annot-iff-marked-or-true-lit*:

$\text{defined-lit } I \ L \longleftrightarrow (\text{lits-of-l } I \models L \vee \text{lits-of-l } I \models \neg L)$
 $\langle \text{proof} \rangle$

lemma *consistent-inter-true-annot-satisfiable*:

$\text{consistent-interp } (\text{lits-of-l } I) \implies I \models_{\text{as}} N \implies \text{satisfiable } N$
 $\langle \text{proof} \rangle$

lemma *defined-lit-map*:

$\text{defined-lit } Ls \ L \longleftrightarrow \text{atm-of } L \in (\lambda l. \text{atm-of } (\text{lit-of } l)) \text{ ' set } Ls$
 $\langle \text{proof} \rangle$

lemma *defined-lit-uminus[iff]*:

$\text{defined-lit } I \ (\neg L) \longleftrightarrow \text{defined-lit } I \ L$
 $\langle \text{proof} \rangle$

lemma *Marked-Propagated-in-iff-in-lits-of-l*:

$\text{defined-lit } I \ L \longleftrightarrow (L \in \text{lits-of-l } I \vee \neg L \in \text{lits-of-l } I)$
 $\langle \text{proof} \rangle$

lemma *consistent-add-undefined-lit-consistent[simp]*:

assumes
 $\text{consistent-interp } (\text{lits-of-l } Ls)$ **and**
 $\text{undefined-lit } Ls \ L$
shows $\text{consistent-interp } (\text{insert } L \ (\text{lits-of-l } Ls))$
 $\langle \text{proof} \rangle$

lemma *decided-empty[simp]*:

$\neg \text{defined-lit } [] \ L$
 $\langle \text{proof} \rangle$

14.2 Backtracking

fun *backtrack-split* :: $('v, 'l, 'm) \text{ marked-lits}$

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

$\text{backtrack-split } [] = ([], [])$ |

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

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

lemma *backtrack-split-fst-not-marked*: $a \in \text{set } (\text{fst } (\text{backtrack-split } l)) \implies \neg \text{is-marked } a$

$\langle \text{proof} \rangle$

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

$\text{snd } (\text{backtrack-split } l) \neq [] \implies \text{is-marked } (\text{hd } (\text{snd } (\text{backtrack-split } l)))$

$\langle \text{proof} \rangle$

lemma *backtrack-split-list-eq[simp]*:
 $\text{fst } (\text{backtrack-split } l) @ (\text{snd } (\text{backtrack-split } l)) = l$
 $\langle \text{proof} \rangle$

lemma *backtrack-snd-empty-not-marked*:
 $\text{backtrack-split } M = (M'', []) \implies \forall l \in \text{set } M. \neg \text{is-marked } l$
 $\langle \text{proof} \rangle$

lemma *backtrack-split-some-is-marked-then-snd-has-hd*:
 $\exists l \in \text{set } M. \text{is-marked } l \implies \exists M' L' M''. \text{backtrack-split } M = (M'', L' \# M')$
 $\langle \text{proof} \rangle$

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma *backtrack-split-takeWhile-dropWhile*:
 $\text{backtrack-split } M = (\text{takeWhile } (\text{Not } o \text{ is-marked}) M, \text{dropWhile } (\text{Not } o \text{ is-marked}) M)$
 $\langle \text{proof} \rangle$

14.3 Decomposition with respect to the First Marked Literals

In this section we define a function that returns a decomposition with the first marked literal. This function is useful to define the backtracking of DPLL.

14.3.1 Definition

The pattern *get-all-marked-decomposition* $[] = [([], [])]$ is necessary otherwise, we can call the *hd* function in the other pattern.

fun *get-all-marked-decomposition* :: ('a, 'l, 'm) marked-lits
 $\implies ((('a, 'l, 'm) \text{ marked-lits} \times ('a, 'l, 'm) \text{ marked-lits}) \text{ list } \mathbf{where}$
 $\text{get-all-marked-decomposition } (\text{Marked } L \ l \ \# \ Ls) =$
 $(\text{Marked } L \ l \ \# \ Ls, []) \ \# \ \text{get-all-marked-decomposition } Ls \mid$
 $\text{get-all-marked-decomposition } (\text{Propagated } L \ P \ \# \ Ls) =$
 $(\text{apsnd } ((op \ \#) (\text{Propagated } L \ P)) (\text{hd } (\text{get-all-marked-decomposition } Ls)))$
 $\ \# \ \text{tl } (\text{get-all-marked-decomposition } Ls) \mid$
 $\text{get-all-marked-decomposition } [] = [([], [])]$

value *get-all-marked-decomposition* [Propagated A5 B5, Marked C4 D4, Propagated A3 B3,
Propagated A2 B2, Marked C1 D1, Propagated A0 B0]

Now we can prove several simple properties about the function.

lemma *get-all-marked-decomposition-never-empty[iff]*:
 $\text{get-all-marked-decomposition } M = [] \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-never-empty-sym[iff]*:
 $[] = \text{get-all-marked-decomposition } M \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-decomp*:
 $\text{hd } (\text{get-all-marked-decomposition } S) = (a, c) \implies S = c @ a$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-backtrack-split:*

backtrack-split $S = (M, M') \longleftrightarrow \text{hd } (\text{get-all-marked-decomposition } S) = (M', M)$
 ⟨proof⟩

lemma *get-all-marked-decomposition-nil-backtrack-split-snd-nil:*

get-all-marked-decomposition $S = [([], A)] \implies \text{snd } (\text{backtrack-split } S) = []$
 ⟨proof⟩

This functions says that the first element is either empty or starts with a marked element of the list.

lemma *get-all-marked-decomposition-length-1-fst-empty-or-length-1:*

assumes *get-all-marked-decomposition* $M = (a, b) \# []$
shows $a = [] \vee (\text{length } a = 1 \wedge \text{is-marked } (\text{hd } a) \wedge \text{hd } a \in \text{set } M)$
 ⟨proof⟩

lemma *get-all-marked-decomposition-fst-empty-or-hd-in-M:*

assumes *get-all-marked-decomposition* $M = (a, b) \# l$
shows $a = [] \vee (\text{is-marked } (\text{hd } a) \wedge \text{hd } a \in \text{set } M)$
 ⟨proof⟩

lemma *get-all-marked-decomposition-snd-not-marked:*

assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
and $L \in \text{set } b$
shows $\neg \text{is-marked } L$
 ⟨proof⟩

lemma *tl-get-all-marked-decomposition-skip-some:*

assumes $x \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } M1))$
shows $x \in \text{set } (\text{tl } (\text{get-all-marked-decomposition } (M0 @ M1)))$
 ⟨proof⟩

lemma *hd-get-all-marked-decomposition-skip-some:*

assumes $(x, y) = \text{hd } (\text{get-all-marked-decomposition } M1)$
shows $(x, y) \in \text{set } (\text{get-all-marked-decomposition } (M0 @ \text{Marked } K \ i \ # \ M1))$
 ⟨proof⟩

lemma *in-get-all-marked-decomposition-in-get-all-marked-decomposition-prepend:*

$(a, b) \in \text{set } (\text{get-all-marked-decomposition } M') \implies$
 $\exists b'. (a, b' @ b) \in \text{set } (\text{get-all-marked-decomposition } (M @ M'))$
 ⟨proof⟩

lemma *get-all-marked-decomposition-remove-unmarked-length:*

assumes $\forall l \in \text{set } M'. \neg \text{is-marked } l$
shows $\text{length } (\text{get-all-marked-decomposition } (M' @ M''))$
 $= \text{length } (\text{get-all-marked-decomposition } M'')$
 ⟨proof⟩

lemma *get-all-marked-decomposition-not-is-marked-length:*

assumes $\forall l \in \text{set } M'. \neg \text{is-marked } l$
shows $1 + \text{length } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) \ P \ # \ M))$
 $= \text{length } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L \ l \ # \ M))$
 ⟨proof⟩

lemma *get-all-marked-decomposition-last-choice:*

assumes $\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L \ l \ # \ M)) \neq []$

and $\forall l \in \text{set } M'. \neg \text{is-marked } l$
and $\text{hd } (\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L \text{ } l \# M))) = (M0', M0)$
shows $\text{hd } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M)) = (M0', \text{Propagated } (-L) P \# M0)$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-except-last-choice-equal*:
assumes $\forall l \in \text{set } M'. \neg \text{is-marked } l$
shows $\text{tl } (\text{get-all-marked-decomposition } (\text{Propagated } (-L) P \# M))$
 $= \text{tl } (\text{tl } (\text{get-all-marked-decomposition } (M' @ \text{Marked } L \text{ } l \# M)))$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-hd-hd*:
assumes $\text{get-all-marked-decomposition } Ls = (M, C) \# (M0, M0') \# l$
shows $\text{tl } M = M0' @ M0 \wedge \text{is-marked } (\text{hd } M)$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-exists-prepend[dest]*:
assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
shows $\exists c. M = c @ b @ a$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-incl*:
assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
shows $\text{set } b \subseteq \text{set } M$ **and** $\text{set } a \subseteq \text{set } M$
 $\langle \text{proof} \rangle$

lemma *get-all-marked-decomposition-exists-prepend'*:
assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
obtains c **where** $M = c @ b @ a$
 $\langle \text{proof} \rangle$

lemma *union-in-get-all-marked-decomposition-is-subset*:
assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
shows $\text{set } a \cup \text{set } b \subseteq \text{set } M$
 $\langle \text{proof} \rangle$

lemma *Marked-cons-in-get-all-marked-decomposition-append-Marked-cons*:
 $\exists M1 M2. (\text{Marked } K \text{ } i \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (c @ \text{Marked } K \text{ } i \# c'))$
 $\langle \text{proof} \rangle$

14.3.2 Entailment of the Propagated by the Marked Literal

lemma *get-all-marked-decomposition-snd-union*:
 $\text{set } M = \bigcup (\text{set 'snd 'set } (\text{get-all-marked-decomposition } M)) \cup \{L \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$
(is $?M \text{ } M = ?U \text{ } M \cup ?Ls \text{ } M$ **)**
 $\langle \text{proof} \rangle$

definition *all-decomposition-implies* :: 'a literal multiset set
 $\Rightarrow ((\text{'a, 'l, 'm}) \text{marked-lit list} \times (\text{'a, 'l, 'm}) \text{marked-lit list}) \text{list} \Rightarrow \text{bool}$ **where**
 $\text{all-decomposition-implies } N \text{ } S \longleftrightarrow (\forall (Ls, \text{seen}) \in \text{set } S. \text{unmark-l } Ls \cup N \models_{ps} \text{unmark-l seen})$

lemma *all-decomposition-implies-empty[iff]*:
 $\text{all-decomposition-implies } N \text{ } [] \langle \text{proof} \rangle$

lemma *all-decomposition-implies-single[iff]*:

all-decomposition-implies $N [(Ls, \text{seen})] \longleftrightarrow \text{unmark-l } Ls \cup N \models_{ps} \text{unmark-l seen}$
 ⟨proof⟩

lemma *all-decomposition-implies-append*[iff]:
all-decomposition-implies $N (S @ S')$
 $\longleftrightarrow (\text{all-decomposition-implies } N S \wedge \text{all-decomposition-implies } N S')$
 ⟨proof⟩

lemma *all-decomposition-implies-cons-pair*[iff]:
all-decomposition-implies $N ((Ls, \text{seen}) \# S')$
 $\longleftrightarrow (\text{all-decomposition-implies } N [(Ls, \text{seen})] \wedge \text{all-decomposition-implies } N S')$
 ⟨proof⟩

lemma *all-decomposition-implies-cons-single*[iff]:
all-decomposition-implies $N (l \# S') \longleftrightarrow$
 $(\text{unmark-l } (\text{fst } l) \cup N \models_{ps} \text{unmark-l } (\text{snd } l) \wedge$
 $\text{all-decomposition-implies } N S')$
 ⟨proof⟩

lemma *all-decomposition-implies-trail-is-implied*:
assumes *all-decomposition-implies* $N (\text{get-all-marked-decomposition } M)$
shows $N \cup \{\text{unmark } L \mid L. \text{is-marked } L \wedge L \in \text{set } M\}$
 $\models_{ps} \text{unmark } ' \bigcup (\text{set } ' \text{snd } ' \text{set } (\text{get-all-marked-decomposition } M))$
 ⟨proof⟩

lemma *all-decomposition-implies-propagated-lits-are-implied*:
assumes *all-decomposition-implies* $N (\text{get-all-marked-decomposition } M)$
shows $N \cup \{\text{unmark } L \mid L. \text{is-marked } L \wedge L \in \text{set } M\} \models_{ps} \text{unmark-l } M$
 $(\text{is } ?I \models_{ps} ?A)$
 ⟨proof⟩

lemma *all-decomposition-implies-insert-single*:
all-decomposition-implies $N M \implies \text{all-decomposition-implies } (\text{insert } C N) M$
 ⟨proof⟩

14.4 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 $\{ \# L \# \} \in CNot \psi \longleftrightarrow -L \in \# \psi$
 ⟨proof⟩

lemma
shows
 $CNot\text{-singleton}[simp]: CNot \{ \# L \# \} = \{ \{ \# - L \# \} \}$ **and**
 $CNot\text{-empty}[simp]: CNot \{ \# \} = \{ \}$ **and**
 $CNot\text{-plus}[simp]: CNot (A + B) = CNot A \cup CNot B$
 ⟨proof⟩

lemma *CNot-eq-empty*[iff]:
 $CNot D = \{ \} \longleftrightarrow D = \{ \# \}$

$\langle \text{proof} \rangle$

lemma *in-CNot-implies-uminus*:

assumes $L \in \# D$ **and** $M \models_{as} CNot D$
shows $M \models_a \{\# - L \#\}$ **and** $-L \in \text{lits-of-l } M$
 $\langle \text{proof} \rangle$

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

$CNot (\text{remdups-mset } A) = CNot A$
 $\langle \text{proof} \rangle$

lemma *Ball-CNot-Ball-mset[simp]*:

$(\forall x \in CNot D. P x) \longleftrightarrow (\forall L \in \# D. P \{\# - L \#\})$
 $\langle \text{proof} \rangle$

lemma *consistent-CNot-not*:

assumes *consistent-interp* I
shows $I \models_s CNot \varphi \implies \neg I \models \varphi$
 $\langle \text{proof} \rangle$

lemma *total-not-true-clb-true-clss-CNot*:

assumes *total-over-m* $I \{\varphi\}$ **and** $\neg I \models \varphi$
shows $I \models_s CNot \varphi$
 $\langle \text{proof} \rangle$

lemma *total-not-CNot*:

assumes *total-over-m* $I \{\varphi\}$ **and** $\neg I \models_s CNot \varphi$
shows $I \models \varphi$
 $\langle \text{proof} \rangle$

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

$\text{atms-of-ms } (CNot C) = \text{atms-of } C$
 $\langle \text{proof} \rangle$

lemma *true-clss-clss-contradiction-true-clss-clb-false*:

$C \in D \implies D \models_{ps} CNot C \implies D \models_p \{\#\}$
 $\langle \text{proof} \rangle$

lemma *true-annots-CNot-all-atms-defined*:

assumes $M \models_{as} CNot T$ **and** $a1: L \in \# T$
shows $\text{atm-of } L \in \text{atm-of } \text{'lits-of-l } M$
 $\langle \text{proof} \rangle$

lemma *true-annots-CNot-all-uminus-atms-defined*:

assumes $M \models_{as} CNot T$ **and** $a1: -L \in \# T$
shows $\text{atm-of } L \in \text{atm-of } \text{'lits-of-l } M$
 $\langle \text{proof} \rangle$

lemma *true-clss-clss-false-left-right*:

assumes $\{\{\# L \#\}\} \cup B \models_p \{\#\}$
shows $B \models_{ps} CNot \{\# L \#\}$
 $\langle \text{proof} \rangle$

lemma *true-annots-true-clb-def-iff-negation-in-model*:

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

$\langle proof \rangle$

lemma *true-annot-CNot-diff*:

$I \models_{as} CNot\ C \implies I \models_{as} CNot\ (C - C')$

$\langle proof \rangle$

lemma *consistent-CNot-not-tautology*:

$consistent_interp\ M \implies M \models_s CNot\ D \implies \neg tautology\ D$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$total_over_m\ I\ (CNot\ C) = total_over_set\ I\ (atms_of\ C)$

$\langle proof \rangle$

The following lemma is very useful when in the goal appears an axioms like $- L = K$: this lemma allows the simplifier to rewrite L.

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

$\langle proof \rangle$

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

assumes

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

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

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

$\langle proof \rangle$

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 $M \models_{as} CNot\ A$

$\langle proof \rangle$

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:

$A \cup \{B\} \models_{ps} \{\{\#\}\} \longleftrightarrow A \models_{ps} CNot\ B$

$\langle proof \rangle$

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 $M' \models_a D$

$\langle proof \rangle$

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 $M' \models_a D$

$\langle proof \rangle$

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 $M' \models_{as} D$ $\langle proof \rangle$

lemma *all-variables-defined-not-imply-cnot*:

assumes

$\forall s \in \text{atms-of-ms } \{B\}. s \in \text{atm-of } \text{' lits-of-l } A \text{ and}$
 $\neg A \models_a B$
shows $A \models_{as} \text{CNot } B$
 $\langle \text{proof} \rangle$

lemma *CNot-union-mset[simp]*:
 $\text{CNot } (A \# \cup B) = \text{CNot } A \cup \text{CNot } B$
 $\langle \text{proof} \rangle$

14.5 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]*:
 $\text{no-dup } (\text{rev } M) \longleftrightarrow \text{no-dup } M$
 $\langle \text{proof} \rangle$

lemma *no-dup-length-eq-card-atm-of-lits-of-l*:
assumes $\text{no-dup } M$
shows $\text{length } M = \text{card } (\text{atm-of } \text{' lits-of-l } M)$
 $\langle \text{proof} \rangle$

lemma *distinct-consistent-interp*:
 $\text{no-dup } M \implies \text{consistent-interp } (\text{lits-of-l } M)$
 $\langle \text{proof} \rangle$

lemma *distinct-get-all-marked-decomposition-no-dup*:
assumes $(a, b) \in \text{set } (\text{get-all-marked-decomposition } M)$
and $\text{no-dup } M$
shows $\text{no-dup } (a @ b)$
 $\langle \text{proof} \rangle$

lemma *true-annots-lit-of-notin-skip*:
assumes $L \# M \models_{as} \text{CNot } A$
and $\neg \text{lit-of } L \notin \# A$
and $\text{no-dup } (L \# M)$
shows $M \models_{as} \text{CNot } A$
 $\langle \text{proof} \rangle$

14.6 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*:: $\text{'v clause multiset} \Rightarrow \text{'v clause multiset} \Rightarrow \text{bool}$ (**infix** \models_{psm} 50)
where
 $I \models_{psm} C \equiv \text{set-mset } I \models_{ps} (\text{set-mset } C)$

Analog of $\llbracket ?N \models_{ps} ?B; ?A \subseteq ?B \rrbracket \implies ?N \models_{ps} ?A$

lemma *true-clss-clssm-subsetE*: $N \models_{psm} B \implies A \subseteq \# B \implies N \models_{psm} A$
 $\langle \text{proof} \rangle$

abbreviation *true-clss-clm* :: 'a clause multiset \Rightarrow 'a clause \Rightarrow bool (**infix** \models_{pm} 50) **where**
 $I \models_{pm} C \equiv \text{set-mset } I \models_p C$

abbreviation *distinct-mset-mset* :: 'a multiset multiset \Rightarrow bool **where**
 $\text{distinct-mset-mset } \Sigma \equiv \text{distinct-mset-set } (\text{set-mset } \Sigma)$

abbreviation *all-decomposition-implies-m* **where**
 $\text{all-decomposition-implies-m } A B \equiv \text{all-decomposition-implies } (\text{set-mset } A) B$

abbreviation *atms-of-mm* :: 'a literal multiset multiset \Rightarrow 'a set **where**
 $\text{atms-of-mm } U \equiv \text{atms-of-ms } (\text{set-mset } U)$

Other definition using *Union-mset*

lemma $\text{atms-of-mm } U \equiv \text{set-mset } (\bigcup \# \text{ image-mset } (\text{image-mset atm-of}) U)$
 <proof>

abbreviation *true-clss-m* :: 'a interp \Rightarrow 'a clause multiset \Rightarrow bool (**infix** \models_{sm} 50) **where**
 $I \models_{sm} C \equiv I \models_s \text{set-mset } C$

abbreviation *true-clss-ext-m* (**infix** \models_{sextm} 49) **where**
 $I \models_{sextm} C \equiv I \models_{sext} \text{set-mset } C$

end

theory *CDCL-Abstract-Clause-Representation*

imports *Main Partial-Clausal-Logic*

begin

type-synonym 'v clause = 'v literal multiset

type-synonym 'v clauses = 'v clause multiset

14.7 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

$\text{mset-cl} :: 'cls \Rightarrow 'v \text{ clause}$ **and**

$\text{insert-cl} :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$ **and**

$\text{remove-lit} :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$

assumes

$\text{insert-cl}[\text{simp}]: \text{mset-cl } (\text{insert-cl } L C) = \text{mset-cl } C + \{\#L\# \}$ **and**

$\text{remove-lit}[\text{simp}]: \text{mset-cl } (\text{remove-lit } L C) = \text{remove1-mset } L (\text{mset-cl } C)$

begin

end

locale *raw-ccls-union* =

fixes

$\text{mset-cl} :: 'cls \Rightarrow 'v \text{ clause}$ **and**

$\text{union-cl} :: 'cls \Rightarrow 'cls \Rightarrow 'cls$ **and**

```

insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls
assumes
insert-ccls[simp]: mset-cls (insert-cls L C) = mset-cls C + {#L#} and
mset-ccls-union-cls[simp]: mset-cls (union-cls C D) = mset-cls C # $\cup$  mset-cls D and
remove-clit[simp]: mset-cls (remove-lit L C) = remove1-mset L (mset-cls C)
begin
end

```

Instantiation of the previous locale, in an unnamed context to avoid polluting with simp rules

```

context
begin
interpretation list-cls: raw-cls mset
  op # remove1
   $\langle$ proof $\rangle$ 

interpretation cls-cls: raw-cls id
   $\lambda L$  C. C + {#L#} remove1-mset
   $\langle$ proof $\rangle$ 

interpretation list-cls: raw-ccls-union mset
  union-mset-list
  op # remove1
   $\langle$ proof $\rangle$ 

interpretation cls-cls: raw-ccls-union id
  op # $\cup$   $\lambda L$  C. C + {#L#} remove1-mset
   $\langle$ proof $\rangle$ 
end

```

Over the abstract clauses, we have the following properties:

- We can insert a clause
- We can take the union (used only in proofs for the definition of *clauses*)
- there is an operator indicating whether the abstract clause is contained or not
- if a concrete clause is contained the abstract clauses, then there is an abstract clause

```

locale raw-clss =
  raw-cls mset-cls insert-cls remove-lit
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls +
fixes
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss
assumes
insert-clss[simp]: mset-clss (insert-clss L C) = mset-clss C + {#mset-cls L#} and
union-clss[simp]: mset-clss (union-clss C D) = mset-clss C + mset-clss D and

```

```

mset-clss-union-clss[simp]: mset-clss (insert-clss C' D) = {#mset-cls C'#} + mset-clss D and
in-clss-mset-clss[dest]: in-clss a C  $\implies$  mset-cls a  $\in$  # mset-clss C and
in-mset-clss-exists-preimage: b  $\in$  # mset-clss C  $\implies \exists b'. \text{in-clss } b' C \wedge \text{mset-cls } b' = b$  and
remove-from-clss-mset-clss[simp]:
  mset-clss (remove-from-clss a C) = mset-clss C - {#mset-cls a#} and
in-clss-union-clss[simp]:
  in-clss a (union-clss C D)  $\longleftrightarrow$  in-clss a C  $\vee$  in-clss a D
begin

end

experiment
begin
  fun remove-first where
    remove-first - [] = [] |
    remove-first C (C' # L) = (if mset C = mset C' then L else C' # remove-first C L)

  lemma mset-map-mset-remove-first:
    mset (map mset (remove-first a C)) = remove1-mset (mset a) (mset (map mset C))
    <proof>

  interpretation clss-clss: raw-clss id  $\lambda L C. C + \{\#L\# \}$  remove1-mset
    id op + op  $\in$  #  $\lambda L C. C + \{\#L\# \}$  remove1-mset
    <proof>

  interpretation list-clss: raw-clss mset
    op # remove1  $\lambda L. \text{mset (map mset L) op @ } \lambda L C. L \in \text{set } C \text{ op } \#$ 
    remove-first
    <proof>
end

end
theory CDCL-WNOT-Measure
imports Main
begin

```

15 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]:
 $\mu_C s b [] = 0$
 <proof>

lemma $\mu_C\text{-single}$ [simp]:
 $\mu_C s b [L] = L * b^{\wedge} (s - \text{Suc } 0)$
 <proof>

lemma *set-sum-atLeastLessThan-add*:
 $(\sum i=k..<k+(b::nat). f\ i) = (\sum i=0..<b. f\ (k+ i))$
 $\langle proof \rangle$

lemma *set-sum-atLeastLessThan-Suc*:
 $(\sum i=1..<Suc\ j. f\ i) = (\sum i=0..<j. f\ (Suc\ i))$
 $\langle proof \rangle$

lemma μ_C -cons:
 $\mu_C\ s\ b\ (L\ \# \ M) = L * b^{\wedge} (s - 1 - length\ M) + \mu_C\ s\ b\ M$
 $\langle proof \rangle$

lemma μ_C -append:
assumes $s \geq length\ (M@M')$
shows $\mu_C\ s\ b\ (M@M') = \mu_C\ (s - length\ M')\ b\ M + \mu_C\ s\ b\ M'$
 $\langle proof \rangle$

lemma μ_C -cons-non-empty-inf:
assumes $M\text{-ge-1}: \forall i \in set\ M. i \geq 1$ **and** $M: M \neq []$
shows $\mu_C\ s\ b\ M \geq b^{\wedge} (s - length\ M)$
 $\langle proof \rangle$

Copy of `~~/src/HOL/ex/NatSum.thy` (but generalized to $0 \leq k$)

lemma *sum-of-powers*: $0 \leq k \implies (k - 1) * (\sum i=0..<n. k^i) = k^n - (1::nat)$
 $\langle proof \rangle$

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**
 $M\text{-le}: \forall i < length\ M. M!i < b$ **and**
 $s \geq length\ M$
shows $\mu_C\ s\ b\ M < b^{\wedge} s$
 $\langle proof \rangle$

In the degenerate case $b = (0::'a)$, the list M is empty (since the list cannot contain any element).

lemma μ_C -bounded:
fixes $b::nat$
assumes
 $M\text{-le}: \forall i < length\ M. M!i < b$ **and**
 $s \geq length\ M$
 $b > 0$
shows $\mu_C\ s\ b\ M < b^{\wedge} s$
 $\langle proof \rangle$

When $b = 0$, we cannot show that the measure is empty, since $0^0 = 1$.

lemma μ_C -base-0:
assumes $length\ M \leq s$
shows $\mu_C\ s\ 0\ M \leq M!0$
 $\langle proof \rangle$

```

end
theory CDCL-NOT
imports CDCL-Abstract-Clause-Representation List-More Wellfounded-More CDCL-WNOT-Measure
Partial-Annotated-Clausal-Logic
begin

```

16 NOT's CDCL

16.1 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*:
 $no_dup\ M \implies -\ lit_of\ xa = lit_of\ x \implies x \in set\ M \implies xa \notin set\ M$
 $\langle proof \rangle$

lemma *atms-of-ms-single-atm-of[simp]*:
 $atms_of_ms\ \{unmark\ L\ |\ L.\ P\ L\} = atm_of\ '\ \{lit_of\ L\ |\ L.\ P\ L\}$
 $\langle proof \rangle$

lemma *atms-of-uminus-lit-atm-of-lit-of*:
 $atms_of\ \{\#\ -lit_of\ x.\ x \in \#\ A\#\} = atm_of\ '\ (lit_of\ '\ (set_mset\ A))$
 $\langle proof \rangle$

lemma *atms-of-ms-single-image-atm-of-lit-of*:
 $atms_of_ms\ (unmark_s\ A) = atm_of\ '\ (lit_of\ '\ A)$
 $\langle proof \rangle$

16.2 Initial definitions

16.2.1 The state

We define here an abstraction over operation on the state we are manipulating.

```

locale dpll-state-ops =
  raw-clss mset-clss insert-clss remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
for
  mset-clss :: 'cls  $\Rightarrow$  'v clause and
  insert-clss :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss +
fixes
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clssNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clssNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st
begin

```

notation *insert-cls* (**infix** $!++$ 50)

notation *in-clss* (**infix** $! \in$ 50)

notation *union-clss* (**infix** \oplus 50)

notation *insert-clss* (**infix** $!++!$ 50)

abbreviation *clauses_{NOT}* **where**

clauses_{NOT} $S \equiv mset-clss$ (*raw-clauses* 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 *dpll-state* =

dpll-state-ops *mset-cls* *insert-cls* *remove-lit* — related to each clause

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss* — related to the clauses

trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls_{NOT}* *remove-cls_{NOT}* — related to the state

for

mset-cls :: $'cls \Rightarrow 'v$ clause **and**

insert-cls :: $'v$ literal $\Rightarrow 'cls \Rightarrow 'cls$ **and**

remove-lit :: $'v$ literal $\Rightarrow 'cls \Rightarrow 'cls$ **and**

mset-clss:: $'clss \Rightarrow 'v$ clauses **and**

union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**

in-clss :: $'cls \Rightarrow 'clss \Rightarrow bool$ **and**

insert-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

remove-from-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

trail :: $'st \Rightarrow ('v, unit, unit)$ marked-lits **and**

raw-clauses :: $'st \Rightarrow 'clss$ **and**

prepend-trail :: $('v, unit, unit)$ marked-lit $\Rightarrow 'st \Rightarrow 'st$ **and**

tl-trail :: $'st \Rightarrow 'st$ **and**

add-cls_{NOT} :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**

remove-cls_{NOT} :: $'cls \Rightarrow 'st \Rightarrow 'st +$

assumes

trail-prepend-trail[simp]:

$\bigwedge st L. \text{undefined-lit } (trail\ st) \text{ (lit-of } L) \implies trail\ (prepend-trail\ L\ st) = L \# trail\ st$

and

tl-trail[simp]: $trail\ (tl-trail\ S) = tl\ (trail\ S)$ **and**

trail-add-cls_{NOT}[simp]: $\bigwedge st C. \text{no-dup } (trail\ st) \implies trail\ (add-cls_{NOT}\ C\ st) = trail\ st$ **and**

trail-remove-cls_{NOT}[simp]: $\bigwedge st C. trail\ (remove-cls_{NOT}\ C\ st) = trail\ st$ **and**

clauses-prepend-trail[simp]:

$\bigwedge st L. \text{undefined-lit } (trail\ st) \text{ (lit-of } L) \implies$

$clauses_{NOT}\ (prepend-trail\ L\ st) = clauses_{NOT}\ st$

and

clauses-tl-trail[simp]: $\bigwedge st. clauses_{NOT}\ (tl-trail\ st) = clauses_{NOT}\ st$ **and**

clauses-add-cls_{NOT}[simp]:

$\bigwedge st C. \text{no-dup } (trail\ st) \implies clauses_{NOT}\ (add-cls_{NOT}\ C\ st) = \{\#mset-cls\ C\} + clauses_{NOT}\ st$

and

clauses-remove-cls_{NOT}[simp]:

$\bigwedge st C. clauses_{NOT}\ (remove-cls_{NOT}\ C\ st) = removeAll-mset\ (mset-cls\ C)\ (clauses_{NOT}\ st)$

begin

We define the following function doing the backtrack in the trail:

function *reduce-trail-to_{NOT}* :: $'a\ list \Rightarrow 'st \Rightarrow 'st$ **where**

$\text{reduce-trail-to}_{NOT} F S =$
 (if $\text{length } (\text{trail } S) = \text{length } F \vee \text{trail } S = []$ then S else $\text{reduce-trail-to}_{NOT} F (\text{tl-trail } S)$)
 $\langle \text{proof} \rangle$
termination $\langle \text{proof} \rangle$
declare $\text{reduce-trail-to}_{NOT}.\text{simps}[\text{simp del}]$

Then we need several lemmas about the $\text{reduce-trail-to}_{NOT}$.

lemma

shows

$\text{reduce-trail-to}_{NOT}.\text{nil}[\text{simp}]: \text{trail } S = [] \implies \text{reduce-trail-to}_{NOT} F S = S$ **and**
 $\text{reduce-trail-to}_{NOT}.\text{eq-length}[\text{simp}]: \text{length } (\text{trail } S) = \text{length } F \implies \text{reduce-trail-to}_{NOT} F S = S$
 $\langle \text{proof} \rangle$

lemma $\text{reduce-trail-to}_{NOT}.\text{length-ne}[\text{simp}]:$

$\text{length } (\text{trail } S) \neq \text{length } F \implies \text{trail } S \neq [] \implies$
 $\text{reduce-trail-to}_{NOT} F S = \text{reduce-trail-to}_{NOT} F (\text{tl-trail } S)$
 $\langle \text{proof} \rangle$

lemma $\text{trail-reduce-trail-to}_{NOT}.\text{length-le}:$

assumes $\text{length } F > \text{length } (\text{trail } S)$
shows $\text{trail } (\text{reduce-trail-to}_{NOT} F S) = []$
 $\langle \text{proof} \rangle$

lemma $\text{trail-reduce-trail-to}_{NOT}.\text{nil}[\text{simp}]:$

$\text{trail } (\text{reduce-trail-to}_{NOT} [] S) = []$
 $\langle \text{proof} \rangle$

lemma $\text{clauses-reduce-trail-to}_{NOT}.\text{nil}:$

$\text{clauses}_{NOT} (\text{reduce-trail-to}_{NOT} [] S) = \text{clauses}_{NOT} S$
 $\langle \text{proof} \rangle$

lemma $\text{trail-reduce-trail-to}_{NOT}.\text{drop}:$

$\text{trail } (\text{reduce-trail-to}_{NOT} F S) =$
 (if $\text{length } (\text{trail } S) \geq \text{length } F$
 then $\text{drop } (\text{length } (\text{trail } S) - \text{length } F) (\text{trail } S)$
 else $[])$
 $\langle \text{proof} \rangle$

lemma $\text{reduce-trail-to}_{NOT}.\text{skip-beginning}:$

assumes $\text{trail } S = F' @ F$
shows $\text{trail } (\text{reduce-trail-to}_{NOT} F S) = F$
 $\langle \text{proof} \rangle$

lemma $\text{reduce-trail-to}_{NOT}.\text{clauses}[\text{simp}]:$

$\text{clauses}_{NOT} (\text{reduce-trail-to}_{NOT} F S) = \text{clauses}_{NOT} S$
 $\langle \text{proof} \rangle$

lemma $\text{trail-eq-reduce-trail-to}_{NOT}.\text{eq}:$

$\text{trail } S = \text{trail } T \implies \text{trail } (\text{reduce-trail-to}_{NOT} F S) = \text{trail } (\text{reduce-trail-to}_{NOT} F T)$
 $\langle \text{proof} \rangle$

lemma $\text{trail-reduce-trail-to}_{NOT}.\text{add-cl}_{NOT}[\text{simp}]:$

$\text{no-dup } (\text{trail } S) \implies$
 $\text{trail } (\text{reduce-trail-to}_{NOT} F (\text{add-cl}_{NOT} C S)) = \text{trail } (\text{reduce-trail-to}_{NOT} F S)$
 $\langle \text{proof} \rangle$

lemma *reduce-trail-to_{NOT}-trail-tl-trail-decomp[simp]*:
 $trail\ S = F' @\ Marked\ K\ () \# F \implies$
 $trail\ (reduce-trail-to_{NOT}\ F\ (tl-trail\ S)) = F$
 $\langle proof \rangle$

lemma *reduce-trail-to_{NOT}-length*:
 $length\ M = length\ M' \implies reduce-trail-to_{NOT}\ M\ S = reduce-trail-to_{NOT}\ M'\ S$
 $\langle proof \rangle$

abbreviation *trail-weight where*
 $trail-weight\ S \equiv map\ ((\lambda l.\ 1 + length\ l)\ o\ snd)\ (get-all-marked-decomposition\ (trail\ S))$

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**
 $S \sim T \longleftrightarrow trail\ S = trail\ T \wedge clauses_{NOT}\ S = clauses_{NOT}\ T$

lemma *state-eq_{NOT}-ref[simp]*:
 $S \sim S$
 $\langle proof \rangle$

lemma *state-eq_{NOT}-sym*:
 $S \sim T \longleftrightarrow T \sim S$
 $\langle proof \rangle$

lemma *state-eq_{NOT}-trans*:
 $S \sim T \implies T \sim U \implies S \sim U$
 $\langle proof \rangle$

lemma
shows
 $state-eq_{NOT}\text{-}trail: S \sim T \implies trail\ S = trail\ T$ **and**
 $state-eq_{NOT}\text{-}clauses: S \sim T \implies clauses_{NOT}\ S = clauses_{NOT}\ T$
 $\langle proof \rangle$

lemmas $state-simp_{NOT}[simp] = state-eq_{NOT}\text{-}trail\ state-eq_{NOT}\text{-}clauses$

lemma *reduce-trail-to_{NOT}-state-eq_{NOT}-compatible*:
assumes *ST*: $S \sim T$
shows $reduce-trail-to_{NOT}\ F\ S \sim reduce-trail-to_{NOT}\ F\ T$
 $\langle proof \rangle$

end

16.2.2 Definition of the operation

Each possible is in its own locale.

locale *propagate-ops* =
 $dpll\text{-}state\ mset\text{-}cls\ insert\text{-}cls\ remove\text{-}lit$
 $mset\text{-}class\ union\text{-}class\ in\text{-}class\ insert\text{-}class\ remove\text{-}from\text{-}class$
 $trail\ raw\text{-}clauses\ prepend\text{-}trail\ tl\text{-}trail\ add\text{-}cls_{NOT}\ remove\text{-}cls_{NOT}$
for
 $mset\text{-}cls :: 'cls \Rightarrow 'v\ clause$ **and**


```

insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
mset-clss:: 'clss  $\Rightarrow$  'v clauses and
union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
raw-clauses :: 'st  $\Rightarrow$  'clss and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  propagate-cond :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive propagateNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  propagateNOT[intro]: C + {#L#}  $\in$  # clausesNOT S  $\Rightarrow$  trail S  $\models_{as}$  CNot C
     $\Rightarrow$  undefined-lit (trail S) L
     $\Rightarrow$  propagate-cond (Propagated L ()) S
     $\Rightarrow$  T  $\sim$  prepend-trail (Propagated L ()) S
     $\Rightarrow$  propagateNOT S T
inductive-cases propagateNOTE[elim]: propagateNOT S T
end

locale decide-ops =
  dpll-state mset-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st
begin
inductive decideNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
  decideNOT[intro]: undefined-lit (trail S) L  $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)
     $\Rightarrow$  T  $\sim$  prepend-trail (Marked L ()) S
     $\Rightarrow$  decideNOT S T
inductive-cases decideNOTE[elim]: decideNOT S S'
end

locale backjumping-ops =

```

```

dpll-state mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

inductive backjump where
trail S = F' @ Marked K ()# F
 $\Rightarrow$  T  $\sim$  prepend-trail (Propagated L ()) (reduce-trail-toNOT F S)
 $\Rightarrow$  C  $\in$  # clausesNOT S
 $\Rightarrow$  trail S  $\models_{as}$  CNot C
 $\Rightarrow$  undefined-lit F L
 $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))
 $\Rightarrow$  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 $atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ ' \ (lits\text{-}of\text{-}l\ (trail\ S))$ is not implied by the the condition $clauses_{NOT}\ S \models_{pm}\ C' + \{ \#L\# \}$ (no negation).

end

16.3 DPLL with backjumping

```

locale dpll-with-backjumping-ops =
  propagate-ops mset-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT propagate-conds +
  decide-ops mset-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT +
  backjumping-ops mset-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT backjump-conds
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and

```

```

mset-clss :: 'clss  $\Rightarrow$  'v clauses and
union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
raw-clauses :: 'st  $\Rightarrow$  'clss and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clssNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clssNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
inv :: 'st  $\Rightarrow$  bool and
backjump-conds :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool +
assumes
  bj-can-jump:
   $\bigwedge S C F' K F L.$ 
  inv S  $\Rightarrow$ 
  no-dup (trail S)  $\Rightarrow$ 
  trail S = F' @ Marked K () # F  $\Rightarrow$ 
  C  $\in$  # 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' @ Marked K () # F))  $\Rightarrow$ 
  clausesNOT S  $\models_{pm}$  C' + {#L#}  $\Rightarrow$ 
  F  $\models_{as}$  CNot C'  $\Rightarrow$ 
   $\neg$ no-step backjump S
begin

```

We cannot add a like condition $atms\text{-}of\ C' \subseteq atms\text{-}of\text{-}ms\ N$ to ensure that we can backjump even if the last decision variable has disappeared from the set of clauses.

The part of the condition $atm\text{-}of\ L \in atm\text{-}of\ ' lits\text{-}of\text{-}l\ (F' @ Marked\ K\ () \# F)$ is important, otherwise you are not sure that you can backtrack.

16.3.1 Definition

We define dpll with backjumping:

```

inductive dpll-bj :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool for S :: 'st where
  bj-decideNOT: decideNOT S S'  $\Rightarrow$  dpll-bj S S' |
  bj-propagateNOT: propagateNOT S S'  $\Rightarrow$  dpll-bj S S' |
  bj-backjump: backjump S S'  $\Rightarrow$  dpll-bj S S'

```

lemmas dpll-bj-induct = dpll-bj.induct[split-format(complete)]

thm dpll-bj-induct[OF dpll-with-backjumping-ops-axioms]

lemma dpll-bj-all-induct[consumes 2, case-names decide_{NOT} propagate_{NOT} backjump]:

fixes S T :: 'st

assumes

dpll-bj S T **and**

inv S

$\bigwedge L T. \text{undefined-lit (trail S) L} \Rightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S)$

$\Rightarrow T \sim \text{prepend-trail (Marked L ()) S}$

$\Rightarrow P\ S\ T$ **and**

$\bigwedge C L T. C + \{ \#L\# \} \in \# clauses_{NOT}\ S \Rightarrow trail\ S \models_{as}\ CNot\ C \Rightarrow \text{undefined-lit (trail S) L}$

$\Rightarrow T \sim \text{prepend-trail (Propagated L ()) S}$

$\Rightarrow P S T$ **and**
 $\wedge C F' K F L C' T. C \in \# \text{ clauses}_{NOT} S \Rightarrow F' @ \text{Marked } K () \# F \models_{as} CNot C$
 $\Rightarrow \text{trail } S = F' @ \text{Marked } K () \# F$
 $\Rightarrow \text{undefined-lit } F L$
 $\Rightarrow \text{atm-of } L \in \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of } ' (\text{lits-of-l } (F' @ \text{Marked } K () \# F))$
 $\Rightarrow \text{clauses}_{NOT} S \models_{pm} C' + \{\#L\# \}$
 $\Rightarrow F \models_{as} CNot C'$
 $\Rightarrow T \sim \text{prepend-trail } (\text{Propagated } L ()) (\text{reduce-trail-to}_{NOT} F S)$
 $\Rightarrow P S T$
shows $P S T$
 $\langle \text{proof} \rangle$

16.3.2 Basic properties

First, some better suited induction principle lemma *dpll-bj-clauses*:

assumes $dpll\text{-}bj\ S\ T$ **and** $inv\ S$
shows $\text{clauses}_{NOT} S = \text{clauses}_{NOT} T$
 $\langle \text{proof} \rangle$

No duplicates in the trail lemma *dpll-bj-no-dup*:

assumes $dpll\text{-}bj\ S\ T$ **and** $inv\ S$
and $no\text{-}dup\ (\text{trail } S)$
shows $no\text{-}dup\ (\text{trail } T)$
 $\langle \text{proof} \rangle$

Valuations lemma *dpll-bj-sat-iff*:

assumes $dpll\text{-}bj\ S\ T$ **and** $inv\ S$
shows $I \models_{sm} \text{clauses}_{NOT} S \longleftrightarrow I \models_{sm} \text{clauses}_{NOT} T$
 $\langle \text{proof} \rangle$

Clauses lemma *dpll-bj-atms-of-ms-clauses-inv*:

assumes
 $dpll\text{-}bj\ S\ T$ **and**
 $inv\ S$
shows $\text{atms-of-mm } (\text{clauses}_{NOT} S) = \text{atms-of-mm } (\text{clauses}_{NOT} T)$
 $\langle \text{proof} \rangle$

lemma *dpll-bj-atms-in-trail*:

assumes
 $dpll\text{-}bj\ S\ T$ **and**
 $inv\ S$ **and**
 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S)$
shows $\text{atm-of } ' (\text{lits-of-l } (\text{trail } T)) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S)$
 $\langle \text{proof} \rangle$

lemma *dpll-bj-atms-in-trail-in-set*:

assumes $dpll\text{-}bj\ S\ T$ **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 $\text{atm-of } ' (\text{lits-of-l } (\text{trail } T)) \subseteq A$
 $\langle \text{proof} \rangle$

lemma *dpll-bj-all-decomposition-implies-inv*:

assumes

$dpll\text{-}bj\ S\ T$ **and**
 $inv: inv\ S$ **and**
 $decomp: all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ S)\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S))$
shows $all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ T)\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ T))$
 $\langle proof \rangle$

16.3.3 Termination

Using a proper measure lemma $length\text{-}get\text{-}all\text{-}marked\text{-}decomposition\text{-}append\text{-}Marked$:
 $length\ (get\text{-}all\text{-}marked\text{-}decomposition\ (F' @ Marked\ K\ () \# F)) =$
 $length\ (get\text{-}all\text{-}marked\text{-}decomposition\ F')$
 $+ length\ (get\text{-}all\text{-}marked\text{-}decomposition\ (Marked\ K\ () \# F))$
 $- 1$
 $\langle proof \rangle$

lemma $take\text{-}length\text{-}get\text{-}all\text{-}marked\text{-}decomposition\text{-}marked\text{-}sandwich$:
 $take\ (length\ (get\text{-}all\text{-}marked\text{-}decomposition\ F))$
 $(map\ (f\ o\ snd)\ (rev\ (get\text{-}all\text{-}marked\text{-}decomposition\ (F' @ Marked\ K\ () \# F))))$
 $=$
 $map\ (f\ o\ snd)\ (rev\ (get\text{-}all\text{-}marked\text{-}decomposition\ F))$

$\langle proof \rangle$

lemma $length\text{-}get\text{-}all\text{-}marked\text{-}decomposition\text{-}length$:
 $length\ (get\text{-}all\text{-}marked\text{-}decomposition\ M) \leq 1 + length\ M$
 $\langle proof \rangle$

lemma $length\text{-}in\text{-}get\text{-}all\text{-}marked\text{-}decomposition\text{-}bounded$:
assumes $i: i \in set\ (trail\text{-}weight\ S)$
shows $i \leq Suc\ (length\ (trail\ S))$
 $\langle proof \rangle$

Well-foundedness The bounds are the following:

- $1 + card\ (atms\text{-}of\text{-}ms\ A)$: $card\ (atms\text{-}of\text{-}ms\ A)$ is an upper bound on the length of the list. As $get\text{-}all\text{-}marked\text{-}decomposition$ appends an possibly empty couple at the end, adding one is needed.
- $2 + card\ (atms\text{-}of\text{-}ms\ A)$: $card\ (atms\text{-}of\text{-}ms\ A)$ 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 $unassigned\text{-}lit :: 'b\ literal\ multiset\ set \Rightarrow 'a\ list \Rightarrow nat$ **where**
 $unassigned\text{-}lit\ N\ M \equiv card\ (atms\text{-}of\text{-}ms\ N) - length\ M$

lemma $dpll\text{-}bj\text{-}trail\text{-}mes\text{-}increasing\text{-}prop$:
fixes $M :: ('v, unit, unit)\ marked\text{-}lits$ **and** $N :: 'v\ clauses$
assumes
 $dpll\text{-}bj\ S\ T$ **and**
 $inv\ S$ **and**
 $NA: atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $MA: atm\text{-}of\ ' lits\text{-}of\text{-}l\ (trail\ S) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ **and**
 $finite: finite\ A$
shows $\mu_C\ (1 + card\ (atms\text{-}of\text{-}ms\ A))\ (2 + card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T)$

$> \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S)$
 $\langle \text{proof} \rangle$

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*) \subseteq *atms-of-ms A* **and**

M-A: *atm-of* ' *lits-of-l* (*trail S*) \subseteq *atms-of-ms A* **and**

nd: *no-dup* (*trail S*) **and**

fin-A: *finite A*

shows $(2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $\quad < (2 + \text{card} (\text{atms-of-ms } A)) \wedge (1 + \text{card} (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card} (\text{atms-of-ms } A)) (2 + \text{card} (\text{atms-of-ms } A)) (\text{trail-weight } S)$

$\langle \text{proof} \rangle$

lemma *wf-dpll-bj*:

assumes *fin*: *finite A*

shows *wf* $\{(T, S). \text{dpll-bj } S T$

$\wedge \text{atms-of-mm} (\text{clauses}_{\text{NOT}} S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l} (\text{trail } S) \subseteq \text{atms-of-ms } A$

$\wedge \text{no-dup} (\text{trail } S) \wedge \text{inv } S\}$

(**is** *wf* ?*A*)

$\langle \text{proof} \rangle$

16.3.4 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*, $\neg M \models_{\text{as}} N$ 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 $\neg M \models_{\text{as}} N$ 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 literal multiset set **and** *S T* :: 'st

assumes

atms-of-mm (*clauses*_{NOT} *S*) \subseteq *atms-of-ms A* **and**

atm-of ' *lits-of-l* (*trail S*) \subseteq *atms-of-ms A* **and**

no-dup (*trail S*) **and**

finite A **and**

inv: *inv S* **and**

n-s: *no-step dpll-bj S* **and**

decomp: *all-decomposition-implies-m* (*clauses*_{NOT} *S*) (*get-all-marked-decomposition* (*trail S*))

shows *unsatisfiable* (*set-mset* (*clauses*_{NOT} *S*))

$\vee (\text{trail } S \models_{\text{asm}} \text{clauses}_{\text{NOT}} S \wedge \text{satisfiable} (\text{set-mset} (\text{clauses}_{\text{NOT}} S)))$

<proof>

end — End of *dpll-with-backjumping-ops*

locale *dpll-with-backjumping* =

dpll-with-backjumping-ops *mset-cls* *insert-cls* *remove-lit*
mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*
trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls_{NOT}* *remove-cls_{NOT}* *inv* *backjump-conds*
propagate-conds

for

mset-cls :: '*cls* ⇒ '*v* clause **and**
insert-cls :: '*v* literal ⇒ '*cls* ⇒ '*cls* **and**
remove-lit :: '*v* literal ⇒ '*cls* ⇒ '*cls* **and**
mset-clss :: '*clss* ⇒ '*v* clauses **and**
union-clss :: '*clss* ⇒ '*clss* ⇒ '*clss* **and**
in-clss :: '*cls* ⇒ '*clss* ⇒ bool **and**
insert-clss :: '*cls* ⇒ '*clss* ⇒ '*clss* **and**
remove-from-clss :: '*cls* ⇒ '*clss* ⇒ '*clss* **and**
trail :: '*st* ⇒ ('*v*, unit, unit) marked-lits **and**
raw-clauses :: '*st* ⇒ '*clss* **and**
prepend-trail :: ('*v*, unit, unit) marked-lit ⇒ '*st* ⇒ '*st* **and**
tl-trail :: '*st* ⇒ '*st* **and**
add-cls_{NOT} :: '*cls* ⇒ '*st* ⇒ '*st* **and**
remove-cls_{NOT} :: '*cls* ⇒ '*st* ⇒ '*st* **and**
inv :: '*st* ⇒ bool **and**
backjump-conds :: '*v* clause ⇒ '*v* clause ⇒ '*v* literal ⇒ '*st* ⇒ '*st* ⇒ bool **and**
propagate-conds :: ('*v*, unit, unit) marked-lit ⇒ '*st* ⇒ bool

+

assumes *dpll-bj-inv*: $\bigwedge S T. \text{dpll-bj } S T \implies \text{inv } S \implies \text{inv } T$

begin

lemma *rtrancpl-dpll-bj-inv*:

assumes *dpll-bj^{**}* *S* *T* **and** *inv* *S*
shows *inv* *T*
<proof>

lemma *rtrancpl-dpll-bj-no-dup*:

assumes *dpll-bj^{**}* *S* *T* **and** *inv* *S*
and *no-dup* (*trail* *S*)
shows *no-dup* (*trail* *T*)
<proof>

lemma *rtrancpl-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*)
<proof>

lemma *rtrancpl-dpll-bj-atms-in-trail*:

assumes
*dpll-bj^{**}* *S* *T* **and**
inv *S* **and**
atm-of ' (*lits-of-l* (*trail* *S*)) \subseteq *atms-of-mm* (*clauses_{NOT}* *S*)
shows *atm-of* ' (*lits-of-l* (*trail* *T*)) \subseteq *atms-of-mm* (*clauses_{NOT}* *T*)
<proof>

lemma *rtrancpl-dpll-bj-sat-iff*:

assumes *dpll-bj** S T* **and** *inv S*

shows $I \models_{sm} clauses_{NOT} S \longleftrightarrow I \models_{sm} clauses_{NOT} T$

<proof>

lemma *rtrancpl-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*

<proof>

lemma *rtrancpl-dpll-bj-all-decomposition-implies-inv*:

assumes

*dpll-bj** S T* **and**

inv S

all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))

shows *all-decomposition-implies-m (clauses_{NOT} T) (get-all-marked-decomposition (trail T))*

<proof>

lemma *rtrancpl-dpll-bj-inv-incl-dpll-bj-inv-tranc*:

$\{(T, S). dpll-bj^{++} S T$

$\wedge atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A \wedge atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A$

$\wedge no-dup (trail S) \wedge inv S\}$

$\subseteq \{(T, S). dpll-bj S T \wedge atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A$

$\wedge atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A \wedge no-dup (trail S) \wedge inv S\}^+$

(is ?A ⊆ ?B⁺)

<proof>

lemma *wf-trancpl-dpll-bj*:

assumes *fin: finite A*

shows *wf* $\{(T, S). dpll-bj^{++} S T$

$\wedge atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A \wedge atm-of ' lits-of-l (trail S) \subseteq atms-of-ms A$

$\wedge no-dup (trail S) \wedge inv S\}$

<proof>

lemma *dpll-bj-sat-ext-iff*:

dpll-bj S T ⇒ inv S ⇒ I ⊨_{sextm} clauses_{NOT} S ⇔ I ⊨_{sextm} clauses_{NOT} T

<proof>

lemma *rtrancpl-dpll-bj-sat-ext-iff*:

*dpll-bj** S T ⇒ inv S ⇒ I ⊨_{sextm} clauses_{NOT} S ⇔ I ⊨_{sextm} clauses_{NOT} T*

<proof>

theorem *full-dpll-backjump-final-state*:

fixes *A :: 'v literal multiset set* **and** *S T :: 'st*

assumes

full: full dpll-bj S T **and**

atms-S: atms-of-mm (clauses_{NOT} S) ⊆ 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**

finite A **and**

inv: *inv S* **and**
decomp: *all-decomposition-implies-m* (*clauses_{NOT} S*) (*get-all-marked-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses_{NOT} S*))
 \vee (*trail T* \models_{asm} *clauses_{NOT} S* \wedge *satisfiable* (*set-mset* (*clauses_{NOT} S*)))
 <proof>

corollary *full-dpll-backjump-final-state-from-init-state*:

fixes *A* :: '*v* literal multiset set **and** *S T* :: '*st*
assumes
full: *full dpll-bj S T* **and**
trail S = [] **and**
clauses_{NOT} S = *N* **and**
inv S
shows *unsatisfiable* (*set-mset N*) \vee (*trail T* \models_{asm} *N* \wedge *satisfiable* (*set-mset N*))
 <proof>

lemma *trancpl-dpll-bj-trail-mes-decreasing-prop*:

assumes *dpll*: *dpll-bj⁺⁺ S T* **and** *inv*: *inv S* **and**
N-A: *atms-of-mm* (*clauses_{NOT} S*) \subseteq *atms-of-ms A* **and**
M-A: *atm-of* '*lits-of-l* (*trail S*) \subseteq *atms-of-ms A* **and**
n-d: *no-dup* (*trail S*) **and**
fin-A: *finite A*
shows $(2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T)$
 $\quad < (2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } S)$
 <proof>

end — End of *dpll-with-backjumping*

16.4 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.

16.4.1 Learn and Forget

Learning adds a new clause where all the literals are already included in the clauses.

locale *learn-ops* =

dpll-state mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
for
mset-cls :: '*cls* \Rightarrow '*v* clause **and**
insert-cls :: '*v* literal \Rightarrow '*cls* \Rightarrow '*cls* **and**
remove-lit :: '*v* literal \Rightarrow '*cls* \Rightarrow '*cls* **and**
mset-clss:: '*clss* \Rightarrow '*v* clauses **and**
union-clss :: '*clss* \Rightarrow '*clss* \Rightarrow '*clss* **and**
in-clss :: '*cls* \Rightarrow '*clss* \Rightarrow bool **and**
insert-clss :: '*cls* \Rightarrow '*clss* \Rightarrow '*clss* **and**
remove-from-clss :: '*cls* \Rightarrow '*clss* \Rightarrow '*clss* **and**
trail :: '*st* \Rightarrow ('*v*, unit, unit) marked-lits **and**
raw-clauses :: '*st* \Rightarrow '*clss* **and**
prepend-trail :: ('*v*, unit, unit) marked-lit \Rightarrow '*st* \Rightarrow '*st* **and**

```

  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  learn-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive learn :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
learnNOT-rule: clausesNOT S  $\models_{pm}$  mset-cls C  $\Rightarrow$ 
  atms-of (mset-cls C)  $\subseteq$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))  $\Rightarrow$ 
  learn-cond C S  $\Rightarrow$ 
  T  $\sim$  add-clsNOT 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-weight S) =  $\mu_C$  A B (trail-weight T)
  <proof>
end

```

Forget removes an information that can be deduced from the context (e.g. redundant clauses, tautologies)

```

locale forget-ops =
  dpll-state mset-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st +
fixes
  forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive forgetNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool where
forgetNOT:
  removeAll-mset (mset-cls C)(clausesNOT S)  $\models_{pm}$  mset-cls C  $\Rightarrow$ 
  forget-cond C S  $\Rightarrow$ 
  C ! $\in$ ! raw-clauses 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  $T$ )
<proof>
end

locale learn-and-forgetNOT =
  learn-ops mset-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT learn-cond +
  forget-ops mset-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT forget-cond
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clsNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  learn-cond forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool
begin
inductive learn-and-forgetNOT :: 'st  $\Rightarrow$  'st  $\Rightarrow$  bool
where
  lf-learn: learn  $S T \Longrightarrow$  learn-and-forgetNOT  $S T$  |
  lf-forget: forgetNOT  $S T \Longrightarrow$  learn-and-forgetNOT  $S T$ 
end

```

16.4.2 Definition of CDCL

```

locale conflict-driven-clause-learning-ops =
  dpll-with-backjumping-ops mset-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT
    inv backjump-conds propagate-conds +
  learn-and-forgetNOT mset-cls insert-cls remove-lit
    mset-clss union-clss in-clss insert-clss remove-from-clss
    trail raw-clauses prepend-trail tl-trail add-clsNOT remove-clsNOT learn-cond
    forget-cond
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and

```

```

prepend-trail :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ 'st and
tl-trail :: 'st ⇒ 'st and
add-clsNOT :: 'cls ⇒ 'st ⇒ 'st and
remove-clsNOT :: 'cls ⇒ 'st ⇒ 'st and
inv :: 'st ⇒ bool and
backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
learn-cond forget-cond :: 'cls ⇒ 'st ⇒ bool
begin

inductive cdclNOT :: 'st ⇒ 'st ⇒ bool for S :: 'st where
c-dpll-bj: dpll-bj S S' ⇒ cdclNOT S S' |
c-learn: learn S S' ⇒ cdclNOT S S' |
c-forgetNOT: forgetNOT S S' ⇒ cdclNOT S S'

lemma cdclNOT-all-induct[consumes 1, case-names dpll-bj learn forgetNOT]:
fixes S T :: 'st
assumes cdclNOT S T and
dpll: ∧T. dpll-bj S T ⇒ P S T and
learning:
  ∧C T. clausesNOT S ⊨pm mset-cls C ⇒
  atms-of (mset-cls C) ⊆ atms-of-mm (clausesNOT S) ∪ atm-of ' (lits-of-l (trail S)) ⇒
  T ~ add-clsNOT C S ⇒
  P S T and
forgetting: ∧C T. removeAll-mset (mset-cls C) (clausesNOT S) ⊨pm mset-cls C ⇒
  C !∈! raw-clauses S ⇒
  T ~ remove-clsNOT C S ⇒
  P S T
shows P S T
⟨proof⟩

lemma cdclNOT-no-dup:
assumes
  cdclNOT S T and
  inv S and
  no-dup (trail S)
shows no-dup (trail T)
⟨proof⟩

```

Consistency of the trail **lemma** cdcl_{NOT}-consistent:

```

assumes
  cdclNOT S T and
  inv S and
  no-dup (trail S)
shows consistent-interp (lits-of-l (trail T))
⟨proof⟩

```

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 cdclNOT-atms-of-ms-clauses-decreasing:
assumes cdclNOT S T and inv S and no-dup (trail S)
shows atms-of-mm (clausesNOT T) ⊆ atms-of-mm (clausesNOT S) ∪ atm-of ' (lits-of-l (trail S))
⟨proof⟩

```

lemma *cdcl_{NOT}-atms-in-trail*:

assumes *cdcl_{NOT} S T and inv S and no-dup (trail S)*
and *atm-of ‘ (lits-of-l (trail S)) \subseteq atms-of-mm (clauses_{NOT} S)*
shows *atm-of ‘ (lits-of-l (trail T)) \subseteq atms-of-mm (clauses_{NOT} S)*
 \langle *proof* \rangle

lemma *cdcl_{NOT}-atms-in-trail-in-set*:

assumes
cdcl_{NOT} S T and inv S and no-dup (trail S) and
atms-of-mm (clauses_{NOT} S) \subseteq A and
atm-of ‘ (lits-of-l (trail S)) \subseteq A
shows *atm-of ‘ (lits-of-l (trail T)) \subseteq A*
 \langle *proof* \rangle

lemma *cdcl_{NOT}-all-decomposition-implies*:

assumes *cdcl_{NOT} S T and inv S and n-d[simp]: no-dup (trail S) and*
all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))
shows
all-decomposition-implies-m (clauses_{NOT} T) (get-all-marked-decomposition (trail T))
 \langle *proof* \rangle

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 *$I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} S \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} T$*
 \langle *proof* \rangle

end — end of *conflict-driven-clause-learning-ops*

16.4.3 CDCL with invariant

locale *conflict-driven-clause-learning* =

conflict-driven-clause-learning-ops +
assumes *cdcl_{NOT}-inv: $\bigwedge S T. \text{cdcl}_{\text{NOT}} S T \implies \text{inv } S \implies \text{inv } T$*

begin

sublocale *dpll-with-backjumping*

\langle *proof* \rangle

lemma *rtrancpl-cdcl_{NOT}-inv*:

*cdcl_{NOT}** S T \implies inv S \implies inv T*
 \langle *proof* \rangle

lemma *rtrancpl-cdcl_{NOT}-no-dup*:

assumes *cdcl_{NOT}** S T and inv S*
and *no-dup (trail S)*
shows *no-dup (trail T)*
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

lemma *rtranclp-cdcl_{NOT}-all-decomposition-implies:*

assumes $cdcl_{NOT}^{**} S T$ **and** $inv S$ **and** $no_dup (trail S)$ **and**
 $all_decomposition_implies_m (clauses_{NOT} S) (get_all_marked_decomposition (trail S))$
shows
 $all_decomposition_implies_m (clauses_{NOT} T) (get_all_marked_decomposition (trail T))$
 $\langle proof \rangle$

lemma *rtranclp-cdcl_{NOT}-bj-sat-ext-iff:*

assumes $cdcl_{NOT}^{**} S T$ **and** $inv S$ **and** $no_dup (trail S)$
shows $I \models_{sextm} clauses_{NOT} S \longleftrightarrow I \models_{sextm} clauses_{NOT} T$
 $\langle proof \rangle$

definition *cdcl_{NOT}-NOT-all-inv where*

$cdcl_{NOT}-NOT-all-inv A S \longleftrightarrow (finite A \wedge inv S \wedge atms_of_mm (clauses_{NOT} S) \subseteq atms_of_ms A$
 $\wedge atm_of \text{ ' } lits_of_l (trail S) \subseteq atms_of_ms A \wedge no_dup (trail S))$

lemma *cdcl_{NOT}-NOT-all-inv:*

assumes $cdcl_{NOT}^{**} S T$ **and** $cdcl_{NOT}-NOT-all-inv A S$
shows $cdcl_{NOT}-NOT-all-inv A T$
 $\langle proof \rangle$

abbreviation *learn-or-forget where*

$learn_or_forget S T \equiv learn S T \vee forget_{NOT} S T$

lemma *rtranclp-learn-or-forget-cdcl_{NOT}:*

$learn_or_forget^{**} S T \implies cdcl_{NOT}^{**} S T$
 $\langle proof \rangle$

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 $(2 + card (atms_of_ms A)) \wedge (1 + card (atms_of_ms A))$
 $- \mu_C (1 + card (atms_of_ms A)) (2 + card (atms_of_ms A)) (trail_weight U)$
 $< (2 + card (atms_of_ms A)) \wedge (1 + card (atms_of_ms A))$
 $- \mu_C (1 + card (atms_of_ms A)) (2 + card (atms_of_ms A)) (trail_weight S)$
 $(is \text{ ?}\mu U < \text{ ?}\mu S)$
 $\langle proof \rangle$

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 $\exists j. \forall i \geq j. learn_or_forget (f i) (f (Suc i))$
 $\langle proof \rangle$

lemma *wf-cdcl_{NOT}-no-learn-and-forget-infinite-chain:*

assumes
 $no_infinite_lf: \bigwedge f j. \neg (\forall i \geq j. learn_or_forget (f i) (f (Suc i)))$
shows $wf \{ (T, S). cdcl_{NOT} S T \wedge cdcl_{NOT}-NOT-all-inv A S \}$
 $(is \text{ wf } \{ (T, S). cdcl_{NOT} S T \wedge ?inv S \})$
 $\langle proof \rangle$

lemma *inv-and-tranclp-cdcl_{NOT}-tranclp-cdcl_{NOT}-and-inv:*

$cdcl_{NOT}^{++} S T \wedge cdcl_{NOT-NOT-all-inv} A S \longleftrightarrow (\lambda S T. cdcl_{NOT} S T \wedge cdcl_{NOT-NOT-all-inv} A S)^{++} S T$

(**is** ?A \wedge ?I \longleftrightarrow ?B)

<proof>

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 $wf \{(T, S). cdcl_{NOT}^{++} S T \wedge cdcl_{NOT-NOT-all-inv} A S\}$

<proof>

lemma *cdcl_{NOT}-final-state:*

assumes

n-s: *no-step* $cdcl_{NOT} S$ **and**

inv: $cdcl_{NOT-NOT-all-inv} A S$ **and**

decomp: *all-decomposition-implies-m* ($clauses_{NOT} S$) (*get-all-marked-decomposition* ($trail S$))

shows *unsatisfiable* (*set-mset* ($clauses_{NOT} S$))

$\vee (trail S \models_{asm} clauses_{NOT} S \wedge \text{satisfiable } (set-mset (clauses_{NOT} S)))$

<proof>

lemma *full-cdcl_{NOT}-final-state:*

assumes

full: *full* $cdcl_{NOT} S T$ **and**

inv: $cdcl_{NOT-NOT-all-inv} A S$ **and**

n-d: *no-dup* ($trail S$) **and**

decomp: *all-decomposition-implies-m* ($clauses_{NOT} S$) (*get-all-marked-decomposition* ($trail S$))

shows *unsatisfiable* (*set-mset* ($clauses_{NOT} T$))

$\vee (trail T \models_{asm} clauses_{NOT} T \wedge \text{satisfiable } (set-mset (clauses_{NOT} T)))$

<proof>

end — end of *conflict-driven-clause-learning*

16.4.4 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.

16.4.5 Restricting learn and forget

locale *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt* =

dpll-state *mset-cls* *insert-cls* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-cl_{NOT}* *remove-cl_{NOT}* +

conflict-driven-clause-learning *mset-cls* *insert-cls* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-cl_{NOT}* *remove-cl_{NOT}*

inv *backjump-conds* *propagate-conds*

$\lambda C S. \text{distinct-mset } (mset-cls C) \wedge \neg \text{tautology } (mset-cls C) \wedge \text{learn-restrictions } C S \wedge$

$(\exists F K d F' C' L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge mset-cls C = C' + \{\#L\} \wedge F \models_{as} C \text{Not}$

$C')$

$\wedge C' + \{\#L\# \} \notin \# \text{ clauses}_{NOT} S)$
 $\lambda C S. \neg(\exists F' F K d L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} CNot (\text{remove1-mset } L (\text{mset-cl } C)))$
 $\wedge \text{forget-restrictions } C S$
for
 $\text{mset-cl } :: 'cls \Rightarrow 'v \text{ clause and}$
 $\text{insert-cl } :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls \text{ and}$
 $\text{remove-lit } :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls \text{ and}$
 $\text{mset-clss} :: 'clss \Rightarrow 'v \text{ clauses and}$
 $\text{union-clss} :: 'clss \Rightarrow 'clss \Rightarrow 'clss \text{ and}$
 $\text{in-clss} :: 'cls \Rightarrow 'clss \Rightarrow \text{bool and}$
 $\text{insert-clss} :: 'cls \Rightarrow 'clss \Rightarrow 'clss \text{ and}$
 $\text{remove-from-clss} :: 'cls \Rightarrow 'clss \Rightarrow 'clss \text{ and}$
 $\text{trail} :: 'st \Rightarrow ('v, \text{unit}, \text{unit}) \text{ marked-lits and}$
 $\text{raw-clauses} :: 'st \Rightarrow 'clss \text{ and}$
 $\text{prepend-trail} :: ('v, \text{unit}, \text{unit}) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st \text{ and}$
 $\text{tl-trail} :: 'st \Rightarrow 'st \text{ and}$
 $\text{add-cl}_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st \text{ and}$
 $\text{remove-cl}_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st \text{ and}$
 $\text{inv} :: 'st \Rightarrow \text{bool and}$
 $\text{backjump-conds} :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow \text{bool and}$
 $\text{propagate-conds} :: ('v, \text{unit}, \text{unit}) \text{ marked-lit} \Rightarrow 'st \Rightarrow \text{bool and}$
 $\text{learn-restrictions forget-restrictions} :: 'cls \Rightarrow 'st \Rightarrow \text{bool}$

begin

lemma $\text{cdcl}_{NOT}\text{-learn-all-induct}[\text{consumes } 1, \text{case-names } \text{dpll-bj learn forget}_{NOT}]$:

fixes $S T :: 'st$

assumes $\text{cdcl}_{NOT} S T$ **and**

$\text{dpll}: \bigwedge T. \text{dpll-bj } S T \implies P S T$ **and**

learning:

$\bigwedge C F K F' C' L T. \text{clauses}_{NOT} S \models_{pm} \text{mset-cl } C \implies$
 $\text{atms-of } (\text{mset-cl } C) \subseteq \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \implies$
 $\text{distinct-mset } (\text{mset-cl } C) \implies$
 $\neg \text{tautology } (\text{mset-cl } C) \implies$
 $\text{learn-restrictions } C S \implies$
 $\text{trail } S = F' @ \text{Marked } K () \# F \implies$
 $\text{mset-cl } C = C' + \{\#L\# \} \implies$
 $F \models_{as} CNot C' \implies$
 $C' + \{\#L\# \} \notin \# \text{ clauses}_{NOT} S \implies$
 $T \sim \text{add-cl}_{NOT} C S \implies$
 $P S T$ **and**

forgetting: $\bigwedge C T. \text{removeAll-mset } (\text{mset-cl } C) (\text{clauses}_{NOT} S) \models_{pm} \text{mset-cl } C \implies$

$C !\in! \text{raw-clauses } S \implies$
 $\neg(\exists F' F K L. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} CNot (\text{mset-cl } C - \{\#L\# \})) \implies$
 $T \sim \text{remove-cl}_{NOT} C S \implies$
 $\text{forget-restrictions } C S \implies$
 $P S T$

shows $P S T$

<proof>

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-inv}$:

$\text{cdcl}_{NOT}^{**} S T \implies \text{inv } S \implies \text{inv } T$

<proof>

lemma *learn-always-simple-clauses*:

assumes

learn: *learn S T* **and**

n-d: *no-dup (trail S)*

shows *set-mset (clauses_{NOT} T - clauses_{NOT} S)*

\subseteq *simple-cls (atms-of-mm (clauses_{NOT} S) \cup atm-of ' lits-of-l (trail S))*

<proof>

definition *conflicting-bj-cls S* \equiv

$\{C + \{\#L\# \} \mid C \text{ L. } C + \{\#L\# \} \in \# \text{ clauses}_{NOT} S \wedge \text{distinct-mset } (C + \{\#L\# \})$

$\wedge \neg \text{tautology } (C + \{\#L\# \})$

$\wedge (\exists F' K F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} C \text{Not } C)\}$

lemma *conflicting-bj-cls-remove-cls_{NOT}[simp]*:

conflicting-bj-cls (remove-cls_{NOT} C S) = conflicting-bj-cls S - {mset-cls C}

<proof>

lemma *conflicting-bj-cls-remove-cls_{NOT}'[simp]*:

$T \sim \text{remove-cls}_{NOT} C S \implies \text{conflicting-bj-cls } T = \text{conflicting-bj-cls } S - \{\text{mset-cls } C\}$

<proof>

lemma *conflicting-bj-cls-add-cls_{NOT}-state-eq*:

assumes

T: $T \sim \text{add-cls}_{NOT} C' S$ **and**

n-d: *no-dup (trail S)*

shows *conflicting-bj-cls T*

$= \text{conflicting-bj-cls } S$

$\cup (\text{if } \exists C L. \text{mset-cls } C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$

$\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} C \text{Not } C)$

$\text{then } \{\text{mset-cls } C'\} \text{ else } \{\})$

<proof>

lemma *conflicting-bj-cls-add-cls_{NOT}*:

no-dup (trail S) \implies

conflicting-bj-cls (add-cls_{NOT} C' S)

$= \text{conflicting-bj-cls } S$

$\cup (\text{if } \exists C L. \text{mset-cls } C' = C + \{\#L\# \} \wedge \text{distinct-mset } (C + \{\#L\# \}) \wedge \neg \text{tautology } (C + \{\#L\# \})$

$\wedge (\exists F' K d F. \text{trail } S = F' @ \text{Marked } K () \# F \wedge F \models_{as} C \text{Not } C)$

$\text{then } \{\text{mset-cls } C'\} \text{ else } \{\})$

<proof>

lemma *conflicting-bj-cls-incl-clauses*:

conflicting-bj-cls S \subseteq set-mset (clauses_{NOT} S)

<proof>

lemma *finite-conflicting-bj-cls[simp]*:

finite (conflicting-bj-cls S)

<proof>

lemma *learn-conflicting-increasing*:

no-dup (trail S) \implies learn S T \implies conflicting-bj-cls S \subseteq conflicting-bj-cls T

<proof>

abbreviation *conflicting-bj-cls-yet b S* \equiv

$3 \wedge b - \text{card } (\text{conflicting-bj-cls } S)$

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 *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched:*
assumes $\text{forget}_{NOT} S T$
shows $\text{conflicting-bj-clss } S = \text{conflicting-bj-clss } T$
 $\langle \text{proof} \rangle$

lemma *forget- μ_L -decrease:*
assumes $\text{forget}_{NOT}: \text{forget}_{NOT} S T$
shows $(\mu_L b T, \mu_L b S) \in \text{less-than } <*\text{lex}*> \text{less-than}$
 $\langle \text{proof} \rangle$

lemma *set-condition-or-split:*
 $\{a. (a = b \vee Q a) \wedge S a\} = (\text{if } S b \text{ then } \{b\} \text{ else } \{\}) \cup \{a. Q a \wedge S a\}$
 $\langle \text{proof} \rangle$

lemma *set-insert-neg:*
 $A \neq \text{insert } a A \longleftrightarrow a \notin A$
 $\langle \text{proof} \rangle$

lemma *learn- μ_L -decrease:*
assumes $\text{learnST}: \text{learn } S T$ **and** $n\text{-d}: \text{no-dup } (\text{trail } S)$ **and**
 $A: \text{atms-of-mm } (\text{clauses}_{NOT} S) \cup \text{atm-of } ' \text{lits-of-l } (\text{trail } S) \subseteq A$ **and**
 $\text{fin-A}: \text{finite } A$
shows $(\mu_L (\text{card } A) T, \mu_L (\text{card } A) S) \in \text{less-than } <*\text{lex}*> \text{less-than}$
 $\langle \text{proof} \rangle$

We have to assume the following:

- $\text{inv } S$: the invariant holds in the initial state.
- A is a (finite $\text{finite } A$) superset of the literals in the trail $\text{atm-of } ' \text{lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$ and in the clauses $\text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$. This can be the set of all the literals in the starting set of clauses.
- $\text{no-dup } (\text{trail } S)$: no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} **where**
 $\mu_{CDCL} A T \equiv ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))$
 $\quad - \mu_C (1 + \text{card } (\text{atms-of-ms } A)) (2 + \text{card } (\text{atms-of-ms } A)) (\text{trail-weight } T),$
 $\quad \text{conflicting-bj-clss-yet } (\text{card } (\text{atms-of-ms } A)) T, \text{card } (\text{set-mset } (\text{clauses}_{NOT} T)))$

lemma *cdcl_{NOT}-decreasing-measure:*
assumes
 $\text{cdcl}_{NOT} S T$ **and**
 $\text{inv}: \text{inv } S$ **and**
 $\text{atm-clss}: \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$ **and**
 $\text{atm-lits}: \text{atm-of } ' \text{lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A$ **and**
 $n\text{-d}: \text{no-dup } (\text{trail } S)$ **and**
 $\text{fin-A}: \text{finite } A$
shows $(\mu_{CDCL} A T, \mu_{CDCL} A S)$
 $\quad \in \text{less-than } <*\text{lex}*> (\text{less-than } <*\text{lex}*> \text{less-than})$
 $\langle \text{proof} \rangle$

lemma *wf-cdcl_{NOT}-restricted-learning:*

assumes *finite A*
shows $wf \{(T, S).$
 $(atms-of-mm \ (clauses_{NOT} \ S) \subseteq atms-of-ms \ A \wedge atm-of \ ' \ lits-of-l \ (trail \ S) \subseteq atms-of-ms \ A$
 $\wedge no-dup \ (trail \ S)$
 $\wedge inv \ S)$
 $\wedge cdcl_{NOT} \ S \ T \}$
 $\langle proof \rangle$

definition $\mu_C' :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow nat$ **where**
 $\mu_C' \ A \ T \equiv \mu_C \ (1 + card \ (atms-of-ms \ A)) \ (2 + card \ (atms-of-ms \ A)) \ (trail-weight \ T)$

definition $\mu_{CDCL}' :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow nat$ **where**
 $\mu_{CDCL}' \ A \ T \equiv$
 $((2 + card \ (atms-of-ms \ A)) \wedge (1 + card \ (atms-of-ms \ A)) - \mu_C' \ A \ T) * (1 + 3^{card \ (atms-of-ms \ A)}) * 2$
 $+ conflicting-bj-clss-yet \ (card \ (atms-of-ms \ A)) \ T * 2$
 $+ card \ (set-mset \ (clauses_{NOT} \ T))$

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 $\mu_{CDCL}' \ A \ T < \mu_{CDCL}' \ A \ S$
 $\langle proof \rangle$

lemma $cdcl_{NOT}$ -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**
 $fin-A[simp]: finite \ A$
shows $set-mset \ (clauses_{NOT} \ T) \subseteq set-mset \ (clauses_{NOT} \ S) \cup simple-clss \ A$
 $\langle proof \rangle$

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 $set-mset \ (clauses_{NOT} \ T) \subseteq set-mset \ (clauses_{NOT} \ S) \cup simple-clss \ A$
 $\langle proof \rangle$

lemma $rtrancpl-cdcl_{NOT}$ -card-clauses-bound:

assumes
 $cdcl_{NOT}^{**} \ S \ T$ **and**
 $inv \ S$ **and**

$atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$ and
 $atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$ and
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ and
 $finite: finite\ A$
shows $card\ (set\text{-}mset\ (clauses_{NOT}\ T)) \leq card\ (set\text{-}mset\ (clauses_{NOT}\ S)) + 3 \wedge (card\ A)$
 $\langle proof \rangle$

lemma $rtranc\text{-}p\text{-}cdcl_{NOT}\text{-}card\text{-}clauses\text{-}bound'$:

assumes
 $cdcl_{NOT}^{**}\ S\ T$ and
 $inv\ S$ and
 $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$ and
 $atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$ and
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ and
 $finite: finite\ A$
shows $card\ \{C \mid C. C \in \# clauses_{NOT}\ T \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\}$
 $\leq card\ \{C \mid C. C \in \# clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3 \wedge (card\ A)$
 $(is\ card\ ?T \leq card\ ?S + -)$
 $\langle proof \rangle$

lemma $rtranc\text{-}p\text{-}cdcl_{NOT}\text{-}card\text{-}simple\text{-}clauses\text{-}bound$:

assumes
 $cdcl_{NOT}^{**}\ S\ T$ and
 $inv\ S$ and
 $NA: atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \subseteq A$ and
 $MA: atm\text{-}of\ ' (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq A$ and
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ and
 $finite: finite\ A$
shows $card\ (set\text{-}mset\ (clauses_{NOT}\ T))$
 $\leq card\ \{C. C \in \# clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3 \wedge (card\ A)$
 $(is\ card\ ?T \leq card\ ?S + -)$
 $\langle proof \rangle$

definition $\mu_{CDCL}'\text{-}bound :: 'v\ literal\ multiset\ set \Rightarrow 'st \Rightarrow nat$ **where**

$\mu_{CDCL}'\text{-}bound\ A\ S =$
 $((2 + card\ (atms\text{-}of\text{-}ms\ A)) \wedge (1 + card\ (atms\text{-}of\text{-}ms\ A))) * (1 + 3 \wedge card\ (atms\text{-}of\text{-}ms\ A)) * 2$
 $+ 2 * 3 \wedge (card\ (atms\text{-}of\text{-}ms\ A))$
 $+ card\ \{C. C \in \# clauses_{NOT}\ S \wedge (tautology\ C \vee \neg distinct\text{-}mset\ C)\} + 3 \wedge (card\ (atms\text{-}of\text{-}ms\ A))$

lemma $\mu_{CDCL}'\text{-}bound\text{-}reduce\text{-}trail\text{-}to_{NOT}[simp]$:

$\mu_{CDCL}'\text{-}bound\ A\ (reduce\text{-}trail\text{-}to_{NOT}\ M\ S) = \mu_{CDCL}'\text{-}bound\ A\ S$
 $\langle proof \rangle$

lemma $rtranc\text{-}p\text{-}cdcl_{NOT}\text{-}\mu_{CDCL}'\text{-}bound\text{-}reduce\text{-}trail\text{-}to_{NOT}$:

assumes
 $cdcl_{NOT}^{**}\ S\ T$ and
 $inv\ S$ 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
 $n\text{-}d: no\text{-}dup\ (trail\ S)$ and
 $finite: finite\ (atms\text{-}of\text{-}ms\ A)$ and
 $U: U \sim reduce\text{-}trail\text{-}to_{NOT}\ M\ T$
shows $\mu_{CDCL}'\ A\ U \leq \mu_{CDCL}'\text{-}bound\ A\ S$
 $\langle proof \rangle$

lemma *rtrancp-cdcl_{NOT}- μ_{CDCL} '-bound*:

assumes

*cdcl_{NOT}** S T and*

inv S and

atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A and

atm-of '(lits-of-l (trail S)) \subseteq atms-of-ms A and

n-d: no-dup (trail S) and

finite: finite (atms-of-ms A)

shows $\mu_{CDCL}' A T \leq \mu_{CDCL}'\text{-bound } A S$

<proof>

lemma *rtrancp- μ_{CDCL} '-bound-decreasing*:

assumes

*cdcl_{NOT}** S T and*

inv S and

atms-of-mm (clauses_{NOT} S) \subseteq atms-of-ms A and

atm-of '(lits-of-l (trail S)) \subseteq atms-of-ms A and

n-d: no-dup (trail S) and

finite[simp]: finite (atms-of-ms A)

shows $\mu_{CDCL}'\text{-bound } A T \leq \mu_{CDCL}'\text{-bound } A S$

<proof>

end — end of *conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt*

16.5 CDCL with restarts

16.5.1 Definition

locale *restart-ops* =

fixes

cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool and

restart :: 'st \Rightarrow 'st \Rightarrow bool

begin

inductive *cdcl_{NOT}-raw-restart* :: 'st \Rightarrow 'st \Rightarrow bool **where**

cdcl_{NOT} S T \Longrightarrow cdcl_{NOT}-raw-restart S T |

restart S T \Longrightarrow cdcl_{NOT}-raw-restart S T

end

locale *conflict-driven-clause-learning-with-restarts* =

conflict-driven-clause-learning mset-cls insert-cls remove-lit

mset-clss union-clss in-clss insert-clss remove-from-clss

trail raw-clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}

inv backjump-conds propagate-conds learn-cond forget-cond

for

mset-cls :: 'cls \Rightarrow 'v clause and

insert-cls :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and

remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls and

mset-clss:: 'clss \Rightarrow 'v clauses and

union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss and

in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool and

insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and

remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss and

trail :: 'st \Rightarrow ('v, unit, unit) marked-lits and

raw-clauses :: 'st \Rightarrow 'clss and

prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st and

```

tl-trail :: 'st ⇒ 'st and
add-clNOT :: 'cls ⇒ 'st ⇒ 'st and
remove-clNOT :: 'cls ⇒ 'st ⇒ 'st and
inv :: 'st ⇒ bool and
backjump-conds :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool and
propagate-conds :: ('v, unit, unit) marked-lit ⇒ 'st ⇒ bool and
learn-cond forget-cond :: 'cls ⇒ 'st ⇒ bool
begin

lemma cdclNOT-iff-cdclNOT-raw-restart-no-restarts:
  cdclNOT S T ⇔ restart-ops.cdclNOT-raw-restart cdclNOT (λ- -. False) S T
  (is ?C S T ⇔ ?R S T)
  ⟨proof⟩

lemma cdclNOT-cdclNOT-raw-restart:
  cdclNOT S T ⇒ restart-ops.cdclNOT-raw-restart cdclNOT restart S T
  ⟨proof⟩
end

```

16.5.2 Increasing restarts

To add restarts we need some assumptions on the predicate (called *cdcl_{NOT}* here):

- a function f that is strictly monotonic. The first step is actually only used as a restart to clean the state (e.g. to ensure that the trail is empty). Then we assume that $(1::'a) \leq f$ n for $(1::'a) \leq n$: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...
- a measure μ : it should decrease under the assumptions *bound-inv*, whenever a *cdcl_{NOT}* or a *restart* is done. A parameter is given to μ : for conflict- driven clause learning, it is an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.
- we also assume that the measure decrease after any *cdcl_{NOT}* step.
- an invariant on the states *cdcl_{NOT}-inv* that also holds after restarts.
- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ -bound taking the same parameter as μ and the initial state of the considered *cdcl_{NOT}* chain.

```

locale cdclNOT-increasing-restarts-ops =
  restart-ops cdclNOT restart for
    restart :: 'st ⇒ 'st ⇒ bool and
    cdclNOT :: 'st ⇒ 'st ⇒ bool +
  fixes
    f :: nat ⇒ nat and
    bound-inv :: 'bound ⇒ 'st ⇒ bool and
    μ :: 'bound ⇒ 'st ⇒ nat and
    cdclNOT-inv :: 'st ⇒ bool and
    μ-bound :: 'bound ⇒ 'st ⇒ nat
  assumes
    f: unbounded f and
    f-ge-1: ∧ n. n ≥ 1 ⇒ f n ≠ 0 and

```

bound-inv: $\bigwedge A S T. \text{cdcl}_{NOT}\text{-inv } S \implies \text{bound-inv } A S \implies \text{cdcl}_{NOT} S T \implies \text{bound-inv } A T$ **and**
cdcl_{NOT}-measure: $\bigwedge A S T. \text{cdcl}_{NOT}\text{-inv } S \implies \text{bound-inv } A S \implies \text{cdcl}_{NOT} S T \implies \mu A T < \mu$
A S and
measure-bound2: $\bigwedge A T U. \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A T \implies \text{cdcl}_{NOT}^{**} T U$
 $\implies \mu A U \leq \mu\text{-bound } A T$ **and**
measure-bound4: $\bigwedge A T U. \text{cdcl}_{NOT}\text{-inv } T \implies \text{bound-inv } A T \implies \text{cdcl}_{NOT}^{**} T U$
 $\implies \mu\text{-bound } A U \leq \mu\text{-bound } A T$ **and**
cdcl_{NOT}-restart-inv: $\bigwedge A U V. \text{cdcl}_{NOT}\text{-inv } U \implies \text{restart } U V \implies \text{bound-inv } A U \implies \text{bound-inv}$
A V
and
exists-bound: $\bigwedge R S. \text{cdcl}_{NOT}\text{-inv } R \implies \text{restart } R S \implies \exists A. \text{bound-inv } A S$ **and**
cdcl_{NOT}-inv: $\bigwedge S T. \text{cdcl}_{NOT}\text{-inv } S \implies \text{cdcl}_{NOT} S T \implies \text{cdcl}_{NOT}\text{-inv } T$ **and**
cdcl_{NOT}-inv-restart: $\bigwedge S T. \text{cdcl}_{NOT}\text{-inv } S \implies \text{restart } S T \implies \text{cdcl}_{NOT}\text{-inv } T$
begin

lemma *cdcl_{NOT}-cdcl_{NOT}-inv*:

assumes
 $(\text{cdcl}_{NOT} \rightsquigarrow n) S T$ **and**
 $\text{cdcl}_{NOT}\text{-inv } S$
shows $\text{cdcl}_{NOT}\text{-inv } T$
 $\langle \text{proof} \rangle$

lemma *cdcl_{NOT}-bound-inv*:

assumes
 $(\text{cdcl}_{NOT} \rightsquigarrow n) S T$ **and**
 $\text{cdcl}_{NOT}\text{-inv } S$
 $\text{bound-inv } A S$
shows $\text{bound-inv } A T$
 $\langle \text{proof} \rangle$

lemma *rtrancpl-cdcl_{NOT}-cdcl_{NOT}-inv*:

assumes
 $\text{cdcl}_{NOT}^{**} S T$ **and**
 $\text{cdcl}_{NOT}\text{-inv } S$
shows $\text{cdcl}_{NOT}\text{-inv } T$
 $\langle \text{proof} \rangle$

lemma *rtrancpl-cdcl_{NOT}-bound-inv*:

assumes
 $\text{cdcl}_{NOT}^{**} S T$ **and**
 $\text{bound-inv } A S$ **and**
 $\text{cdcl}_{NOT}\text{-inv } S$
shows $\text{bound-inv } A T$
 $\langle \text{proof} \rangle$

lemma *cdcl_{NOT}-comp-n-le*:

assumes
 $(\text{cdcl}_{NOT} \rightsquigarrow (\text{Suc } n)) S T$ **and**
 $\text{bound-inv } A S$
 $\text{cdcl}_{NOT}\text{-inv } S$
shows $\mu A T < \mu A S - n$
 $\langle \text{proof} \rangle$

lemma *wf-cdcl_{NOT}*:

$\text{wf } \{(T, S). \text{cdcl}_{NOT} S T \wedge \text{cdcl}_{NOT}\text{-inv } S \wedge \text{bound-inv } A S\}$ (**is** $\text{wf } ?A$)

$\langle proof \rangle$

lemma *rtrancplp-cdcl_{NOT}-measure:*

assumes

*cdcl_{NOT}** S T and*

bound-inv A S and

cdcl_{NOT}-inv S

shows $\mu A T \leq \mu A S$

$\langle proof \rangle$

lemma *cdcl_{NOT}-comp-bounded:*

assumes

bound-inv A S and cdcl_{NOT}-inv S and $m \geq 1 + \mu A S$

shows $\neg(cdcl_{NOT} \rightsquigarrow m) S T$

$\langle proof \rangle$

- $f n < m$ ensures that at least one step has been done.

inductive *cdcl_{NOT}-restart where*

restart-step: $(cdcl_{NOT} \rightsquigarrow m) S T \implies m \geq f n \implies restart T U$

$\implies cdcl_{NOT}\text{-restart } (S, n) (U, Suc n) \mid$

restart-full: $full1\ cdcl_{NOT} S T \implies cdcl_{NOT}\text{-restart } (S, n) (T, Suc n)$

lemmas *cdcl_{NOT}-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)*

$\langle proof \rangle$

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)*

$\langle proof \rangle$

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)*

$\langle proof \rangle$

lemma *rtrancplp-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)*

$\langle proof \rangle$

lemma *rtrancplp-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 $bound-inv A (fst T)$
 $\langle proof \rangle$

lemma $cdcl_{NOT-with-restart-increasing-number}$:

$cdcl_{NOT-restart} S T \implies snd T = 1 + snd S$

$\langle proof \rangle$

end

locale $cdcl_{NOT-increasing-restarts} =$

$cdcl_{NOT-increasing-restarts-ops}$ $restart$ $cdcl_{NOT}$ f $bound-inv$ μ $cdcl_{NOT-inv}$ $\mu-bound$ $+$
 $dpll-state$ $mset-cls$ $insert-cls$ $remove-lit$

$mset-clss$ $union-clss$ $in-clss$ $insert-clss$ $remove-from-clss$

$trail$ $raw-clauses$ $prepend-trail$ $tl-trail$ $add-cls_{NOT}$ $remove-cls_{NOT}$

for

$mset-cls :: 'cls \Rightarrow 'v \text{ clause}$ **and**

$insert-cls :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$ **and**

$remove-lit :: 'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$ **and**

$mset-clss :: 'clss \Rightarrow 'v \text{ clauses}$ **and**

$union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**

$in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool$ **and**

$insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

$remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

$trail :: 'st \Rightarrow ('v, unit, unit) \text{ marked-lits}$ **and**

$raw-clauses :: 'st \Rightarrow 'clss$ **and**

$prepend-trail :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st$ **and**

$tl-trail :: 'st \Rightarrow 'st$ **and**

$add-cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**

$remove-cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**

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

$restart :: 'st \Rightarrow 'st \Rightarrow bool$ **and**

$bound-inv :: 'bound \Rightarrow 'st \Rightarrow bool$ **and**

$\mu :: 'bound \Rightarrow 'st \Rightarrow nat$ **and**

$cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$ **and**

$cdcl_{NOT-inv} :: 'st \Rightarrow bool$ **and**

$\mu-bound :: 'bound \Rightarrow 'st \Rightarrow nat +$

assumes

$measure-bound: \bigwedge A T V n. cdcl_{NOT-inv} T \implies bound-inv A T$

$\implies cdcl_{NOT-restart} (T, n) (V, Suc n) \implies \mu A V \leq \mu-bound A T$ **and**

$cdcl_{NOT-raw-restart-\mu-bound}$:

$cdcl_{NOT-restart} (T, a) (V, b) \implies cdcl_{NOT-inv} T \implies bound-inv A T$

$\implies \mu-bound A V \leq \mu-bound A T$

begin

lemma $rtrancp-cdcl_{NOT-raw-restart-\mu-bound}$:

$cdcl_{NOT-restart}^{**} (T, a) (V, b) \implies cdcl_{NOT-inv} T \implies bound-inv A T$

$\implies \mu-bound A V \leq \mu-bound A T$

$\langle proof \rangle$

lemma $cdcl_{NOT-raw-restart-measure-bound}$:

$cdcl_{NOT-restart} (T, a) (V, b) \implies cdcl_{NOT-inv} T \implies bound-inv A T$

$\implies \mu A V \leq \mu-bound A T$

$\langle proof \rangle$

lemma *rtrancpl-cdcl_{NOT}-raw-restart-measure-bound*:
 $cdcl_{NOT}\text{-restart}^{**} (T, a) (V, b) \implies cdcl_{NOT}\text{-inv } T \implies bound\text{-inv } A \ T$
 $\implies \mu \ A \ V \leq \mu\text{-bound } A \ T$
 $\langle proof \rangle$

lemma *wf-cdcl_{NOT}-restart*:
 $wf \{ (T, S). \ cdcl_{NOT}\text{-restart } S \ T \wedge cdcl_{NOT}\text{-inv } (fst \ S) \} \text{ (is } wf \ ?A)$
 $\langle proof \rangle$

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**

$f \ (snd \ S) > \mu\text{-bound } A \ (fst \ S)$

shows *full1* *cdcl_{NOT}* $(fst \ S) \ (fst \ T)$

$\langle proof \rangle$

lemma *rtrancpl-cdcl_{NOT}-with-inv-inv-rtrancpl-cdcl_{NOT}*:

assumes

inv: *cdcl_{NOT}-inv* S **and**

binv: *bound-inv* $A \ S$

shows $(\lambda S \ T. \ cdcl_{NOT} \ S \ T \wedge cdcl_{NOT}\text{-inv } S \wedge bound\text{-inv } A \ S)^{**} \ S \ T \longleftrightarrow cdcl_{NOT}^{**} \ S \ T$

(is $?A^{**} \ S \ T \longleftrightarrow ?B^{**} \ S \ T$)

$\langle proof \rangle$

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}* $(fst \ S)$

$\langle proof \rangle$

end

16.6 Merging backjump and learning

locale *cdcl_{NOT}-merge-bj-learn-ops* =

decide-ops *mset-cls* *insert-cls* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls_{NOT}* *remove-cls_{NOT}* +

forget-ops *mset-cls* *insert-cls* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls_{NOT}* *remove-cls_{NOT}* *forget-cond* +

propagate-ops *mset-cls* *insert-cls* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-cls_{NOT}* *remove-cls_{NOT}* *propagate-conds*

for

mset-cls :: $'cls \Rightarrow 'v \text{ clause}$ **and**

insert-cls :: $'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$ **and**

remove-lit :: $'v \text{ literal} \Rightarrow 'cls \Rightarrow 'cls$ **and**

mset-clss:: $'clss \Rightarrow 'v \text{ clauses}$ **and**

union-clss :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**

in-clss :: $'cls \Rightarrow 'clss \Rightarrow \text{bool}$ **and**

insert-clss :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

```

remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
raw-clauses :: 'st  $\Rightarrow$  'clss and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool +
fixes backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool
begin

```

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' @ Marked K () # F
 $\Rightarrow$  no-dup (trail S)
 $\Rightarrow$  T  $\sim$  prepend-trail (Propagated L ()) (reduce-trail-toNOT F (add-clNOT C'' S))
 $\Rightarrow$  C  $\in$  # clausesNOT S
 $\Rightarrow$  trail S  $\models_{as}$  CNot C
 $\Rightarrow$  undefined-lit F L
 $\Rightarrow$  atm-of L  $\in$  atms-of-mm (clausesNOT S)  $\cup$  atm-of ' (lits-of-l (trail S))
 $\Rightarrow$  clausesNOT S  $\models_{pm}$  C' + {#L#}
 $\Rightarrow$  mset-cls C'' = C' + {#L#}
 $\Rightarrow$  F  $\models_{as}$  CNot C'
 $\Rightarrow$  backjump-l-cond C C' L S T
 $\Rightarrow$  backjump-l S T

```

Avoid (meaningless) simplification 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 cdcl_{NOT}-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 cdcl_{NOT}-merged-bj-learn-no-dup-inv:

```

cdclNOT-merged-bj-learn S T  $\Rightarrow$  no-dup (trail S)  $\Rightarrow$  no-dup (trail T)
<proof>
end

```

locale cdcl_{NOT}-merge-bj-learn-proxy =

```

cdclNOT-merge-bj-learn-ops mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT propagate-conds
forget-cond
 $\lambda$ C C' L' S T. backjump-l-cond C C' L' S T
 $\wedge$  distinct-mset (C' + {#L'#})  $\wedge$   $\neg$ tautology (C' + {#L'#})
for
mset-cls :: 'cls  $\Rightarrow$  'v clause and
insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and

```

remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**
mset-clss:: 'clss \Rightarrow 'v clauses **and**
union-clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss **and**
in-clss :: 'cls \Rightarrow 'clss \Rightarrow bool **and**
insert-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**
remove-from-clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**
trail :: 'st \Rightarrow ('v, unit, unit) marked-lits **and**
raw-clauses :: 'st \Rightarrow 'clss **and**
prepend-trail :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-trail :: 'st \Rightarrow 'st **and**
add-clss_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
remove-clss_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
propagate-conds :: ('v, unit, unit) marked-lit \Rightarrow 'st \Rightarrow bool **and**
forget-cond :: 'cls \Rightarrow 'st \Rightarrow bool **and**
backjump-l-cond :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool +

fixes

inv :: 'st \Rightarrow bool

assumes

bj-merge-can-jump:

$\bigwedge S C F' K F L.$

inv S

\Rightarrow trail *S* = *F'* @ Marked *K* () # *F*

\Rightarrow *C* \in # clauses_{NOT} *S*

\Rightarrow trail *S* \models_{as} CNot *C*

\Rightarrow undefined-lit *F L*

\Rightarrow atm-of *L* \in atms-of-mm (clauses_{NOT} *S*) \cup atm-of ' (lits-of-l (*F'* @ Marked *K* () # *F*))

\Rightarrow clauses_{NOT} *S* \models_{pm} *C'* + {#*L*#}

\Rightarrow *F* \models_{as} CNot *C'*

\Rightarrow \neg no-step backjump-l *S* **and**

cdcl-merged-inv: $\bigwedge S T. \text{cdcl}_{NOT}\text{-merged-bj-learn } S T \Rightarrow \text{inv } S \Rightarrow \text{inv } T$

begin

abbreviation *backjump-conds* :: 'v clause \Rightarrow 'v clause \Rightarrow 'v literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool

where

backjump-conds $\equiv \lambda C C' L' S T. \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$

Without additional knowledge on *backjump-l-cond*, it is impossible to have the same invariant.

sublocale *dpll-with-backjumping-ops* *mset-clss* *insert-clss* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-clss_{NOT}* *remove-clss_{NOT}* *inv*

backjump-conds *propagate-conds*

<proof>

end

locale *cdcl_{NOT}-merge-bj-learn-proxy2* =

cdcl_{NOT}-merge-bj-learn-proxy *mset-clss* *insert-clss* *remove-lit*

mset-clss *union-clss* *in-clss* *insert-clss* *remove-from-clss*

trail *raw-clauses* *prepend-trail* *tl-trail* *add-clss_{NOT}* *remove-clss_{NOT}*

propagate-conds *forget-cond* *backjump-l-cond* *inv*

for

mset-clss :: 'cls \Rightarrow 'v clause **and**

insert-clss :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**

remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**

mset-clss:: 'clss \Rightarrow 'v clauses **and**

```

union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
raw-clauses :: 'st  $\Rightarrow$  'clss and
prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
tl-trail :: 'st  $\Rightarrow$  'st and
add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool and
backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
inv :: 'st  $\Rightarrow$  bool

```

begin

```

sublocale conflict-driven-clause-learning-ops mset-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT
  inv backjump-conds propagate-conds
 $\lambda C$  -. distinct-mset (mset-cls C)  $\wedge$   $\neg$ tautology (mset-cls C)
forget-cond
<proof>

```

end

```

locale cdclNOT-merge-bj-learn =
  cdclNOT-merge-bj-learn-proxy2 mset-cls insert-cls remove-lit
  mset-clss union-clss in-clss insert-clss remove-from-clss
  trail raw-clauses prepend-trail tl-trail add-clNOT remove-clNOT
  propagate-conds forget-cond backjump-l-cond inv
for
  mset-cls :: 'cls  $\Rightarrow$  'v clause and
  insert-cls :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  remove-lit :: 'v literal  $\Rightarrow$  'cls  $\Rightarrow$  'cls and
  mset-clss:: 'clss  $\Rightarrow$  'v clauses and
  union-clss :: 'clss  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  in-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  bool and
  insert-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  remove-from-clss :: 'cls  $\Rightarrow$  'clss  $\Rightarrow$  'clss and
  trail :: 'st  $\Rightarrow$  ('v, unit, unit) marked-lits and
  raw-clauses :: 'st  $\Rightarrow$  'clss and
  prepend-trail :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  'st and
  tl-trail :: 'st  $\Rightarrow$  'st and
  add-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  remove-clNOT :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  'st and
  backjump-l-cond :: 'v clause  $\Rightarrow$  'v clause  $\Rightarrow$  'v literal  $\Rightarrow$  'st  $\Rightarrow$  'st  $\Rightarrow$  bool and
  propagate-conds :: ('v, unit, unit) marked-lit  $\Rightarrow$  'st  $\Rightarrow$  bool and
  forget-cond :: 'cls  $\Rightarrow$  'st  $\Rightarrow$  bool and
  inv :: 'st  $\Rightarrow$  bool +

```

assumes

```

  dpll-merge-bj-inv:  $\bigwedge S T. \text{dpll-bj } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$  and
  learn-inv:  $\bigwedge S T. \text{learn } S T \Longrightarrow \text{inv } S \Longrightarrow \text{inv } T$ 

```

begin

sublocale

conflict-driven-clause-learning mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
trail raw-clauses prepend-trail tl-trail add-cl_{NOT} remove-cl_{NOT}
inv backjump-conds propagate-conds
 $\lambda C \cdot \text{distinct-mset (mset-cl_s C)} \wedge \neg \text{tautology (mset-cl_s C)}$
forget-cond
 <proof>

lemma *backjump-l-learn-backjump:*

assumes *bt: backjump-l S T and inv: inv S and n-d: no-dup (trail S)*

shows $\exists C' L D. \text{learn } S \text{ (add-cl_{sNOT} D S)}$

$\wedge \text{mset-cl_s D} = (C' + \{\#L\# \})$

$\wedge \text{backjump (add-cl_{sNOT} D S) T}$

$\wedge \text{atms-of (C' + \{\#L\# \})} \subseteq \text{atms-of-mm (clauses}_{NOT} S) \cup \text{atm-of ' (lits-of-l (trail S))}$

<proof>

lemma *cdcl_{NOT}-merged-bj-learn-is-tranclp-cdcl_{NOT}:*

$\text{cdcl}_{NOT}\text{-merged-bj-learn } S T \implies \text{inv } S \implies \text{no-dup (trail S)} \implies \text{cdcl}_{NOT}^{++} S T$

<proof>

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}-and-inv:*

$\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T \implies \text{inv } S \implies \text{no-dup (trail S)} \implies \text{cdcl}_{NOT}^{**} S T \wedge \text{inv } T$

<proof>

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-is-rtranclp-cdcl_{NOT}:*

$\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T \implies \text{inv } S \implies \text{no-dup (trail S)} \implies \text{cdcl}_{NOT}^{**} S T$

<proof>

lemma *rtranclp-cdcl_{NOT}-merged-bj-learn-inv:*

$\text{cdcl}_{NOT}\text{-merged-bj-learn}^{**} S T \implies \text{inv } S \implies \text{no-dup (trail S)} \implies \text{inv } T$

<proof>

definition $\mu_C' :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$ **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 $\mu_{CDCL}'\text{-merged} :: 'v \text{ literal multiset set} \Rightarrow 'st \Rightarrow \text{nat}$ **where**

$\mu_{CDCL}'\text{-merged } A T \equiv$

$((2 + \text{card (atms-of-ms A)}) \wedge (1 + \text{card (atms-of-ms A)}) - \mu_C' A T) * 2 + \text{card (set-mset (clauses}_{NOT} T))$

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 $\mu_{CDCL}'\text{-merged } A T < \mu_{CDCL}'\text{-merged } A S$

<proof>

lemma *wf-cdcl_{NOT}-merged-bj-learn:*

assumes

fin-A: finite A

shows *wf {(T, S)}.*

$(inv\ S \wedge atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$
 $\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}$
 $\langle proof \rangle$

lemma *trancpl-cdcl_{NOT}-cdcl_{NOT}-trancpl*:

assumes

cdcl_{NOT}-merged-bj-learn⁺⁺ *S T* **and**

inv: *inv S* **and**

atm-clss: *atms-of-mm (clauses_{NOT} S) ⊆ atms-of-ms A* **and**

atm-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*

shows $(T, S) \in \{(T, S).$

$(inv\ S \wedge atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$

$\wedge cdcl_{NOT}\text{-merged-bj-learn}\ S\ T\}^+ \text{ (is - } \in ?P^+)$

$\langle proof \rangle$

lemma *wf-trancpl-cdcl_{NOT}-merged-bj-learn*:

assumes *finite A*

shows *wf* $\{(T, S).$

$(inv\ S \wedge atms-of-mm\ (clauses_{NOT}\ S) \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of-l\ (trail\ S) \subseteq atms-of-ms\ A$
 $\wedge no-dup\ (trail\ S))$

$\wedge cdcl_{NOT}\text{-merged-bj-learn}^{++}\ S\ T\}$

$\langle proof \rangle$

lemma *backjump-no-step-backjump-l*:

backjump S T $\implies inv\ S \implies \neg no\text{-step}\ backjump\text{-l}\ S$

$\langle proof \rangle$

lemma *cdcl_{NOT}-merged-bj-learn-final-state*:

fixes *A* :: '*v* literal multiset set **and** *S T* :: '*st*

assumes

n-s: *no-step cdcl_{NOT}-merged-bj-learn S* **and**

atms-S: *atms-of-mm (clauses_{NOT} 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**

finite A **and**

inv: *inv S* **and**

decomp: *all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))*

shows *unsatisfiable (set-mset (clauses_{NOT} S))*

$\vee (trail\ S \models_{asm}\ clauses_{NOT}\ S \wedge satisfiable\ (set\text{-}mset\ (clauses_{NOT}\ S)))$

$\langle proof \rangle$

lemma *full-cdcl_{NOT}-merged-bj-learn-final-state*:

fixes *A* :: '*v* literal multiset set **and** *S T* :: '*st*

assumes

full: *full cdcl_{NOT}-merged-bj-learn S T* **and**

atms-S: *atms-of-mm (clauses_{NOT} 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**

finite A **and**

inv: *inv S* **and**

decomp: *all-decomposition-implies-m (clauses_{NOT} S) (get-all-marked-decomposition (trail S))*

shows *unsatisfiable* (*set-mset* (*clauses*_{NOT} *T*))
 \vee (*trail* *T* \models_{asm} *clauses*_{NOT} *T* \wedge *satisfiable* (*set-mset* (*clauses*_{NOT} *T*)))
<proof>

end

16.7 Instantiations

In this section, we instantiate the previous locales to ensure that the assumption are not contradictory.

locale *cdcl*_{NOT}-*with-backtrack-and-restarts* =
conflict-driven-clause-learning-learning-before-backjump-only-distinct-learnt
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
*trail raw-clauses prepend-trail tl-trail add-cls*_{NOT} *remove-cls*_{NOT}
inv backjump-conds propagate-conds learn-restrictions forget-restrictions
for
mset-cls :: '*cls* \Rightarrow '*v clause* **and**
insert-cls :: '*v literal* \Rightarrow '*cls* \Rightarrow '*cls* **and**
remove-lit :: '*v literal* \Rightarrow '*cls* \Rightarrow '*cls* **and**
mset-clss:: '*clss* \Rightarrow '*v clauses* **and**
union-clss :: '*clss* \Rightarrow '*clss* \Rightarrow '*clss* **and**
in-clss :: '*cls* \Rightarrow '*clss* \Rightarrow *bool* **and**
insert-clss :: '*cls* \Rightarrow '*clss* \Rightarrow '*clss* **and**
remove-from-clss :: '*cls* \Rightarrow '*clss* \Rightarrow '*clss* **and**
trail :: '*st* \Rightarrow ('*v*, *unit*, *unit*) *marked-lits* **and**
raw-clauses :: '*st* \Rightarrow '*clss* **and**
prepend-trail :: ('*v*, *unit*, *unit*) *marked-lit* \Rightarrow '*st* \Rightarrow '*st* **and**
tl-trail :: '*st* \Rightarrow '*st* **and**
*add-cls*_{NOT} :: '*cls* \Rightarrow '*st* \Rightarrow '*st* **and**
*remove-cls*_{NOT} :: '*cls* \Rightarrow '*st* \Rightarrow '*st* **and**
inv :: '*st* \Rightarrow *bool* **and**
backjump-conds :: '*v clause* \Rightarrow '*v clause* \Rightarrow '*v literal* \Rightarrow '*st* \Rightarrow '*st* \Rightarrow *bool* **and**
propagate-conds :: ('*v*, *unit*, *unit*) *marked-lit* \Rightarrow '*st* \Rightarrow *bool* **and**
learn-restrictions forget-restrictions :: '*cls* \Rightarrow '*st* \Rightarrow *bool*
+
fixes *f* :: *nat* \Rightarrow *nat*
assumes
unbounded: *unbounded* *f* **and** *f-ge-1*: $\bigwedge n. n \geq 1 \Rightarrow f\ n \geq 1$ **and**
inv-restart: $\bigwedge S\ T. inv\ S \Rightarrow T \sim reduce-trail-to_{NOT} ([::'a\ list)\ S \Rightarrow inv\ T$
begin

lemma *bound-inv-inv*:

assumes
inv *S* **and**
n-d: *no-dup* (*trail* *S*) **and**
atms-clss-S-A: *atms-of-mm* (*clauses*_{NOT} *S*) \subseteq *atms-of-ms* *A* **and**
atms-trail-S-A: *atm-of* ' *lits-of-l* (*trail* *S*) \subseteq *atms-of-ms* *A* **and**
finite *A* **and**
*cdcl*_{NOT}: *cdcl*_{NOT} *S* *T*
shows
atms-of-mm (*clauses*_{NOT} *T*) \subseteq *atms-of-ms* *A* **and**
atm-of ' *lits-of-l* (*trail* *T*) \subseteq *atms-of-ms* *A* **and**
finite *A*
<proof>

sublocale *cdcl_{NOT}-increasing-restarts-ops* $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}) S \text{ cdcl}_{NOT} f$
 $\lambda A S. \text{atms-of-mm} (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A \wedge \text{atm-of ' lits-of-l} (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge$
finite A
 $\mu_{CDCL}' \lambda S. \text{inv } S \wedge \text{no-dup} (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
 $\langle \text{proof} \rangle$

lemma *cdcl_{NOT}-with-restart- μ_{CDCL}' -le- μ_{CDCL}' -bound:*

assumes

cdcl_{NOT}: *cdcl_{NOT}-restart* $(T, a) (V, b)$ **and**

cdcl_{NOT}-inv:

inv T

no-dup $(\text{trail } T)$ **and**

bound-inv:

atms-of-mm $(\text{clauses}_{NOT} T) \subseteq \text{atms-of-ms } A$

atm-of ' lits-of-l $(\text{trail } T) \subseteq \text{atms-of-ms } A$

finite A

shows $\mu_{CDCL}' A V \leq \mu_{CDCL}'\text{-bound } A T$

$\langle \text{proof} \rangle$

lemma *cdcl_{NOT}-with-restart- μ_{CDCL}' -bound-le- μ_{CDCL}' -bound:*

assumes

cdcl_{NOT}: *cdcl_{NOT}-restart* $(T, a) (V, b)$ **and**

cdcl_{NOT}-inv:

inv T

no-dup $(\text{trail } T)$ **and**

bound-inv:

atms-of-mm $(\text{clauses}_{NOT} T) \subseteq \text{atms-of-ms } A$

atm-of ' lits-of-l $(\text{trail } T) \subseteq \text{atms-of-ms } A$

finite A

shows $\mu_{CDCL}'\text{-bound } A V \leq \mu_{CDCL}'\text{-bound } A T$

$\langle \text{proof} \rangle$

sublocale *cdcl_{NOT}-increasing-restarts* - - - - -

f
 $\lambda S T. T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}) S$
 $\lambda A S. \text{atms-of-mm} (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of-l} (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}' \text{ cdcl}_{NOT}$
 $\lambda S. \text{inv } S \wedge \text{no-dup} (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
 $\langle \text{proof} \rangle$

lemma *cdcl_{NOT}-restart-all-decomposition-implies:*

assumes *cdcl_{NOT}-restart* $S T$ **and**

inv $(\text{fst } S)$ **and**

no-dup $(\text{trail } (\text{fst } S))$

all-decomposition-implies-m $(\text{clauses}_{NOT} (\text{fst } S)) (\text{get-all-marked-decomposition } (\text{trail } (\text{fst } S)))$

shows

all-decomposition-implies-m $(\text{clauses}_{NOT} (\text{fst } T)) (\text{get-all-marked-decomposition } (\text{trail } (\text{fst } T)))$

$\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_{NOT}-restart-all-decomposition-implies:*

assumes *cdcl_{NOT}-restart*** $S T$ **and**

inv: *inv* (*fst* *S*) **and**
n-d: *no-dup* (*trail* (*fst* *S*)) **and**
decomp:
all-decomposition-implies-m (*clauses*_{NOT} (*fst* *S*)) (*get-all-marked-decomposition* (*trail* (*fst* *S*)))
shows
all-decomposition-implies-m (*clauses*_{NOT} (*fst* *T*)) (*get-all-marked-decomposition* (*trail* (*fst* *T*)))
 ⟨*proof*⟩

lemma *cdcl*_{NOT}-restart-sat-ext-iff:
assumes
st: *cdcl*_{NOT}-restart *S* *T* **and**
n-d: *no-dup* (*trail* (*fst* *S*)) **and**
inv: *inv* (*fst* *S*)
shows $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } T)$
 ⟨*proof*⟩

lemma *rtrancpl-cdcl*_{NOT}-restart-sat-ext-iff:
fixes *S* *T* :: 'st × nat
assumes
st: *cdcl*_{NOT}-restart** *S* *T* **and**
n-d: *no-dup* (*trail* (*fst* *S*)) **and**
inv: *inv* (*fst* *S*)
shows $I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{\text{NOT}} (\text{fst } T)$
 ⟨*proof*⟩

theorem *full-cdcl*_{NOT}-restart-backjump-final-state:
fixes *A* :: 'v literal multiset set **and** *S* *T* :: 'st
assumes
full: *full cdcl*_{NOT}-restart (*S*, *n*) (*T*, *m*) **and**
atms-S: *atms-of-mm* (*clauses*_{NOT} *S*) ⊆ *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* (*clauses*_{NOT} *S*) (*get-all-marked-decomposition* (*trail* *S*))
shows *unsatisfiable* (*set-mset* (*clauses*_{NOT} *S*))
 ∨ (*lits-of-l* (*trail* *T*) $\models_{\text{sextm}} \text{clauses}_{\text{NOT}} S \wedge \text{satisfiable} (\text{set-mset} (\text{clauses}_{\text{NOT}} S))$)
 ⟨*proof*⟩
end — end of *cdcl*_{NOT}-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 *cdcl*_{NOT}-merge-bj-learn-with-backtrack-restarts =
*cdcl*_{NOT}-merge-bj-learn *mset-cls insert-cls remove-lit*
mset-clss union-clss in-clss insert-clss remove-from-clss
*trail raw-clauses prepend-trail tl-trail add-cls*_{NOT} *remove-cls*_{NOT}
 $\lambda C \ C' \ L' \ S \ T. \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \text{backjump-l-cond } C \ C' \ L' \ S \ T$
propagate-conds forget-conds inv
for
mset-cls :: 'cls ⇒ 'v clause **and**
insert-cls :: 'v literal ⇒ 'cls ⇒ 'cls **and**
remove-lit :: 'v literal ⇒ 'cls ⇒ 'cls **and**
mset-clss:: 'clss ⇒ 'v clauses **and**
union-clss :: 'clss ⇒ 'clss ⇒ 'clss **and**
in-clss :: 'cls ⇒ 'clss ⇒ bool **and**

$insert_cls :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
 $remove_from_clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
 $trail :: 'st \Rightarrow ('v, unit, unit) \text{ marked-lits}$ **and**
 $raw_clauses :: 'st \Rightarrow 'clss$ **and**
 $prepend_trail :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow 'st$ **and**
 $tl_trail :: 'st \Rightarrow 'st$ **and**
 $add_cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
 $remove_cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
 $propagate_conds :: ('v, unit, unit) \text{ marked-lit} \Rightarrow 'st \Rightarrow bool$ **and**
 $inv :: 'st \Rightarrow bool$ **and**
 $forget_conds :: 'cls \Rightarrow 'st \Rightarrow bool$ **and**
 $backjump_l_cond :: 'v \text{ clause} \Rightarrow 'v \text{ clause} \Rightarrow 'v \text{ literal} \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$
 $+$
fixes $f :: nat \Rightarrow nat$
assumes
 $unbounded: unbounded\ f$ **and** $f_ge_1: \bigwedge n. n \geq 1 \implies f\ n \geq 1$ **and**
 $inv_restart: \bigwedge S\ T. inv\ S \implies T \sim reduce_trail_to_{NOT} \ \square\ S \implies inv\ T$
begin

definition $not_simplified_cls\ A = \{\#C \in \# A. \text{tautology } C \vee \neg distinct_mset\ C\ \#\}$

lemma $simple_clss_or_not_simplified_cls$:
assumes $atms_of_mm\ (clauses_{NOT}\ S) \subseteq atms_of_ms\ A$ **and**
 $x \in \# clauses_{NOT}\ S$ **and** $finite\ A$
shows $x \in simple_clss\ (atms_of_ms\ A) \vee x \in \# not_simplified_cls\ (clauses_{NOT}\ S)$
 $\langle proof \rangle$

lemma $cdcl_{NOT}\text{-merged-bj-learn-clauses-bound}$:
assumes
 $cdcl_{NOT}\text{-merged-bj-learn}\ S\ T$ **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 $set_mset\ (clauses_{NOT}\ T) \subseteq set_mset\ (not_simplified_cls\ (clauses_{NOT}\ S))$
 $\cup simple_clss\ (atms_of_ms\ A)$
 $\langle proof \rangle$

lemma $cdcl_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}$:
assumes $cdcl_{NOT}\text{-merged-bj-learn}\ S\ T$
shows $(not_simplified_cls\ (clauses_{NOT}\ T)) \subseteq \# (not_simplified_cls\ (clauses_{NOT}\ S))$
 $\langle proof \rangle$

lemma $rtranclp_cdcl_{NOT}\text{-merged-bj-learn-not-simplified-decreasing}$:
assumes $cdcl_{NOT}\text{-merged-bj-learn}^{**}\ S\ T$
shows $(not_simplified_cls\ (clauses_{NOT}\ T)) \subseteq \# (not_simplified_cls\ (clauses_{NOT}\ S))$
 $\langle proof \rangle$

lemma $rtranclp_cdcl_{NOT}\text{-merged-bj-learn-clauses-bound}$:
assumes
 $cdcl_{NOT}\text{-merged-bj-learn}^{**}\ S\ T$ **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 $\text{set-mset } (\text{clauses}_{NOT} T) \subseteq \text{set-mset } (\text{not-simplified-cls } (\text{clauses}_{NOT} S))$
 $\cup \text{simple-clss } (\text{atms-of-ms } A)$
 ⟨proof⟩

abbreviation $\mu_{CDCL}'\text{-bound}$ **where**
 $\mu_{CDCL}'\text{-bound } A \ T \equiv ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * 2$
 $+ \text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses}_{NOT} T)))$
 $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$

lemma $\text{rtranchp-cdcl}_{NOT}\text{-merged-bj-learn-clauses-bound-card}$:

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 $\mu_{CDCL}'\text{-merged } A \ T \leq \mu_{CDCL}'\text{-bound } A \ S$
 ⟨proof⟩

sublocale $\text{cdcl}_{NOT}\text{-increasing-restarts-ops } \lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}]) \ S$
cdcl_{NOT}-merged-bj-learn f
 $\lambda A \ S. \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$
 $\wedge \text{atm-of ' lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}'\text{-merged}$
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\mu_{CDCL}'\text{-bound}$
 ⟨proof⟩

lemma $\text{cdcl}_{NOT}\text{-restart-}\mu_{CDCL}'\text{-merged-le-}\mu_{CDCL}'\text{-bound}$:

assumes
cdcl_{NOT}-restart T V
inv (fst T) and
no-dup (trail (fst T)) and
atms-of-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 $\mu_{CDCL}'\text{-merged } A \ (\text{fst } V) \leq \mu_{CDCL}'\text{-bound } A \ (\text{fst } T)$
 ⟨proof⟩

lemma $\text{cdcl}_{NOT}\text{-restart-}\mu_{CDCL}'\text{-bound-le-}\mu_{CDCL}'\text{-bound}$:

assumes
cdcl_{NOT}-restart T V and
no-dup (trail (fst T)) and
inv (fst T) and
fin: finite A
shows $\mu_{CDCL}'\text{-bound } A \ (\text{fst } V) \leq \mu_{CDCL}'\text{-bound } A \ (\text{fst } T)$
 ⟨proof⟩

sublocale $\text{cdcl}_{NOT}\text{-increasing-restarts} \text{ - - - - - } f$

$\lambda S \ T. \ T \sim \text{reduce-trail-to}_{NOT} ([::'a \text{ list}]) \ S$
 $\lambda A \ S. \text{atms-of-mm } (\text{clauses}_{NOT} S) \subseteq \text{atms-of-ms } A$

$\wedge \text{atm-of } \text{'lits-of-l } (\text{trail } S) \subseteq \text{atms-of-ms } A \wedge \text{finite } A$
 $\mu_{CDCL}'\text{-merged } \text{cdcl}_{NOT}\text{-merged-bj-learn}$
 $\lambda S. \text{inv } S \wedge \text{no-dup } (\text{trail } S)$
 $\lambda A \ T. ((2 + \text{card } (\text{atms-of-ms } A)) \wedge (1 + \text{card } (\text{atms-of-ms } A))) * 2$
 $+ \text{card } (\text{set-mset } (\text{not-simplified-cls}(\text{clauses}_{NOT} \ T)))$
 $+ 3 \wedge \text{card } (\text{atms-of-ms } A)$
 $\langle \text{proof} \rangle$

lemma $\text{cdcl}_{NOT}\text{-restart-eq-sat-iff}$:

assumes

$\text{cdcl}_{NOT}\text{-restart } S \ T$ **and**

$\text{no-dup } (\text{trail } (\text{fst } S))$

$\text{inv } (\text{fst } S)$

shows $I \models_{\text{sextm}} \text{clauses}_{NOT} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} (\text{fst } T)$

$\langle \text{proof} \rangle$

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-restart-eq-sat-iff}$:

assumes

$\text{cdcl}_{NOT}\text{-restart}^{**} S \ T$ **and**

$\text{inv: inv } (\text{fst } S)$ **and** $n\text{-d: no-dup}(\text{trail } (\text{fst } S))$

shows $I \models_{\text{sextm}} \text{clauses}_{NOT} (\text{fst } S) \longleftrightarrow I \models_{\text{sextm}} \text{clauses}_{NOT} (\text{fst } T)$

$\langle \text{proof} \rangle$

lemma $\text{cdcl}_{NOT}\text{-restart-all-decomposition-implies-m}$:

assumes

$\text{cdcl}_{NOT}\text{-restart } S \ T$ **and**

$\text{inv: inv } (\text{fst } S)$ **and** $n\text{-d: no-dup}(\text{trail } (\text{fst } S))$ **and**

$\text{all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } S))$

$(\text{get-all-marked-decomposition } (\text{trail } (\text{fst } S)))$

shows $\text{all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } T))$

$(\text{get-all-marked-decomposition } (\text{trail } (\text{fst } T)))$

$\langle \text{proof} \rangle$

lemma $\text{rtrancpl-cdcl}_{NOT}\text{-restart-all-decomposition-implies-m}$:

assumes

$\text{cdcl}_{NOT}\text{-restart}^{**} S \ T$ **and**

$\text{inv: inv } (\text{fst } S)$ **and** $n\text{-d: no-dup}(\text{trail } (\text{fst } S))$ **and**

$\text{decomp: all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } S))$

$(\text{get-all-marked-decomposition } (\text{trail } (\text{fst } S)))$

shows $\text{all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } T))$

$(\text{get-all-marked-decomposition } (\text{trail } (\text{fst } T)))$

$\langle \text{proof} \rangle$

lemma $\text{full-cdcl}_{NOT}\text{-restart-normal-form}$:

assumes

$\text{full: full } \text{cdcl}_{NOT}\text{-restart } S \ T$ **and**

$\text{inv: inv } (\text{fst } S)$ **and** $n\text{-d: no-dup}(\text{trail } (\text{fst } S))$ **and**

$\text{decomp: all-decomposition-implies-m } (\text{clauses}_{NOT} (\text{fst } S))$

$(\text{get-all-marked-decomposition } (\text{trail } (\text{fst } S)))$ **and**

$\text{atms-cls: atms-of-mm } (\text{clauses}_{NOT} (\text{fst } S)) \subseteq \text{atms-of-ms } A$ **and**

$\text{atms-trail: atm-of } \text{'lits-of-l } (\text{trail } (\text{fst } S)) \subseteq \text{atms-of-ms } A$ **and**

$\text{fin: finite } A$

shows $\text{unsatisfiable } (\text{set-mset } (\text{clauses}_{NOT} (\text{fst } S)))$

$\vee \text{lits-of-l } (\text{trail } (\text{fst } T)) \models_{\text{sextm}} \text{clauses}_{NOT} (\text{fst } S) \wedge \text{satisfiable } (\text{set-mset } (\text{clauses}_{NOT} (\text{fst } S)))$

$\langle \text{proof} \rangle$

corollary *full-cdcl_{NOT}-restart-normal-form-init-state*:

assumes

init-state: $\text{trail } S = [] \text{ clauses}_{NOT} S = N$ **and**

full: *full cdcl_{NOT}-restart* ($S, 0$) T **and**

inv: *inv* S

shows *unsatisfiable* (*set-mset* N)

$\vee \text{ lits-of-l } (\text{trail } (\text{fst } T)) \models_{\text{sextm}} N \wedge \text{satisfiable } (\text{set-mset } N)$

$\langle \text{proof} \rangle$

end

end

theory *DPLL-NOT*

imports *CDCL-NOT*

begin

17 DPLL as an instance of NOT

17.1 DPLL with simple backtrack

We are using a concrete couple instead of an abstract state.

locale *dpll-with-backtrack*

begin

inductive *backtrack* :: $('v, \text{unit}, \text{unit}) \text{ marked-lit list} \times 'v \text{ clauses}$

$\Rightarrow ('v, \text{unit}, \text{unit}) \text{ marked-lit list} \times 'v \text{ clauses} \Rightarrow \text{bool}$ **where**

backtrack-split ($\text{fst } S$) = $(M', L \# M) \Longrightarrow \text{is-marked } L \Longrightarrow D \in \# \text{ snd } S$

$\Longrightarrow \text{fst } S \models_{\text{as}} \text{CNot } D \Longrightarrow \text{backtrack } S \text{ (Propagated } (- (\text{lit-of } L)) () \# M, \text{snd } S)$

inductive-cases *backtrackE*[*elim*]: *backtrack* (M, N) (M', N')

lemma *backtrack-is-backjump*:

fixes $M M' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list}$

assumes

backtrack: *backtrack* (M, N) (M', N') **and**

no-dup: $(\text{no-dup} \circ \text{fst}) (M, N)$ **and**

decomp: *all-decomposition-implies-m* N (*get-all-marked-decomposition* M)

shows

$\exists C F' K F L l C'.$

$M = F' @ \text{Marked } K () \# F \wedge$

$M' = \text{Propagated } L l \# F \wedge N = N' \wedge C \in \# N \wedge F' @ \text{Marked } K d \# F \models_{\text{as}} \text{CNot } C \wedge$

undefined-lit $F L \wedge \text{atm-of } L \in \text{atms-of-mm } N \cup \text{atm-of } ' \text{ lits-of-l } (F' @ \text{Marked } K d \# F) \wedge$

$N \models_{\text{pm}} C' + \{\#L\} \wedge F \models_{\text{as}} \text{CNot } C'$

$\langle \text{proof} \rangle$

lemma *backtrack-is-backjump'*:

fixes $M M' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list}$

assumes

backtrack: *backtrack* $S T$ **and**

no-dup: $(\text{no-dup} \circ \text{fst}) S$ **and**

decomp: *all-decomposition-implies-m* (*snd* S) (*get-all-marked-decomposition* (*fst* S))

shows

$\exists C F' K F L l C'.$

$\text{fst } S = F' @ \text{Marked } K () \# F \wedge$

$T = (\text{Propagated } L l \# F, \text{snd } S) \wedge C \in \# \text{snd } S \wedge \text{fst } S \models_{\text{as}} \text{CNot } C$

$\wedge \text{undefined-lit } F L \wedge \text{atm-of } L \in \text{atms-of-mm } (\text{snd } S) \cup \text{atm-of } ' \text{ lits-of-l } (\text{fst } S) \wedge$

$\text{snd } S \models_{pm} C' + \{\#L\# \} \wedge F \models_{as} CNot \ C'$
 $\langle proof \rangle$

sublocale *dpll-state*

$id \ \lambda L \ C. \ C + \{\#L\# \} \ \text{remove1-mset}$
 $id \ op + op \in \# \ \lambda L \ C. \ C + \{\#L\# \} \ \text{remove1-mset}$
 $\text{fst } \text{snd } \lambda L \ (M, N). \ (L \# \ M, N) \ \lambda(M, N). \ (tl \ M, N)$
 $\lambda C \ (M, N). \ (M, \{\#C\# \} + N) \ \lambda C \ (M, N). \ (M, \text{removeAll-mset } C \ N)$
 $\langle proof \rangle$

sublocale *backjumping-ops*

$id \ \lambda L \ C. \ C + \{\#L\# \} \ \text{remove1-mset}$
 $id \ op + op \in \# \ \lambda L \ C. \ C + \{\#L\# \} \ \text{remove1-mset}$
 $\text{fst } \text{snd } \lambda L \ (M, N). \ (L \# \ M, N) \ \lambda(M, N). \ (tl \ M, N)$
 $\lambda C \ (M, N). \ (M, \{\#C\# \} + N) \ \lambda C \ (M, N). \ (M, \text{removeAll-mset } C \ N) \ \lambda - - S \ T. \ \text{backtrack } S \ T$
 $\langle proof \rangle$

lemma *reduce-trail-to_{NOT}-snd:*

$\text{snd } (\text{reduce-trail-to}_{NOT} \ F \ S) = \text{snd } S$
 $\langle proof \rangle$

lemma *reduce-trail-to_{NOT}:*

$\text{reduce-trail-to}_{NOT} \ F \ S =$
 $(\text{if } \text{length } (\text{fst } S) \geq \text{length } F$
 $\text{then } \text{drop } (\text{length } (\text{fst } S) - \text{length } F) \ (\text{fst } S)$
 $\text{else } [],$
 $\text{snd } S) \ (\text{is } ?R = ?C)$
 $\langle proof \rangle$

lemma *backtrack-is-backjump'':*

fixes $M \ M' :: ('v, \text{unit}, \text{unit}) \text{ marked-lit list}$
assumes
 $\text{backtrack: } \text{backtrack } S \ T \ \text{and}$
 $\text{no-dup: } (\text{no-dup} \circ \text{fst}) \ S \ \text{and}$
 $\text{decomp: } \text{all-decomposition-implies-m } (\text{snd } S) \ (\text{get-all-marked-decomposition } (\text{fst } S))$
shows $\text{backjump } S \ T$
 $\langle proof \rangle$

lemma *can-do-bt-step:*

assumes
 $M: \text{fst } S = F' \ @ \ \text{Marked } K \ d \ \# \ F \ \text{and}$
 $C \in \# \ \text{snd } S \ \text{and}$
 $C: \text{fst } S \models_{as} CNot \ C$
shows $\neg \text{no-step } \text{backtrack } S$
 $\langle proof \rangle$

end

sublocale *dpll-with-backtrack* \subseteq *dpll-with-backjumping-ops*

$id \ \lambda L \ C. \ C + \{\#L\# \} \ \text{remove1-mset}$
 $id \ op + op \in \# \ \lambda L \ C. \ C + \{\#L\# \} \ \text{remove1-mset}$
 $\text{fst } \text{snd } \lambda L \ (M, N). \ (L \# \ M, N)$
 $\lambda(M, N). \ (tl \ M, N) \ \lambda C \ (M, N). \ (M, \{\#C\# \} + N) \ \lambda C \ (M, N). \ (M, \text{removeAll-mset } C \ N)$
 $\lambda(M, N). \ \text{no-dup } M \wedge \text{all-decomposition-implies-m } N \ (\text{get-all-marked-decomposition } M)$
 $\lambda - - S \ T. \ \text{backtrack } S \ T$

λ - -. *True*
 \langle *proof* \rangle

sublocale *dpll-with-backtrack* \subseteq *dpll-with-backjumping*

id λL *C*. *C* + $\{\#L\#\}$ *remove1-mset*
id *op* + *op* $\in \#$ λL *C*. *C* + $\{\#L\#\}$ *remove1-mset*
fst snd λL (*M*, *N*). (*L* $\#$ *M*, *N*)
 $\lambda(M, N).$ (*tl* *M*, *N*) λC (*M*, *N*). (*M*, $\{\#C\#\}$ + *N*) λC (*M*, *N*). (*M*, *removeAll-mset C N*)
 $\lambda(M, N).$ *no-dup M* \wedge *all-decomposition-implies-m N* (*get-all-marked-decomposition M*)
 λ - - - *S T. backtrack S T*
 λ - -. *True*
 \langle *proof* \rangle

context *dpll-with-backtrack*

begin

term *learn*

end

context *dpll-with-backtrack*

begin

lemma *wf-tranclp-dpll-inital-state*:

assumes *fin*: *finite A*
shows *wf* $\{((M'::('v, unit, unit) \text{ marked-lits}, N'::'v \text{ clauses}), ([], N)) \mid M' N' N.$
 $\text{dpll-bj}^{++} ([], N) (M', N') \wedge \text{atms-of-mm } N \subseteq \text{atms-of-ms } A\}$
 \langle *proof* \rangle

corollary *full-dpll-final-state-conclusive*:

fixes *M M' :: ('v, unit, unit) marked-lit list*
assumes
full: *full dpll-bj* ($[], N$) (*M'*, *N'*)
shows *unsatisfiable (set-mset N)* \vee (*M'* \models_{asm} *N* \wedge *satisfiable (set-mset N)*)
 \langle *proof* \rangle

corollary *full-dpll-normal-form-from-init-state*:

fixes *M M' :: ('v, unit, unit) marked-lit list*
assumes
full: *full dpll-bj* ($[], N$) (*M'*, *N'*)
shows *M' \models_{asm} N \longleftrightarrow satisfiable (set-mset N)*
 \langle *proof* \rangle

interpretation *conflict-driven-clause-learning-ops*

id λL *C*. *C* + $\{\#L\#\}$ *remove1-mset*
id *op* + *op* $\in \#$ λL *C*. *C* + $\{\#L\#\}$ *remove1-mset*
fst snd λL (*M*, *N*). (*L* $\#$ *M*, *N*)
 $\lambda(M, N).$ (*tl* *M*, *N*) λC (*M*, *N*). (*M*, $\{\#C\#\}$ + *N*) λC (*M*, *N*). (*M*, *removeAll-mset C N*)
 $\lambda(M, N).$ *no-dup M* \wedge *all-decomposition-implies-m N* (*get-all-marked-decomposition M*)
 λ - - - *S T. backtrack S T*
 λ - -. *True* λ - -. *False* λ - -. *False*
 \langle *proof* \rangle

interpretation *conflict-driven-clause-learning*

id λL *C*. *C* + $\{\#L\#\}$ *remove1-mset*
id *op* + *op* $\in \#$ λL *C*. *C* + $\{\#L\#\}$ *remove1-mset*
fst snd λL (*M*, *N*). (*L* $\#$ *M*, *N*)

$\lambda(M, N). (tl\ M, N) \lambda C\ (M, N). (M, \{\#C\# \} + N) \lambda C\ (M, N). (M, removeAll-mset\ C\ N)$
 $\lambda(M, N). no-dup\ M \wedge all-decomposition-implies-m\ N\ (get-all-marked-decomposition\ M)$
 $\lambda- - S\ T. backtrack\ S\ T$
 $\lambda- -. True\ \lambda- -. False\ \lambda- -. False$
 $\langle proof \rangle$

lemma $cdcl_{NOT-is-dpll}$:
 $cdcl_{NOT} S\ T \longleftrightarrow dpll-bj\ S\ T$
 $\langle proof \rangle$

Another proof of termination:

lemma $wf\ \{(T, S). dpll-bj\ S\ T \wedge cdcl_{NOT-ALL-INV}\ A\ S\}$
 $\langle proof \rangle$
end

17.2 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 $unbounded: unbounded\ f\ \text{and}\ f-ge-1: \bigwedge n. n \geq 1 \implies f\ n \geq 1$
begin
sublocale $cdcl_{NOT-increasing-restarts}$
 $id\ \lambda L\ C. C + \{\#L\# \} remove1-mset$
 $id\ op + op \in \# \lambda L\ C. C + \{\#L\# \} remove1-mset$
 $fst\ snd\ \lambda L\ (M, N). (L \# M, N) \lambda(M, N). (tl\ M, N)$
 $\lambda C\ (M, N). (M, \{\#C\# \} + N) \lambda C\ (M, N). (M, removeAll-mset\ C\ N) f\ \lambda(-, N)\ S. S = ([], N)$
 $\lambda A\ (M, N). atms-of-mm\ N \subseteq atms-of-ms\ A \wedge atm-of\ 'lits-of-l\ M \subseteq atms-of-ms\ A \wedge finite\ A$
 $\wedge all-decomposition-implies-m\ N\ (get-all-marked-decomposition\ M)$
 $\lambda A\ T. (2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $\quad - \mu_C\ (1 + card\ (atms-of-ms\ A))\ (2 + card\ (atms-of-ms\ A))\ (trail-weight\ T)\ dpll-bj$
 $\lambda(M, N). no-dup\ M \wedge all-decomposition-implies-m\ N\ (get-all-marked-decomposition\ M)$
 $\lambda A\ -. (2 + card\ (atms-of-ms\ A)) \wedge (1 + card\ (atms-of-ms\ A))$
 $\langle proof \rangle$
end

end
theory $DPLL-W$
imports $Main\ Partial-Clausal-Logic\ Partial-Annotated-Clausal-Logic\ List-More\ Wellfounded-More$
 $DPLL-NOT$
begin

18 DPLL

18.1 Rules

type-synonym $'a\ dpll_W\text{-marked-lit} = ('a, unit, unit)\ marked-lit$
type-synonym $'a\ dpll_W\text{-marked-lits} = ('a, unit, unit)\ marked-lits$
type-synonym $'v\ dpll_W\text{-state} = 'v\ dpll_W\text{-marked-lits} \times 'v\ clauses$

abbreviation $trail :: 'v\ dpll_W\text{-state} \Rightarrow 'v\ dpll_W\text{-marked-lits}\ \text{where}$
 $trail \equiv fst$

abbreviation $clauses :: 'v\ dpll_W\text{-state} \Rightarrow 'v\ clauses\ \text{where}$

$clauses \equiv snd$

The definition of DPLL is given in figure 2.13 page 70 of CW.

inductive $dpll_W :: 'v \text{ dpll}_W\text{-state} \Rightarrow 'v \text{ dpll}_W\text{-state} \Rightarrow \text{bool}$ **where**
propagate: $C + \{\#L\# \} \in \# \text{ clauses } S \Longrightarrow \text{trail } S \models_{as} CNot \ C \Longrightarrow \text{undefined-lit } (\text{trail } S) \ L$
 $\Longrightarrow dpll_W \ S \ (\text{Propagated } L \ ()) \ \# \ \text{trail } S, \text{ clauses } S) \mid$
decided: $\text{undefined-lit } (\text{trail } S) \ L \Longrightarrow \text{atm-of } L \in \text{atms-of-mm } (\text{clauses } S)$
 $\Longrightarrow dpll_W \ S \ (\text{Marked } L \ ()) \ \# \ \text{trail } S, \text{ clauses } S) \mid$
backtrack: $\text{backtrack-split } (\text{trail } S) = (M', L \# M) \Longrightarrow \text{is-marked } L \Longrightarrow D \in \# \text{ clauses } S$
 $\Longrightarrow \text{trail } S \models_{as} CNot \ D \Longrightarrow dpll_W \ S \ (\text{Propagated } (- \ (\text{lit-of } L)) \ ()) \ \# \ M, \text{ clauses } S)$

18.2 Invariants

lemma $dpll_W\text{-distinct-inv}$:

assumes $dpll_W \ S \ S'$
and $\text{no-dup } (\text{trail } S)$
shows $\text{no-dup } (\text{trail } S')$
 $\langle \text{proof} \rangle$

lemma $dpll_W\text{-consistent-interp-inv}$:

assumes $dpll_W \ S \ S'$
and $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$
and $\text{no-dup } (\text{trail } S)$
shows $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S'))$
 $\langle \text{proof} \rangle$

lemma $dpll_W\text{-vars-in-snd-inv}$:

assumes $dpll_W \ S \ S'$
and $\text{atm-of } ' \ (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{clauses } S)$
shows $\text{atm-of } ' \ (\text{lits-of-l } (\text{trail } S')) \subseteq \text{atms-of-mm } (\text{clauses } S')$
 $\langle \text{proof} \rangle$

lemma $\text{atms-of-ms-lit-of-atms-of}$: $\text{atms-of-ms } ((\lambda a. \{\# \text{lit-of } a\# \}) \ ' \ c) = \text{atm-of } ' \ \text{lit-of } ' \ c$
 $\langle \text{proof} \rangle$

Lemma theorem 2.8.2 page 71 of CW

lemma $dpll_W\text{-propagate-is-conclusion}$:

assumes $dpll_W \ S \ S'$
and $\text{all-decomposition-implies-m } (\text{clauses } S) \ (\text{get-all-marked-decomposition } (\text{trail } S))$
and $\text{atm-of } ' \ \text{lits-of-l } (\text{trail } S) \subseteq \text{atms-of-mm } (\text{clauses } S)$
shows $\text{all-decomposition-implies-m } (\text{clauses } S') \ (\text{get-all-marked-decomposition } (\text{trail } S'))$
 $\langle \text{proof} \rangle$

Lemma theorem 2.8.3 page 72 of CW

theorem $dpll_W\text{-propagate-is-conclusion-of-decided}$:

assumes $dpll_W \ S \ S'$
and $\text{all-decomposition-implies-m } (\text{clauses } S) \ (\text{get-all-marked-decomposition } (\text{trail } S))$
and $\text{atm-of } ' \ \text{lits-of-l } (\text{trail } S) \subseteq \text{atms-of-mm } (\text{clauses } S)$
shows $\text{set-mset } (\text{clauses } S') \cup \{\{\# \text{lit-of } L\# \} \mid L. \text{is-marked } L \wedge L \in \text{set } (\text{trail } S')\}$
 $\models_{ps} (\lambda a. \{\# \text{lit-of } a\# \}) \ ' \ \bigcup (\text{set } ' \ \text{snd } ' \ \text{set } (\text{get-all-marked-decomposition } (\text{trail } S')))$
 $\langle \text{proof} \rangle$

Lemma theorem 2.8.4 page 72 of CW

lemma $\text{only-propagated-vars-unsat}$:

assumes $\text{marked}: \forall x \in \text{set } M. \neg \text{is-marked } x$

and $DN: D \in N$ **and** $D: M \models_{as} CNot\ D$
and $inv: all-decomposition-implies\ N$ (*get-all-marked-decomposition* M)
and $atm-incl: atm-of\ ' lits-of-l\ M \subseteq atms-of-ms\ N$
shows *unsatisfiable* N
 $\langle proof \rangle$

lemma $dpll_W$ -same-clauses:
assumes $dpll_W\ S\ S'$
shows *clauses* $S = clauses\ S'$
 $\langle proof \rangle$

lemma $rtrancpl$ - $dpll_W$ -inv:
assumes $rtrancpl\ dpll_W\ S\ S'$
and $inv: all-decomposition-implies-m\ (clauses\ S)$ (*get-all-marked-decomposition* (*trail* S))
and $atm-incl: atm-of\ ' lits-of-l\ (trail\ S) \subseteq atms-of-mm\ (clauses\ S)$
and *consistent-interp* (*lits-of-l* (*trail* S))
and *no-dup* (*trail* S)
shows *all-decomposition-implies-m* (*clauses* S') (*get-all-marked-decomposition* (*trail* S'))
and $atm-of\ ' lits-of-l\ (trail\ S') \subseteq atms-of-mm\ (clauses\ S')$
and *clauses* $S = clauses\ S'$
and *consistent-interp* (*lits-of-l* (*trail* S'))
and *no-dup* (*trail* S')
 $\langle proof \rangle$

definition $dpll_W$ -all-inv $S \equiv$
(all-decomposition-implies-m (*clauses* S) (*get-all-marked-decomposition* (*trail* S))
 $\wedge atm-of\ ' lits-of-l\ (trail\ S) \subseteq atms-of-mm\ (clauses\ S)$
 $\wedge consistent-interp\ (lits-of-l\ (trail\ S))$
 $\wedge no-dup\ (trail\ S)$)

lemma $dpll_W$ -all-inv-dest[$dest$]:
assumes $dpll_W$ -all-inv S
shows *all-decomposition-implies-m* (*clauses* S) (*get-all-marked-decomposition* (*trail* S))
and $atm-of\ ' lits-of-l\ (trail\ S) \subseteq atms-of-mm\ (clauses\ S)$
and *consistent-interp* (*lits-of-l* (*trail* S)) $\wedge no-dup\ (trail\ S)$
 $\langle proof \rangle$

lemma $rtrancpl$ - $dpll_W$ -all-inv:
assumes $rtrancpl\ dpll_W\ S\ S'$
and $dpll_W$ -all-inv S
shows $dpll_W$ -all-inv S'
 $\langle proof \rangle$

lemma $dpll_W$ -all-inv:
assumes $dpll_W\ S\ S'$
and $dpll_W$ -all-inv S
shows $dpll_W$ -all-inv S'
 $\langle proof \rangle$

lemma $rtrancpl$ - $dpll_W$ -inv-starting-from-0:
assumes $rtrancpl\ dpll_W\ S\ S'$
and $inv: trail\ S = []$
shows $dpll_W$ -all-inv S'
 $\langle proof \rangle$

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 *rtranclp* *dpll_W* ([], *N*) (map (λM . *Marked* *M* ()) *M*, *N*)

<proof>

definition *conclusive-dpll_W-state* (*S*:: 'v *dpll_W-state*) \longleftrightarrow

(*trail* *S* \models_{asm} *clauses* *S* \vee ($\forall L \in \text{set } (\text{trail } S). \neg \text{is-marked } L$)

$\wedge (\exists C \in \# \text{ clauses } S. \text{trail } S \models_{as} C \text{Not } C))$

lemma *dpll_W-strong-completeness*:

assumes *set* *M* \models_{sm} *N*

and *consistent-interp* (set *M*)

and *distinct* *M*

and *atm-of* ' (set *M*) \subseteq *atms-of-mm* *N*

shows *dpll_W*** ([], *N*) (map (λM . *Marked* *M* ()) *M*, *N*)

and *conclusive-dpll_W-state* (map (λM . *Marked* *M* ()) *M*, *N*)

<proof>

lemma *dpll_W-sound*:

assumes

rtranclp *dpll_W* ([], *N*) (*M*, *N*) **and**

$\forall S. \neg \text{dpll}_W (M, N) S$

shows *M* \models_{asm} *N* \longleftrightarrow *satisfiable* (set-mset *N*) (**is** ?*A* \longleftrightarrow ?*B*)

<proof>

18.3 Termination

definition *dpll_W-mes* *M* *n* =

map (λl . if *is-marked* *l* then 2 else (1::nat)) (rev *M*) @ replicate (*n* - length *M*) 3

lemma *length-dpll_W-mes*:

assumes length *M* \leq *n*

shows length (*dpll_W-mes* *M* *n*) = *n*

<proof>

lemma *distinctcard-atm-of-lit-of-eq-length*:

assumes *no-dup* *S*

shows card (*atm-of* ' *lits-of-l* *S*) = length *S*

<proof>

lemma *dpll_W-card-decrease*:

assumes *dpll*: *dpll_W* *S* *S'* **and** length (trail *S'*) \leq card vars

and length (trail *S*) \leq card vars

shows (*dpll_W-mes* (trail *S'*) (card vars), *dpll_W-mes* (trail *S*) (card vars))

$\in \text{lexn } \{(a, b). a < b\}$ (card vars)

<proof>

Proposition theorem 2.8.7 page 73 of CW

lemma *dpll_W-card-decrease'*:

assumes *dpll*: *dpll_W* *S* *S'*

and *atm-incl*: *atm-of* ' *lits-of-l* (trail *S*) \subseteq *atms-of-mm* (clauses *S*)

and *no-dup*: *no-dup* (trail *S*)

shows $(dpll_W\text{-mes } (trail\ S')\ (card\ (atms\text{-of-mm } (clauses\ S'))),$
 $dpll_W\text{-mes } (trail\ S)\ (card\ (atms\text{-of-mm } (clauses\ S)))) \in lex\ \{(a, b). a < b\}$
 $\langle proof \rangle$

lemma $wf\text{-lexn}$: $wf\ (lexn\ \{(a, b). (a::nat) < b\}\ (card\ (atms\text{-of-mm } (clauses\ S))))$
 $\langle proof \rangle$

lemma $dpll_W\text{-wf}$:
 $wf\ \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W\ S\ S'\}$
 $\langle proof \rangle$

lemma $dpll_W\text{-trancpl-star-commute}$:
 $\{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W\ S\ S'\}^+ = \{(S', S). dpll_W\text{-all-inv } S \wedge trancpl\ dpll_W\ S\ S'\}$
 $(is\ ?A = ?B)$
 $\langle proof \rangle$

lemma $dpll_W\text{-wf-trancpl}$: $wf\ \{(S', S). dpll_W\text{-all-inv } S \wedge dpll_W^{++}\ S\ S'\}$
 $\langle proof \rangle$

lemma $dpll_W\text{-wf-plus}$:
shows $wf\ \{(S', ([], N)) \mid S'. dpll_W^{++}\ ([], N)\ S'\}\ (is\ wf\ ?P)$
 $\langle proof \rangle$

18.4 Final States

lemma $dpll_W\text{-no-more-step-is-a-conclusive-state}$:
assumes $\forall S'. \neg dpll_W\ S\ S'$
shows $conclusive\text{-}dpll_W\text{-state } S$
 $\langle proof \rangle$

lemma $dpll_W\text{-conclusive-state-correct}$:
assumes $dpll_W^{**}\ ([], N)\ (M, N)$ **and** $conclusive\text{-}dpll_W\text{-state } (M, N)$
shows $M \models_{asm} N \longleftrightarrow satisfiable\ (set\text{-mset } N)\ (is\ ?A \longleftrightarrow ?B)$
 $\langle proof \rangle$

18.5 Link with NOT's DPLL

interpretation $dpll_W\text{-NOT}$: $dpll\text{-with-backtrack } \langle proof \rangle$

declare $dpll_W\text{-NOT.state-simp}_{NOT}[simp\ del]$
lemma $state\text{-eq}_{NOT}\text{-iff-eq}[iff, simp]$: $dpll_W\text{-NOT.state-eq}_{NOT}\ S\ T \longleftrightarrow S = T$
 $\langle proof \rangle$

lemma $dpll_W\text{-dpll_W-bj}$:
assumes inv : $dpll_W\text{-all-inv } S$ **and** $dpll$: $dpll_W\ S\ T$
shows $dpll_W\text{-NOT.dpll-bj } S\ T$
 $\langle proof \rangle$

lemma $dpll_W\text{-bj-dpll}$:
assumes inv : $dpll_W\text{-all-inv } S$ **and** $dpll$: $dpll_W\text{-NOT.dpll-bj } S\ T$
shows $dpll_W\ S\ T$
 $\langle proof \rangle$

lemma $rtrancpl\text{-}dpll_W\text{-rtrancpl}\text{-}dpll_W\text{-NOT}$:
assumes $dpll_W^{**}\ S\ T$ **and** $dpll_W\text{-all-inv } S$
shows $dpll_W\text{-NOT.dpll-bj}^{**}\ S\ T$

$\langle \text{proof} \rangle$

lemma *rtrancp-dpll-rtrancp-dpll_W*:
assumes *dpll_W-NOT.dpll-bj** S T* **and** *dpll_W-all-inv S*
shows *dpll_W** S T*
 $\langle \text{proof} \rangle$

lemma *dpll-conclusive-state-correctness*:
assumes *dpll_W-NOT.dpll-bj** ([], N) (M, N)* **and** *conclusive-dpll_W-state (M, N)*
shows $M \models_{asm} N \longleftrightarrow \text{satisfiable } (\text{set-mset } N)$
 $\langle \text{proof} \rangle$

end
theory *CDCL-W-Level*
imports *Partial-Annotated-Clausal-Logic*
begin

18.5.1 Level of literals and clauses

Getting the level of a variable, implies that the list has to be reversed. Here is the function after reversing.

fun *get-rev-level* :: ('v, nat, 'a) *marked-lits* \Rightarrow nat \Rightarrow 'v *literal* \Rightarrow nat **where**
get-rev-level [] - = 0 |
get-rev-level (Marked l level # Ls) n L =
 (if *atm-of* l = *atm-of* L then level else *get-rev-level* Ls level L) |
get-rev-level (Propagated l - # Ls) n L =
 (if *atm-of* l = *atm-of* L then n else *get-rev-level* Ls n L)

abbreviation *get-level* M L \equiv *get-rev-level* (rev M) 0 L

lemma *get-rev-level-uminus[simp]*: *get-rev-level* M n (−L) = *get-rev-level* M n L
 $\langle \text{proof} \rangle$

lemma *atm-of-notin-get-rev-level-eq-0*:
assumes *atm-of* L \notin *atm-of* ' lits-of-l M
shows *get-rev-level* M n L = 0
 $\langle \text{proof} \rangle$

lemma *get-rev-level-ge-0-atm-of-in*:
assumes *get-rev-level* M n L > n
shows *atm-of* L \in *atm-of* ' lits-of-l M
 $\langle \text{proof} \rangle$

In *get-rev-level* (resp. *get-level*), the beginning (resp. the end) can be skipped if the literal is not in the beginning (resp. the end).

lemma *get-rev-level-skip[simp]*:
assumes *atm-of* L \notin *atm-of* ' lits-of-l M
shows *get-rev-level* (M @ Marked K i # M') n L = *get-rev-level* (Marked K i # M') i L
 $\langle \text{proof} \rangle$

lemma *get-rev-level-notin-end[simp]*:
assumes *atm-of* L \notin *atm-of* ' lits-of-l M'
shows *get-rev-level* (M @ M') n L = *get-rev-level* M n L
 $\langle \text{proof} \rangle$

If the literal is at the beginning, then the end can be skipped

lemma *get-rev-level-skip-end[simp]*:
assumes $\text{atm-of } L \in \text{atm-of ' lits-of-l } M$
shows $\text{get-rev-level } (M @ M') \ n \ L = \text{get-rev-level } M \ n \ L$
 $\langle \text{proof} \rangle$

lemma *get-level-skip-beginning*:
assumes $\text{atm-of } L' \neq \text{atm-of } (\text{lit-of } K)$
shows $\text{get-level } (K \# M) \ L' = \text{get-level } M \ L'$
 $\langle \text{proof} \rangle$

lemma *get-level-skip-beginning-not-marked-rev*:
assumes $\text{atm-of } L \notin \text{atm-of ' lit-of ' (set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows $\text{get-level } (M @ \text{rev } S) \ L = \text{get-level } M \ L$
 $\langle \text{proof} \rangle$

lemma *get-level-skip-beginning-not-marked[simp]*:
assumes $\text{atm-of } L \notin \text{atm-of ' lit-of ' (set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows $\text{get-level } (M @ S) \ L = \text{get-level } M \ L$
 $\langle \text{proof} \rangle$

lemma *get-rev-level-skip-beginning-not-marked[simp]*:
assumes $\text{atm-of } L \notin \text{atm-of ' lit-of ' (set } S)$
and $\forall s \in \text{set } S. \neg \text{is-marked } s$
shows $\text{get-rev-level } (\text{rev } S @ \text{rev } M) \ 0 \ L = \text{get-level } M \ L$
 $\langle \text{proof} \rangle$

lemma *get-level-skip-in-all-not-marked*:
fixes $M :: ('a, \text{nat}, 'b) \text{ marked-lit list}$ **and** $L :: 'a \text{ literal}$
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
and $\text{atm-of } L \in \text{atm-of ' lit-of ' (set } M)$
shows $\text{get-rev-level } M \ n \ L = n$
 $\langle \text{proof} \rangle$

lemma *get-level-skip-all-not-marked[simp]*:
fixes M
defines $M' \equiv \text{rev } M$
assumes $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\text{get-level } M \ L = 0$
 $\langle \text{proof} \rangle$

abbreviation $M\text{Max } M \equiv \text{Max } (\text{set-mset } M)$

the $\{\#0 :: 'a\# \}$ is there to ensures that the set is not empty.

definition *get-maximum-level* :: $('a, \text{nat}, 'b) \text{ marked-lit list} \Rightarrow 'a \text{ literal multiset} \Rightarrow \text{nat}$
where
 $\text{get-maximum-level } M \ D = M\text{Max } (\{\#0\# \} + \text{image-mset } (\text{get-level } M) \ D)$

lemma *get-maximum-level-ge-get-level*:
 $L \in \# \ D \Longrightarrow \text{get-maximum-level } M \ D \geq \text{get-level } M \ L$
 $\langle \text{proof} \rangle$

lemma *get-maximum-level-empty[simp]*:

get-maximum-level $M \ \{\#\} = 0$
 ⟨proof⟩

lemma *get-maximum-level-exists-lit-of-max-level*:
 $D \neq \{\#\} \implies \exists L \in \#D. \text{get-level } M \ L = \text{get-maximum-level } M \ D$
 ⟨proof⟩

lemma *get-maximum-level-empty-list[simp]*:
 $\text{get-maximum-level } [] \ D = 0$
 ⟨proof⟩

lemma *get-maximum-level-single[simp]*:
 $\text{get-maximum-level } M \ \{\#L\# \} = \text{get-level } M \ L$
 ⟨proof⟩

lemma *get-maximum-level-plus*:
 $\text{get-maximum-level } M \ (D + D') = \max (\text{get-maximum-level } M \ D) (\text{get-maximum-level } M \ D')$
 ⟨proof⟩

lemma *get-maximum-level-exists-lit*:
assumes $n: n > 0$
and $\text{max}: \text{get-maximum-level } M \ D = n$
shows $\exists L \in \#D. \text{get-level } M \ L = n$
 ⟨proof⟩

lemma *get-maximum-level-skip-first[simp]*:
assumes $\text{atm-of } L \notin \text{atms-of } D$
shows $\text{get-maximum-level } (\text{Propagated } L \ C \ \# \ M) \ D = \text{get-maximum-level } M \ D$
 ⟨proof⟩

lemma *get-maximum-level-skip-beginning*:
assumes $DH: \text{atms-of } D \subseteq \text{atm-of } \text{'lits-of-l } H$
shows $\text{get-maximum-level } (c \ @ \ \text{Marked } Kh \ i \ \# \ H) \ D = \text{get-maximum-level } H \ D$
 ⟨proof⟩

lemma *get-maximum-level-D-single-propagated*:
 $\text{get-maximum-level } [\text{Propagated } x21 \ x22] \ D = 0$
 ⟨proof⟩

lemma *get-maximum-level-skip-notin*:
assumes $D: \forall L \in \#D. \text{atm-of } L \in \text{atm-of } \text{'lits-of-l } M$
shows $\text{get-maximum-level } M \ D = \text{get-maximum-level } (\text{Propagated } x21 \ x22 \ \# \ M) \ D$
 ⟨proof⟩

lemma *get-maximum-level-skip-un-marked-not-present*:
assumes $\forall L \in \#D. \text{atm-of } L \in \text{atm-of } \text{'lits-of-l } aa$ **and**
 $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\text{get-maximum-level } aa \ D = \text{get-maximum-level } (M \ @ \ aa) \ D$
 ⟨proof⟩

lemma *get-maximum-level-union-mset*:
 $\text{get-maximum-level } M \ (A \ \#\cup \ B) = \text{get-maximum-level } M \ (A + B)$
 ⟨proof⟩

fun *get-maximum-possible-level*:: ('b, nat, 'c) marked-lit list \Rightarrow nat **where**
get-maximum-possible-level [] = 0 |
get-maximum-possible-level (Marked *K i* # *l*) = max *i* (*get-maximum-possible-level* *l*) |
get-maximum-possible-level (Propagated - - # *l*) = *get-maximum-possible-level* *l*

lemma *get-maximum-possible-level-append*[simp]:
get-maximum-possible-level (*M* @ *M'*)
= max (*get-maximum-possible-level* *M*) (*get-maximum-possible-level* *M'*)
⟨proof⟩

lemma *get-maximum-possible-level-rev*[simp]:
get-maximum-possible-level (rev *M*) = *get-maximum-possible-level* *M*
⟨proof⟩

lemma *get-maximum-possible-level-ge-get-rev-level*:
max (*get-maximum-possible-level* *M*) *i* \geq *get-rev-level* *M i L*
⟨proof⟩

lemma *get-maximum-possible-level-ge-get-level*[simp]:
get-maximum-possible-level *M* \geq *get-level* *M L*
⟨proof⟩

lemma *get-maximum-possible-level-ge-get-maximum-level*[simp]:
get-maximum-possible-level *M* \geq *get-maximum-level* *M D*
⟨proof⟩

fun *get-all-mark-of-propagated* **where**
get-all-mark-of-propagated [] = [] |
get-all-mark-of-propagated (Marked - - # *L*) = *get-all-mark-of-propagated* *L* |
get-all-mark-of-propagated (Propagated - mark # *L*) = mark # *get-all-mark-of-propagated* *L*

lemma *get-all-mark-of-propagated-append*[simp]:
get-all-mark-of-propagated (*A* @ *B*) = *get-all-mark-of-propagated* *A* @ *get-all-mark-of-propagated* *B*
⟨proof⟩

18.5.2 Properties about the levels

fun *get-all-levels-of-marked* :: ('b, 'a, 'c) marked-lit list \Rightarrow 'a list **where**
get-all-levels-of-marked [] = [] |
get-all-levels-of-marked (Marked *l level* # *Ls*) = *level* # *get-all-levels-of-marked* *Ls* |
get-all-levels-of-marked (Propagated - - # *Ls*) = *get-all-levels-of-marked* *Ls*

lemma *get-all-levels-of-marked-nil-iff-not-is-marked*:
get-all-levels-of-marked *xs* = [] \longleftrightarrow ($\forall x \in \text{set } xs. \neg \text{is-marked } x$)
⟨proof⟩

lemma *get-all-levels-of-marked-cons*:
get-all-levels-of-marked (*a* # *b*) =
(if *is-marked* *a* then [level-of *a*] else []) @ *get-all-levels-of-marked* *b*
⟨proof⟩

lemma *get-all-levels-of-marked-append*[simp]:
get-all-levels-of-marked (*a* @ *b*) = *get-all-levels-of-marked* *a* @ *get-all-levels-of-marked* *b*
⟨proof⟩

lemma *in-get-all-levels-of-marked-iff-decomp*:

$i \in \text{set } (\text{get-all-levels-of-marked } M) \longleftrightarrow (\exists c K c'. M = c @ \text{Marked } K i \# c') \text{ (is } ?A \longleftrightarrow ?B)$
 <proof>

lemma *get-rev-level-less-max-get-all-levels-of-marked:*
get-rev-level $M n L \leq \text{Max } (\text{set } (n \# \text{get-all-levels-of-marked } M))$
 <proof>

lemma *get-rev-level-ge-min-get-all-levels-of-marked:*
assumes *atm-of* $L \in \text{atm-of } ' \text{ lits-of-l } M$
shows *get-rev-level* $M n L \geq \text{Min } (\text{set } (n \# \text{get-all-levels-of-marked } M))$
 <proof>

lemma *get-all-levels-of-marked-rev-eq-rev-get-all-levels-of-marked[simp]:*
get-all-levels-of-marked (*rev* M) = *rev* (*get-all-levels-of-marked* M)
 <proof>

lemma *get-maximum-possible-level-max-get-all-levels-of-marked:*
get-maximum-possible-level $M = \text{Max } (\text{insert } 0 (\text{set } (\text{get-all-levels-of-marked } M)))$
 <proof>

lemma *get-rev-level-in-levels-of-marked:*
get-rev-level $M n L \in \{0, n\} \cup \text{set } (\text{get-all-levels-of-marked } M)$
 <proof>

lemma *get-rev-level-in-atms-in-levels-of-marked:*
atm-of $L \in \text{atm-of } ' (\text{lits-of-l } M) \implies$
get-rev-level $M n L \in \{n\} \cup \text{set } (\text{get-all-levels-of-marked } M)$
 <proof>

lemma *get-all-levels-of-marked-no-marked:*
 $(\forall l \in \text{set } Ls. \neg \text{is-marked } l) \longleftrightarrow \text{get-all-levels-of-marked } Ls = []$
 <proof>

lemma *get-level-in-levels-of-marked:*
get-level $M L \in \{0\} \cup \text{set } (\text{get-all-levels-of-marked } M)$
 <proof>

The zero is here to avoid empty-list issues with *last*:

lemma *get-level-get-rev-level-get-all-levels-of-marked:*
assumes *atm-of* $L \notin \text{atm-of } ' (\text{lits-of-l } M)$
shows
get-level ($K @ M$) $L = \text{get-rev-level } (\text{rev } K) (\text{last } (0 \# \text{get-all-levels-of-marked } (\text{rev } M))) L$
 <proof>

lemma *get-rev-level-can-skip-correctly-ordered:*
assumes
no-dup M **and**
atm-of $L \notin \text{atm-of } ' (\text{lits-of-l } M)$ **and**
get-all-levels-of-marked $M = \text{rev } [\text{Suc } 0..<\text{Suc } (\text{length } (\text{get-all-levels-of-marked } M))]$
shows *get-rev-level* (*rev* $M @ K$) $0 L = \text{get-rev-level } K (\text{length } (\text{get-all-levels-of-marked } M)) L$
 <proof>

lemma *get-level-skip-beginning-hd-get-all-levels-of-marked:*
assumes *atm-of* $L \notin \text{atm-of } ' \text{ lits-of-l } S$ **and** *get-all-levels-of-marked* $S \neq []$
shows *get-level* ($M @ S$) $L = \text{get-rev-level } (\text{rev } M) (\text{hd } (\text{get-all-levels-of-marked } S)) L$

$\langle proof \rangle$

end

theory *CDCL-W*

imports *CDCL-Abstract-Clause-Representation List-More CDCL-W-Level Wellfounded-More*

begin

19 Weidenbach's CDCL

declare *upt.simps(2)[simp del]*

19.1 The State

We will abstract the representation of clause and clauses via two locales. We expect our representation to behave like multiset, but the internal representation can be done using list or whatever other representation.

locale *state_W-ops* =

raw-clss mset-cls insert-cls remove-lit

mset-clss union-clss in-clss insert-clss remove-from-clss

+

raw-ccls-union mset-ccls union-ccls insert-ccls remove-clit

for

— Clause

mset-cls :: *'cls* \Rightarrow *'v clause* **and**

insert-cls :: *'v literal* \Rightarrow *'cls* \Rightarrow *'cls* **and**

remove-lit :: *'v literal* \Rightarrow *'cls* \Rightarrow *'cls* **and**

— Multiset of Clauses

mset-clss :: *'clss* \Rightarrow *'v clauses* **and**

union-clss :: *'clss* \Rightarrow *'clss* \Rightarrow *'clss* **and**

in-clss :: *'cls* \Rightarrow *'clss* \Rightarrow *bool* **and**

insert-clss :: *'cls* \Rightarrow *'clss* \Rightarrow *'clss* **and**

remove-from-clss :: *'cls* \Rightarrow *'clss* \Rightarrow *'clss* **and**

mset-ccls :: *'ccls* \Rightarrow *'v clause* **and**

union-ccls :: *'ccls* \Rightarrow *'ccls* \Rightarrow *'ccls* **and**

insert-ccls :: *'v literal* \Rightarrow *'ccls* \Rightarrow *'ccls* **and**

remove-clit :: *'v literal* \Rightarrow *'ccls* \Rightarrow *'ccls*

+

fixes

ccls-of-cls :: *'cls* \Rightarrow *'ccls* **and**

cls-of-ccls :: *'ccls* \Rightarrow *'cls* **and**

trail :: *'st* \Rightarrow (*'v, nat, 'v clause*) *marked-lits* **and**

hd-raw-trail :: *'st* \Rightarrow (*'v, nat, 'cls*) *marked-lit* **and**

raw-init-clss :: *'st* \Rightarrow *'clss* **and**

raw-learned-clss :: *'st* \Rightarrow *'clss* **and**

backtrack-lvl :: *'st* \Rightarrow *nat* **and**

raw-conflicting :: *'st* \Rightarrow *'ccls option* **and**

cons-trail :: (*'v, nat, 'cls*) *marked-lit* \Rightarrow *'st* \Rightarrow *'st* **and**

tl-trail :: *'st* \Rightarrow *'st* **and**

add-init-cls :: *'cls* \Rightarrow *'st* \Rightarrow *'st* **and**

```

add-learned-clb :: 'cls ⇒ 'st ⇒ 'st and
remove-clb :: 'cls ⇒ 'st ⇒ 'st and
update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
update-conflicting :: 'ccls option ⇒ 'st ⇒ 'st and

init-state :: 'clss ⇒ 'st and
restart-state :: 'st ⇒ 'st
assumes
  mset-ccls-ccls-of-clb[simp]:
    mset-ccls (ccls-of-clb C) = mset-clb C and
  mset-clb-clb-of-ccls[simp]:
    mset-clb (clb-of-ccls D) = mset-ccls D and
  ex-mset-clb: ∃ a. mset-clb a = E
begin
fun mmset-of-mlit :: ('a, 'b, 'cls) marked-lit ⇒ ('a, 'b, 'v clause) marked-lit
  where
    mmset-of-mlit (Propagated L C) = Propagated L (mset-clb C) |
    mmset-of-mlit (Marked L i) = Marked L i

lemma lit-of-mmset-of-mlit[simp]:
  lit-of (mmset-of-mlit a) = lit-of a
  ⟨proof⟩

lemma lit-of-mmset-of-mlit-set-lit-of-l[simp]:
  lit-of ' mmset-of-mlit ' set M' = lits-of-l M'
  ⟨proof⟩

lemma map-mmset-of-mlit-true-annotb-true-clb[simp]:
  map mmset-of-mlit M' ⊨as C ⟷ M' ⊨as C
  ⟨proof⟩

abbreviation init-clss ≡ λS. mset-clss (raw-init-clss S)
abbreviation learned-clss ≡ λS. mset-clss (raw-learned-clss S)
abbreviation conflicting ≡ λS. map-option mset-ccls (raw-conflicting S)

notation insert-clb (infix !++ 50)

notation in-clss (infix !∈! 50)
notation union-clss (infix ⊕ 50)
notation insert-clss (infix !++! 50)

notation union-ccls (infix !∪ 50)

definition raw-clauses :: 'st ⇒ 'clss where
  raw-clauses S = union-clss (raw-init-clss S) (raw-learned-clss S)

abbreviation clauses :: 'st ⇒ 'v clauses where
  clauses S ≡ mset-clss (raw-clauses 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 marked 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 (*'ccl'*, 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 *'cls'* is enough (needed for function *hd-raw-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* =

state_w-ops

— functions for clauses:

mset-cls insert-cls remove-lit

mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:

mset-ccls union-ccls insert-ccls remove-clit

— Conversion between conflicting and non-conflicting

ccls-of-cls cls-of-ccls

— functions about the state:

— getter:

trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting

— setter:

cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl

update-conflicting

— Some specific states:

init-state

restart-state

for

mset-cls :: *'cls* \Rightarrow *'v clause* **and**

insert-cls :: *'v literal* \Rightarrow *'cls* \Rightarrow *'cls* **and**

remove-lit :: *'v literal* \Rightarrow *'cls* \Rightarrow *'cls* **and**

mset-clss :: *'clss* \Rightarrow *'v clauses* **and**

union-clss :: *'clss* \Rightarrow *'clss* \Rightarrow *'clss* **and**

in-clss :: *'cls* \Rightarrow *'clss* \Rightarrow *bool* **and**

insert-clss :: *'cls* \Rightarrow *'clss* \Rightarrow *'clss* **and**

remove-from-clss :: *'cls* \Rightarrow *'clss* \Rightarrow *'clss* **and**

mset-ccls :: *'ccls* \Rightarrow *'v clause* **and**

union-ccls :: *'ccls* \Rightarrow *'ccls* \Rightarrow *'ccls* **and**

insert-ccls :: *'v literal* \Rightarrow *'ccls* \Rightarrow *'ccls* **and**

remove-clit :: *'v literal* \Rightarrow *'ccls* \Rightarrow *'ccls* **and**

ccls-of-cls :: 'cls \Rightarrow 'ccls **and**
cls-of-ccls :: 'ccls \Rightarrow 'cls **and**

trail :: 'st \Rightarrow ('v, nat, 'v clause) marked-lits **and**
hd-raw-trail :: 'st \Rightarrow ('v, nat, 'cls) marked-lit **and**
raw-init-clss :: 'st \Rightarrow 'clss **and**
raw-learned-clss :: 'st \Rightarrow 'clss **and**
backtrack-lvl :: 'st \Rightarrow nat **and**
raw-conflicting :: 'st \Rightarrow 'ccls option **and**

cons-trail :: ('v, nat, 'cls) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-trail :: 'st \Rightarrow 'st **and**
add-init-cls :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
add-learned-cls :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
remove-cls :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st **and**
update-conflicting :: 'ccls option \Rightarrow 'st \Rightarrow 'st **and**

init-state :: 'clss \Rightarrow 'st **and**
restart-state :: 'st \Rightarrow 'st +

assumes

hd-raw-trail: trail $S \neq [] \implies \text{mset-of-mlit } (\text{hd-raw-trail } S) = \text{hd } (\text{trail } S)$ **and**

trail-cons-trail[simp]:

$\bigwedge L \text{ st. undefined-lit } (\text{trail } st) \text{ (lit-of } L) \implies$
 $\text{trail } (\text{cons-trail } L \text{ st}) = \text{mset-of-mlit } L \# \text{ trail } st$ **and**

trail-tl-trail[simp]: $\bigwedge st. \text{trail } (\text{tl-trail } st) = \text{tl } (\text{trail } st)$ **and**

trail-add-init-cls[simp]:

$\bigwedge st \ C. \text{no-dup } (\text{trail } st) \implies \text{trail } (\text{add-init-cls } C \text{ st}) = \text{trail } st$ **and**

trail-add-learned-cls[simp]:

$\bigwedge C \text{ st. no-dup } (\text{trail } st) \implies \text{trail } (\text{add-learned-cls } C \text{ st}) = \text{trail } st$ **and**

trail-remove-cls[simp]:

$\bigwedge C \text{ st. trail } (\text{remove-cls } C \text{ st}) = \text{trail } st$ **and**

trail-update-backtrack-lvl[simp]: $\bigwedge st \ C. \text{trail } (\text{update-backtrack-lvl } C \text{ st}) = \text{trail } st$ **and**

trail-update-conflicting[simp]: $\bigwedge C \text{ st. trail } (\text{update-conflicting } C \text{ st}) = \text{trail } st$ **and**

init-clss-cons-trail[simp]:

$\bigwedge M \text{ st. undefined-lit } (\text{trail } st) \text{ (lit-of } M) \implies$
 $\text{init-clss } (\text{cons-trail } M \text{ st}) = \text{init-clss } st$

and

init-clss-tl-trail[simp]:

$\bigwedge st. \text{init-clss } (\text{tl-trail } st) = \text{init-clss } st$ **and**

init-clss-add-init-cls[simp]:

$\bigwedge st \ C. \text{no-dup } (\text{trail } st) \implies \text{init-clss } (\text{add-init-cls } C \text{ st}) = \{\# \text{mset-cls } C \# \} + \text{init-clss } st$
and

init-clss-add-learned-cls[simp]:

$\bigwedge C \text{ st. no-dup } (\text{trail } st) \implies \text{init-clss } (\text{add-learned-cls } C \text{ st}) = \text{init-clss } st$ **and**

init-clss-remove-cls[simp]:

$\bigwedge C \text{ st. init-clss } (\text{remove-cls } C \text{ st}) = \text{removeAll-mset } (\text{mset-cls } C) (\text{init-clss } st)$ **and**

init-clss-update-backtrack-lvl[simp]:

$\bigwedge st \ C. \text{init-clss } (\text{update-backtrack-lvl } C \text{ st}) = \text{init-clss } st$ **and**

init-clss-update-conflicting[simp]:

$\bigwedge C \text{ st. init-clss } (\text{update-conflicting } C \text{ st}) = \text{init-clss } st$ **and**

learned-clss-cons-trail[simp]:

$\bigwedge M \text{ st. undefined-lit } (\text{trail } st) \text{ (lit-of } M) \implies$

$\text{learned-clss} (\text{cons-trail } M \text{ st}) = \text{learned-clss st}$ **and**
 $\text{learned-clss-tl-trail[simp]}:$
 $\bigwedge st. \text{learned-clss} (\text{tl-trail st}) = \text{learned-clss st}$ **and**
 $\text{learned-clss-add-init-cls[simp]}:$
 $\bigwedge st \ C. \text{no-dup} (\text{trail st}) \implies \text{learned-clss} (\text{add-init-cls } C \text{ st}) = \text{learned-clss st}$ **and**
 $\text{learned-clss-add-learned-cls[simp]}:$
 $\bigwedge C \text{ st. no-dup} (\text{trail st}) \implies$
 $\text{learned-clss} (\text{add-learned-cls } C \text{ st}) = \{\#mset-cls \ C\# \} + \text{learned-clss st}$ **and**
 $\text{learned-clss-remove-cls[simp]}:$
 $\bigwedge C \text{ st. learned-clss} (\text{remove-cls } C \text{ st}) = \text{removeAll-mset} (\text{mset-cls } C) (\text{learned-clss st})$ **and**
 $\text{learned-clss-update-backtrack-lvl[simp]}:$
 $\bigwedge st \ C. \text{learned-clss} (\text{update-backtrack-lvl } C \text{ st}) = \text{learned-clss st}$ **and**
 $\text{learned-clss-update-conflicting[simp]}:$
 $\bigwedge C \text{ st. learned-clss} (\text{update-conflicting } C \text{ st}) = \text{learned-clss st}$ **and**

$\text{backtrack-lvl-cons-trail[simp]}:$
 $\bigwedge M \text{ st. undefined-lit} (\text{trail st}) (\text{lit-of } M) \implies$
 $\text{backtrack-lvl} (\text{cons-trail } M \text{ st}) = \text{backtrack-lvl st}$ **and**
 $\text{backtrack-lvl-tl-trail[simp]}:$
 $\bigwedge st. \text{backtrack-lvl} (\text{tl-trail st}) = \text{backtrack-lvl st}$ **and**
 $\text{backtrack-lvl-add-init-cls[simp]}:$
 $\bigwedge st \ C. \text{no-dup} (\text{trail st}) \implies \text{backtrack-lvl} (\text{add-init-cls } C \text{ st}) = \text{backtrack-lvl st}$ **and**
 $\text{backtrack-lvl-add-learned-cls[simp]}:$
 $\bigwedge C \text{ st. no-dup} (\text{trail st}) \implies \text{backtrack-lvl} (\text{add-learned-cls } C \text{ st}) = \text{backtrack-lvl st}$ **and**
 $\text{backtrack-lvl-remove-cls[simp]}:$
 $\bigwedge C \text{ st. backtrack-lvl} (\text{remove-cls } C \text{ st}) = \text{backtrack-lvl st}$ **and**
 $\text{backtrack-lvl-update-backtrack-lvl[simp]}:$
 $\bigwedge st \ k. \text{backtrack-lvl} (\text{update-backtrack-lvl } k \text{ st}) = k$ **and**
 $\text{backtrack-lvl-update-conflicting[simp]}:$
 $\bigwedge C \text{ st. backtrack-lvl} (\text{update-conflicting } C \text{ st}) = \text{backtrack-lvl st}$ **and**

$\text{conflicting-cons-trail[simp]}:$
 $\bigwedge M \text{ st. undefined-lit} (\text{trail st}) (\text{lit-of } M) \implies$
 $\text{conflicting} (\text{cons-trail } M \text{ st}) = \text{conflicting st}$ **and**
 $\text{conflicting-tl-trail[simp]}:$
 $\bigwedge st. \text{conflicting} (\text{tl-trail st}) = \text{conflicting st}$ **and**
 $\text{conflicting-add-init-cls[simp]}:$
 $\bigwedge st \ C. \text{no-dup} (\text{trail st}) \implies \text{conflicting} (\text{add-init-cls } C \text{ st}) = \text{conflicting st}$ **and**
 $\text{conflicting-add-learned-cls[simp]}:$
 $\bigwedge C \text{ st. no-dup} (\text{trail st}) \implies \text{conflicting} (\text{add-learned-cls } C \text{ st}) = \text{conflicting st}$
and
 $\text{conflicting-remove-cls[simp]}:$
 $\bigwedge C \text{ st. conflicting} (\text{remove-cls } C \text{ st}) = \text{conflicting st}$ **and**
 $\text{conflicting-update-backtrack-lvl[simp]}:$
 $\bigwedge st \ C. \text{conflicting} (\text{update-backtrack-lvl } C \text{ st}) = \text{conflicting st}$ **and**
 $\text{conflicting-update-conflicting[simp]}:$
 $\bigwedge C \text{ st. raw-conflicting} (\text{update-conflicting } C \text{ st}) = C$ **and**

$\text{init-state-trail[simp]}: \bigwedge N. \text{trail} (\text{init-state } N) = []$ **and**
 $\text{init-state-clss[simp]}: \bigwedge N. (\text{init-clss} (\text{init-state } N)) = \text{mset-clss } N$ **and**
 $\text{init-state-learned-clss[simp]}: \bigwedge N. \text{learned-clss} (\text{init-state } N) = \{\#\}$ **and**
 $\text{init-state-backtrack-lvl[simp]}: \bigwedge N. \text{backtrack-lvl} (\text{init-state } N) = 0$ **and**
 $\text{init-state-conflicting[simp]}: \bigwedge N. \text{conflicting} (\text{init-state } N) = \text{None}$ **and**

$\text{trail-restart-state[simp]}: \text{trail} (\text{restart-state } S) = []$ **and**

$init-clss-restart-state[simp]: init-clss (restart-state S) = init-clss S$ **and**
 $learned-clss-restart-state[intro]:$
 $learned-clss (restart-state S) \subseteq \# learned-clss S$ **and**
 $backtrack-lvl-restart-state[simp]: backtrack-lvl (restart-state S) = 0$ **and**
 $conflicting-restart-state[simp]: conflicting (restart-state S) = None$
begin

lemma
shows
 $clauses-cons-trail[simp]:$
 $undefined-lit (trail S) (lit-of M) \implies clauses (cons-trail M S) = clauses S$ **and**

$clss-tl-trail[simp]: clauses (tl-trail S) = clauses S$ **and**
 $clauses-add-learned-cls-unfolded:$
 $no-dup (trail S) \implies clauses (add-learned-cls U S) =$
 $\{\#mset-cls U\} + learned-clss S + init-clss S$
and
 $clauses-add-init-cls[simp]:$
 $no-dup (trail S) \implies$
 $clauses (add-init-cls N S) = \{\#mset-cls N\} + init-clss S + learned-clss S$ **and**
 $clauses-update-backtrack-lvl[simp]: clauses (update-backtrack-lvl k S) = clauses S$ **and**
 $clauses-update-conflicting[simp]: clauses (update-conflicting D S) = clauses S$ **and**
 $clauses-remove-cls[simp]:$
 $clauses (remove-cls C S) = removeAll-mset (mset-cls C) (clauses S)$ **and**
 $clauses-add-learned-cls[simp]:$
 $no-dup (trail S) \implies clauses (add-learned-cls C S) = \{\#mset-cls C\} + clauses S$ **and**
 $clauses-restart[simp]: clauses (restart-state S) \subseteq \# clauses S$ **and**
 $clauses-init-state[simp]: \bigwedge N. clauses (init-state N) = mset-clss N$
 $\langle proof \rangle$

abbreviation $state :: 'st \Rightarrow ('v, nat, 'v clause) \text{ marked-lit list } \times 'v clauses \times 'v clauses$
 $\times nat \times 'v clause \text{ option}$ **where**
 $state S \equiv (trail S, init-clss S, learned-clss S, backtrack-lvl S, conflicting S)$

abbreviation $incr-lvl :: 'st \Rightarrow 'st$ **where**
 $incr-lvl S \equiv update-backtrack-lvl (backtrack-lvl S + 1) S$

definition $state-eq :: 'st \Rightarrow 'st \Rightarrow bool$ (**infix** ~ 50) **where**
 $S \sim T \iff state S = state T$

lemma $state-eq-ref[simp, intro]:$
 $S \sim S$
 $\langle proof \rangle$

lemma $state-eq-sym:$
 $S \sim T \iff T \sim S$
 $\langle proof \rangle$

lemma $state-eq-trans:$
 $S \sim T \implies T \sim U \implies S \sim U$
 $\langle proof \rangle$

lemma
shows
 $state-eq-trail: S \sim T \implies trail S = trail T$ **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: $S \sim T \implies \text{undefined-lit } (\text{trail } S) L = \text{undefined-lit } (\text{trail } T) L$
 <proof>

lemma *state-eq-raw-conflicting-None*:

$S \sim T \implies \text{conflicting } T = \text{None} \implies \text{raw-conflicting } S = \text{None}$
 <proof>

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-raw-conflicting-None*

lemma *atms-of-ms-learned-clss-restart-state-in-atms-of-ms-learned-clssI*[intro]:

$x \in \text{atms-of-mm } (\text{learned-clss } (\text{restart-state } S)) \implies x \in \text{atms-of-mm } (\text{learned-clss } S)$
 <proof>

function *reduce-trail-to* :: 'a list \Rightarrow 'st \Rightarrow 'st **where**

reduce-trail-to $F S =$
 (if length (trail S) = length $F \vee \text{trail } S = []$ then S else *reduce-trail-to* F (tl-trail S))
 <proof>

termination

<proof>

declare *reduce-trail-to.simps*[simp del]

lemma

shows

reduce-trail-to-nil[simp]: trail $S = [] \implies \text{reduce-trail-to } F S = S$ **and**
reduce-trail-to-eq-length[simp]: length (trail S) = length $F \implies \text{reduce-trail-to } F S = S$
 <proof>

lemma *reduce-trail-to-length-ne*:

length (trail S) \neq length $F \implies \text{trail } S \neq [] \implies$
 $\text{reduce-trail-to } F S = \text{reduce-trail-to } F (\text{tl-trail } S)$
 <proof>

lemma *trail-reduce-trail-to-length-le*:

assumes length $F >$ length (trail S)
shows trail (reduce-trail-to $F S$) = []
 <proof>

lemma *trail-reduce-trail-to-nil*[simp]:

trail (reduce-trail-to [] S) = []
 <proof>

lemma *clauses-reduce-trail-to-nil*:

clauses (reduce-trail-to [] S) = clauses S
 <proof>

lemma *reduce-trail-to-skip-beginning*:
assumes $\text{trail } S = F' @ F$
shows $\text{trail } (\text{reduce-trail-to } F S) = F$
 $\langle \text{proof} \rangle$

lemma *clauses-reduce-trail-to[simp]*:
 $\text{clauses } (\text{reduce-trail-to } F S) = \text{clauses } S$
 $\langle \text{proof} \rangle$

lemma *conflicting-update-trail[simp]*:
 $\text{conflicting } (\text{reduce-trail-to } F S) = \text{conflicting } S$
 $\langle \text{proof} \rangle$

lemma *backtrack-lvl-update-trail[simp]*:
 $\text{backtrack-lvl } (\text{reduce-trail-to } F S) = \text{backtrack-lvl } S$
 $\langle \text{proof} \rangle$

lemma *init-clss-update-trail[simp]*:
 $\text{init-clss } (\text{reduce-trail-to } F S) = \text{init-clss } S$
 $\langle \text{proof} \rangle$

lemma *learned-clss-update-trail[simp]*:
 $\text{learned-clss } (\text{reduce-trail-to } F S) = \text{learned-clss } S$
 $\langle \text{proof} \rangle$

lemma *raw-conflicting-reduce-trail-to[simp]*:
 $\text{raw-conflicting } (\text{reduce-trail-to } F S) = \text{None} \longleftrightarrow \text{raw-conflicting } S = \text{None}$
 $\langle \text{proof} \rangle$

lemma *trail-eq-reduce-trail-to-eq*:
 $\text{trail } S = \text{trail } T \implies \text{trail } (\text{reduce-trail-to } F S) = \text{trail } (\text{reduce-trail-to } F T)$
 $\langle \text{proof} \rangle$

lemma *reduce-trail-to-state-eq_{NOT}-compatible*:
assumes $ST: S \sim T$
shows $\text{reduce-trail-to } F S \sim \text{reduce-trail-to } F T$
 $\langle \text{proof} \rangle$

lemma *reduce-trail-to-trail-tl-trail-decomp[simp]*:
 $\text{trail } S = F' @ \text{Marked } K d \# F \implies (\text{trail } (\text{reduce-trail-to } F S)) = F$
 $\langle \text{proof} \rangle$

lemma *reduce-trail-to-add-learned-cls[simp]*:
 $\text{no-dup } (\text{trail } S) \implies$
 $\text{trail } (\text{reduce-trail-to } F (\text{add-learned-cls } C S)) = \text{trail } (\text{reduce-trail-to } F S)$
 $\langle \text{proof} \rangle$

lemma *reduce-trail-to-add-init-cls[simp]*:
 $\text{no-dup } (\text{trail } S) \implies$
 $\text{trail } (\text{reduce-trail-to } F (\text{add-init-cls } C S)) = \text{trail } (\text{reduce-trail-to } F S)$
 $\langle \text{proof} \rangle$

lemma *reduce-trail-to-remove-learned-cls[simp]*:

trail (*reduce-trail-to* *F* (*remove-cls* *C S*)) = *trail* (*reduce-trail-to* *F S*)
 ⟨*proof*⟩

lemma *reduce-trail-to-update-conflicting*[*simp*]:
trail (*reduce-trail-to* *F* (*update-conflicting* *C S*)) = *trail* (*reduce-trail-to* *F S*)
 ⟨*proof*⟩

lemma *reduce-trail-to-update-backtrack-lvl*[*simp*]:
trail (*reduce-trail-to* *F* (*update-backtrack-lvl* *C S*)) = *trail* (*reduce-trail-to* *F S*)
 ⟨*proof*⟩

lemma *in-get-all-marked-decomposition-marked-or-empty*:
assumes (*a*, *b*) ∈ *set* (*get-all-marked-decomposition* *M*)
shows *a* = [] ∨ (*is-marked* (*hd a*))
 ⟨*proof*⟩

lemma *reduce-trail-to-length*:
length M = *length M'* ⇒ *reduce-trail-to M S* = *reduce-trail-to M' S*
 ⟨*proof*⟩

lemma *trail-reduce-trail-to-drop*:
trail (*reduce-trail-to* *F S*) =
 (*if length* (*trail S*) ≥ *length F*
 then *drop* (*length* (*trail S*) − *length F*) (*trail S*)
 else [])
 ⟨*proof*⟩

lemma *in-get-all-marked-decomposition-trail-update-trail*[*simp*]:
assumes *H*: (*L* # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*))
shows *trail* (*reduce-trail-to M1 S*) = *M1*
 ⟨*proof*⟩

lemma *raw-conflicting-cons-trail*[*simp*]:
assumes *undefined-lit* (*trail S*) (*lit-of L*)
shows
raw-conflicting (*cons-trail L S*) = *None* ⇔ *raw-conflicting S* = *None*
 ⟨*proof*⟩

lemma *raw-conflicting-add-init-cls*[*simp*]:
no-dup (*trail S*) ⇒
raw-conflicting (*add-init-cls C S*) = *None* ⇔ *raw-conflicting S* = *None*
 ⟨*proof*⟩

lemma *raw-conflicting-add-learned-cls*[*simp*]:
no-dup (*trail S*) ⇒
raw-conflicting (*add-learned-cls C S*) = *None* ⇔ *raw-conflicting S* = *None*
 ⟨*proof*⟩

lemma *raw-conflicting-update-backtrack-lvl*[*simp*]:
raw-conflicting (*update-backtrack-lvl k S*) = *None* ⇔ *raw-conflicting S* = *None*
 ⟨*proof*⟩

end — end of *state_W* locale

19.2 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 clauses:

mset-cls insert-cls remove-lit

mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:

mset-ccls union-ccls insert-ccls remove-clit

— conversion

ccls-of-cls cls-of-ccls

— functions for the state:

— access functions:

trail hd-trail trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting

— changing state:

*cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
update-conflicting*

— get state:

init-state

restart-state

for

mset-cls :: '*cls* ⇒ '*v* clause **and**

insert-cls :: '*v* literal ⇒ '*cls* ⇒ '*cls* **and**

remove-lit :: '*v* literal ⇒ '*cls* ⇒ '*cls* **and**

mset-clss :: '*clss* ⇒ '*v* clauses **and**

union-clss :: '*clss* ⇒ '*clss* ⇒ '*clss* **and**

in-clss :: '*cls* ⇒ '*clss* ⇒ bool **and**

insert-clss :: '*cls* ⇒ '*clss* ⇒ '*clss* **and**

remove-from-clss :: '*cls* ⇒ '*clss* ⇒ '*clss* **and**

mset-ccls :: '*ccls* ⇒ '*v* clause **and**

union-ccls :: '*ccls* ⇒ '*ccls* ⇒ '*ccls* **and**

insert-ccls :: '*v* literal ⇒ '*ccls* ⇒ '*ccls* **and**

remove-clit :: '*v* literal ⇒ '*ccls* ⇒ '*ccls* **and**

ccls-of-cls :: '*cls* ⇒ '*ccls* **and**

cls-of-ccls :: '*ccls* ⇒ '*cls* **and**

trail :: '*st* ⇒ ('*v*, nat, '*v* clause) marked-lits **and**

hd-trail :: '*st* ⇒ ('*v*, nat, '*cls*) marked-lit **and**

raw-init-clss :: '*st* ⇒ '*clss* **and**

raw-learned-clss :: '*st* ⇒ '*clss* **and**

backtrack-lvl :: '*st* ⇒ nat **and**

raw-conflicting :: '*st* ⇒ '*ccls* option **and**

cons-trail :: ('*v*, nat, '*cls*) marked-lit ⇒ '*st* ⇒ '*st* **and**

tl-trail :: '*st* ⇒ '*st* **and**

add-init-cls :: '*cls* ⇒ '*st* ⇒ '*st* **and**

add-learned-cls :: '*cls* ⇒ '*st* ⇒ '*st* **and**

remove-cls :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st **and**
update-conflicting :: 'ccls option \Rightarrow 'st \Rightarrow 'st **and**

init-state :: 'clss \Rightarrow 'st **and**
restart-state :: 'st \Rightarrow 'st

begin

inductive *propagate* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

propagate-rule: *conflicting S* = None \Rightarrow

E ! \in ! *raw-clauses S* \Rightarrow

L \in # *mset-cls E* \Rightarrow

trail S \models as *CNot* (*mset-cls* (*remove-lit L E*)) \Rightarrow

undefined-lit (*trail S*) *L* \Rightarrow

T \sim *cons-trail* (*Propagated L E*) *S* \Rightarrow

propagate S T

inductive-cases *propagateE*: *propagate S T*

inductive *conflict* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

conflict-rule:

conflicting S = None \Rightarrow

D ! \in ! *raw-clauses S* \Rightarrow

trail S \models as *CNot* (*mset-cls D*) \Rightarrow

T \sim *update-conflicting* (*Some* (*ccls-of-cls D*)) *S* \Rightarrow

conflict S T

inductive-cases *conflictE*: *conflict S T*

inductive *backtrack* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

backtrack-rule:

raw-conflicting S = *Some D* \Rightarrow

L \in # *mset-ccls D* \Rightarrow

(*Marked K* (*i*+1) # *M1*, *M2*) \in *set* (*get-all-marked-decomposition* (*trail S*)) \Rightarrow

get-level (*trail S*) *L* = *backtrack-lvl S* \Rightarrow

get-level (*trail S*) *L* = *get-maximum-level* (*trail S*) (*mset-ccls D*) \Rightarrow

get-maximum-level (*trail S*) (*mset-ccls* (*remove-clit L D*)) \equiv *i* \Rightarrow

T \sim *cons-trail* (*Propagated L* (*cls-of-ccls D*))

(*reduce-trail-to M1*

(*add-learned-cls* (*cls-of-ccls D*)

(*update-backtrack-lvl i*

(*update-conflicting None S*)))) \Rightarrow

backtrack S T

inductive-cases *backtrackE*: *backtrack S T*

thm *backtrackE*

inductive *decide* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

decide-rule:

conflicting S = None \Rightarrow

undefined-lit (*trail S*) *L* \Rightarrow

atm-of L \in *atms-of-mm* (*init-clss S*) \Rightarrow

T \sim *cons-trail* (*Marked L* (*backtrack-lvl S* + 1)) (*incr-lvl S*) \Rightarrow

decide S T

inductive-cases *decideE*: *decide S T*

inductive *skip* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

skip-rule:

trail S = Propagated L C' # M \Rightarrow
raw-conflicting S = Some E \Rightarrow
 $-L \notin \# \text{ mset-ccls } E \Rightarrow$
**mset-ccls E* $\neq \{\#\}$ \Rightarrow*
**T* \sim *tl-trail S* \Rightarrow*
skip S T

inductive-cases *skipE*: *skip S T*

get-maximum-level (Propagated L (C + $\{\#L\#\}$) # M) D = k \vee k = 0 (that was in a previous version of the book) is equivalent to *get-maximum-level (Propagated L (C + $\{\#L\#\}$) # M) D = k*, when the structural invariants holds.

inductive *resolve* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

resolve-rule: *trail S $\neq [] \Rightarrow$*

hd-raw-trail S = Propagated L E \Rightarrow
L $\in \# \text{ mset-cls } E \Rightarrow$
raw-conflicting S = Some D' \Rightarrow
 $-L \in \# \text{ mset-ccls } D' \Rightarrow$
get-maximum-level (trail S) (mset-ccls (remove-clit (-L) D')) = backtrack-lvl S \Rightarrow
T* \sim *update-conflicting (Some (union-ccls (remove-clit (-L) D') (ccls-of-cls (remove-lit L E))))
*(*tl-trail S*) \Rightarrow*
resolve S T

inductive-cases *resolveE*: *resolve S T*

inductive *restart* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

restart: *state S = (M, N, U, k, None) $\Rightarrow \neg M \models_{asm} \text{ clauses } S$*
 \Rightarrow *T \sim restart-state S*
 \Rightarrow *restart S T*

inductive-cases *restartE*: *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:

conflicting S = None \Rightarrow
C ! \in ! raw-learned-clss S \Rightarrow
 $\neg(\text{trail } S) \models_{asm} \text{ clauses } S \Rightarrow$
mset-cls C* \notin *set (get-all-mark-of-propagated (trail S)) \Rightarrow
**mset-cls C* $\notin \# \text{ init-clss } S \Rightarrow$*
T* \sim *remove-cls C S \Rightarrow
forget S T

inductive-cases *forgetE*: *forget S T*

inductive *cdcl_W-rf* :: 'st \Rightarrow 'st \Rightarrow bool **for** *S* :: 'st **where**

restart: *restart S T \Rightarrow cdcl_W-rf S T |*

forget: *forget S T \Rightarrow cdcl_W-rf S T*

inductive *cdcl_W-bj* :: 'st \Rightarrow 'st \Rightarrow bool **where**

skip: *skip S S' \Rightarrow cdcl_W-bj S S' |*

resolve: $\text{resolve } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S' \mid$
backtrack: $\text{backtrack } S \ S' \implies \text{cdcl}_W\text{-bj } S \ S'$

inductive-cases $\text{cdcl}_W\text{-bjE}$: $\text{cdcl}_W\text{-bj } S \ T$

inductive $\text{cdcl}_W\text{-o} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**

decide: $\text{decide } S \ S' \implies \text{cdcl}_W\text{-o } S \ S' \mid$

bj: $\text{cdcl}_W\text{-bj } S \ S' \implies \text{cdcl}_W\text{-o } S \ S'$

inductive $\text{cdcl}_W :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**

propagate: $\text{propagate } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$

conflict: $\text{conflict } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$

other: $\text{cdcl}_W\text{-o } S \ S' \implies \text{cdcl}_W \ S \ S' \mid$

rf: $\text{cdcl}_W\text{-rf } S \ S' \implies \text{cdcl}_W \ S \ S'$

lemma *rtrancpl-propagate-is-rtrancpl-cdcl_W*:

$\text{propagate}^{**} \ S \ S' \implies \text{cdcl}_W^{**} \ S \ S'$

$\langle \text{proof} \rangle$

lemma *cdcl_W-all-rules-induct*[consumes 1, case-names propagate conflict forget restart decide skip
resolve backtrack]:

fixes $S :: 'st$

assumes

cdcl_W : $\text{cdcl}_W \ S \ S'$ **and**

propagate: $\bigwedge T. \text{propagate } S \ T \implies P \ S \ T$ **and**

conflict: $\bigwedge T. \text{conflict } S \ T \implies P \ S \ T$ **and**

forget: $\bigwedge T. \text{forget } S \ T \implies P \ S \ T$ **and**

restart: $\bigwedge T. \text{restart } S \ T \implies P \ S \ T$ **and**

decide: $\bigwedge T. \text{decide } S \ T \implies P \ S \ T$ **and**

skip: $\bigwedge T. \text{skip } S \ T \implies P \ S \ T$ **and**

resolve: $\bigwedge T. \text{resolve } S \ T \implies P \ S \ T$ **and**

backtrack: $\bigwedge T. \text{backtrack } S \ T \implies P \ S \ T$

shows $P \ S \ S'$

$\langle \text{proof} \rangle$

lemma *cdcl_W-all-induct*[consumes 1, case-names propagate conflict forget restart decide skip
resolve backtrack]:

fixes $S :: 'st$

assumes

cdcl_W : $\text{cdcl}_W \ S \ S'$ **and**

propagateH: $\bigwedge C \ L \ T. \text{conflicting } S = \text{None} \implies$

$C \ !\in \text{raw-clauses } S \implies$

$L \in \# \text{mset-cls } C \implies$

$\text{trail } S \models_{\text{as}} C \text{Not } (\text{remove1-mset } L \ (\text{mset-cls } C)) \implies$

$\text{undefined-lit } (\text{trail } S) \ L \implies$

$T \sim \text{cons-trail } (\text{Propagated } L \ C) \ S \implies$

$P \ S \ T$ **and**

conflictH: $\bigwedge D \ T. \text{conflicting } S = \text{None} \implies$

$D \ !\in \text{raw-clauses } S \implies$

$\text{trail } S \models_{\text{as}} C \text{Not } (\text{mset-cls } D) \implies$

$T \sim \text{update-conflicting } (\text{Some } (\text{ccls-of-cls } D)) \ S \implies$

$P \ S \ T$ **and**

forgetH: $\bigwedge C \ U \ T. \text{conflicting } S = \text{None} \implies$

$C \ !\in \text{raw-learned-clss } S \implies$

$\neg(\text{trail } S) \models_{\text{asm}} \text{clauses } S \implies$

$mset-cls\ C \notin set\ (get-all-mark-of-propagated\ (trail\ S)) \implies$
 $mset-cls\ C \notin \# init-clss\ S \implies$
 $T \sim remove-cls\ C\ S \implies$
 $P\ S\ T\ \mathbf{and}$
 $restartH: \bigwedge T. \neg trail\ S \models_{asm} clauses\ S \implies$
 $conflicting\ S = None \implies$
 $T \sim restart-state\ S \implies$
 $P\ S\ T\ \mathbf{and}$
 $decideH: \bigwedge L\ T. conflicting\ S = None \implies$
 $undefined-lit\ (trail\ S)\ L \implies$
 $atm-of\ L \in atms-of-mm\ (init-clss\ S) \implies$
 $T \sim cons-trail\ (Marked\ L\ (backtrack-lvl\ S + 1))\ (incr-lvl\ S) \implies$
 $P\ S\ T\ \mathbf{and}$
 $skipH: \bigwedge L\ C'\ M\ E\ T.$
 $trail\ S = Propagated\ L\ C'\ \# M \implies$
 $raw-conflicting\ S = Some\ E \implies$
 $-L \notin \# mset-ccls\ E \implies mset-ccls\ E \neq \{\#\} \implies$
 $T \sim tl-trail\ S \implies$
 $P\ S\ T\ \mathbf{and}$
 $resolveH: \bigwedge L\ E\ M\ D\ T.$
 $trail\ S = Propagated\ L\ (mset-cls\ E)\ \# M \implies$
 $L \in \# mset-cls\ E \implies$
 $hd-raw-trail\ S = Propagated\ L\ E \implies$
 $raw-conflicting\ S = Some\ D \implies$
 $-L \in \# mset-ccls\ D \implies$
 $get-maximum-level\ (trail\ S)\ (mset-ccls\ (remove-clit\ (-L)\ D)) = backtrack-lvl\ S \implies$
 $T \sim update-conflicting$
 $(Some\ (union-ccls\ (remove-clit\ (-L)\ D)\ (ccls-of-cls\ (remove-lit\ L\ E))))\ (tl-trail\ S) \implies$
 $P\ S\ T\ \mathbf{and}$
 $backtrackH: \bigwedge L\ D\ K\ i\ M1\ M2\ T.$
 $raw-conflicting\ S = Some\ D \implies$
 $L \in \# mset-ccls\ D \implies$
 $(Marked\ K\ (i+1)\ \# M1,\ M2) \in set\ (get-all-marked-decomposition\ (trail\ S)) \implies$
 $get-level\ (trail\ S)\ L = backtrack-lvl\ S \implies$
 $get-level\ (trail\ S)\ L = get-maximum-level\ (trail\ S)\ (mset-ccls\ D) \implies$
 $get-maximum-level\ (trail\ S)\ (remove1-mset\ L\ (mset-ccls\ D)) \equiv i \implies$
 $T \sim cons-trail\ (Propagated\ L\ (cls-of-ccls\ D))$
 $(reduce-trail-to\ M1$
 $(add-learned-cls\ (cls-of-ccls\ D)$
 $(update-backtrack-lvl\ i$
 $(update-conflicting\ None\ S)))) \implies$
 $P\ S\ T$
shows $P\ S\ S'$
 $\langle proof \rangle$

lemma $cdcl_W\text{-}o\text{-induct}[consumes\ 1,\ case\text{-}names\ decide\ skip\ resolve\ backtrack]:$

fixes $S :: 'st$

assumes $cdcl_W: cdcl_W\text{-}o\ S\ T\ \mathbf{and}$

$decideH: \bigwedge L\ T. conflicting\ S = None \implies undefined-lit\ (trail\ S)\ L$
 $\implies atm-of\ L \in atms-of-mm\ (init-clss\ S)$
 $\implies T \sim cons-trail\ (Marked\ L\ (backtrack-lvl\ S + 1))\ (incr-lvl\ S)$
 $\implies P\ S\ T\ \mathbf{and}$

skipH: $\bigwedge L\ C'\ M\ E\ T.$

$trail\ S = Propagated\ L\ C'\ \# M \implies$
 $raw-conflicting\ S = Some\ E \implies$

$-L \notin \# \text{ mset-ccls } E \implies \text{ mset-ccls } E \neq \{\#\} \implies$
 $T \sim \text{tl-trail } S \implies$
 $P \text{ } S \text{ } T \text{ and}$
resolveH: $\bigwedge L \text{ } E \text{ } M \text{ } D \text{ } T.$
 $\text{trail } S = \text{Propagated } L \text{ (mset-cls } E) \# M \implies$
 $L \in \# \text{ mset-cls } E \implies$
 $\text{hd-raw-trail } S = \text{Propagated } L \text{ } E \implies$
 $\text{raw-conflicting } S = \text{Some } D \implies$
 $-L \in \# \text{ mset-ccls } D \implies$
 $\text{get-maximum-level } (\text{trail } S) \text{ (mset-ccls (remove-clit } (-L) \text{ } D)) = \text{backtrack-lvl } S \implies$
 $T \sim \text{update-conflicting}$
 $(\text{Some } (\text{union-ccls } (\text{remove-clit } (-L) \text{ } D) \text{ (ccls-of-cls (remove-lit } L \text{ } E)))) \text{ (tl-trail } S) \implies$
 $P \text{ } S \text{ } T \text{ and}$
backtrackH: $\bigwedge L \text{ } D \text{ } K \text{ } i \text{ } M1 \text{ } M2 \text{ } T.$
 $\text{raw-conflicting } S = \text{Some } D \implies$
 $L \in \# \text{ mset-ccls } D \implies$
 $(\text{Marked } K \text{ (} i+1 \text{) } \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S)) \implies$
 $\text{get-level } (\text{trail } S) \text{ } L = \text{backtrack-lvl } S \implies$
 $\text{get-level } (\text{trail } S) \text{ } L = \text{get-maximum-level } (\text{trail } S) \text{ (mset-ccls } D) \implies$
 $\text{get-maximum-level } (\text{trail } S) \text{ (remove1-mset } L \text{ (mset-ccls } D)) \equiv i \implies$
 $T \sim \text{cons-trail } (\text{Propagated } L \text{ (cls-of-ccls } D))$
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } (\text{cls-of-ccls } D)$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting None } S)))) \implies$
 $P \text{ } S \text{ } T$
shows $P \text{ } S \text{ } T$
 $\langle \text{proof} \rangle$

thm *cdcl_W-o.induct*
lemma *cdcl_W-o-all-rules-induct*[consumes 1, case-names decide backtrack skip resolve]:
fixes $S \text{ } T :: 'st$
assumes
 $\text{cdcl}_W\text{-o } S \text{ } T \text{ and}$
 $\bigwedge T. \text{decide } S \text{ } T \implies P \text{ } S \text{ } T \text{ and}$
 $\bigwedge T. \text{backtrack } S \text{ } T \implies P \text{ } S \text{ } T \text{ and}$
 $\bigwedge T. \text{skip } S \text{ } T \implies P \text{ } S \text{ } T \text{ and}$
 $\bigwedge T. \text{resolve } S \text{ } T \implies P \text{ } S \text{ } T$
shows $P \text{ } S \text{ } T$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-o-rule-cases*[consumes 1, case-names decide backtrack skip resolve]:
fixes $S \text{ } T :: 'st$
assumes
 $\text{cdcl}_W\text{-o } S \text{ } T \text{ and}$
 $\text{decide } S \text{ } T \implies P \text{ and}$
 $\text{backtrack } S \text{ } T \implies P \text{ and}$
 $\text{skip } S \text{ } T \implies P \text{ and}$
 $\text{resolve } S \text{ } T \implies P$
shows P
 $\langle \text{proof} \rangle$

19.3 Invariants

19.3.1 Properties of the trail

We here establish that: * the marks are exactly 1..k where k is the level * the consistency of the trail * the fact that there is no duplicate in the trail.

lemma *backtrack-lit-skipped*:

assumes

L: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**

M1: (*Marked K* (*i* + 1) # *M1*, *M2*) ∈ *set* (*get-all-marked-decomposition* (*trail S*)) **and**

no-dup: *no-dup* (*trail S*) **and**

bt-l: *backtrack-lvl S* = *length* (*get-all-levels-of-marked* (*trail S*)) **and**

order: *get-all-levels-of-marked* (*trail S*)

= *rev* [*1*..*(1*+*length* (*get-all-levels-of-marked* (*trail S*)))

shows *atm-of L* ∉ *atm-of* ‘*lits-of-l M1*

⟨*proof*⟩

lemma *cdcl_W-distinctinv-1*:

assumes

cdcl_W S S' **and**

no-dup (*trail S*) **and**

backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**

get-all-levels-of-marked (*trail S*) = *rev* [*1*..*(1*+*length* (*get-all-levels-of-marked* (*trail S*)))

shows *no-dup* (*trail S'*)

⟨*proof*⟩

lemma *cdcl_W-consistent-inv-2*:

assumes

cdcl_W S S' **and**

no-dup (*trail S*) **and**

backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**

get-all-levels-of-marked (*trail S*) = *rev* [*1*..*(1*+*length* (*get-all-levels-of-marked* (*trail S*)))

shows *consistent-interp* (*lits-of-l* (*trail S'*))

⟨*proof*⟩

lemma *cdcl_W-o-bt*:

assumes

cdcl_W-o S S' **and**

backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**

get-all-levels-of-marked (*trail S*) =

rev ([*1*..*(1*+*length* (*get-all-levels-of-marked* (*trail S*)))]) **and**

n-d[simp]: *no-dup* (*trail S*)

shows *backtrack-lvl S'* = *length* (*get-all-levels-of-marked* (*trail S'*))

⟨*proof*⟩

lemma *cdcl_W-rf-bt*:

assumes

cdcl_W-rf S S' **and**

backtrack-lvl S = *length* (*get-all-levels-of-marked* (*trail S*)) **and**

get-all-levels-of-marked (*trail S*) = *rev* [*1*..*(1*+*length* (*get-all-levels-of-marked* (*trail S*)))

shows *backtrack-lvl S'* = *length* (*get-all-levels-of-marked* (*trail S'*))

⟨*proof*⟩

lemma *cdcl_W-bt*:

assumes

$cdcl_W S S'$ and
 $backtrack_lvl S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ and
 $\text{get-all-levels-of-marked } (\text{trail } S)$
 $= \text{rev } ([1..<(1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))))]$ and
 $\text{no-dup } (\text{trail } S)$
shows $backtrack_lvl S' = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S'))$
 $\langle \text{proof} \rangle$

lemma $cdcl_W\text{-bt-level'}$:

assumes
 $cdcl_W S S'$ and
 $backtrack_lvl S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ and
 $\text{get-all-levels-of-marked } (\text{trail } S)$
 $= \text{rev } ([1..<(1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))))]$ and
 $n\text{-d: no-dup } (\text{trail } S)$
shows $\text{get-all-levels-of-marked } (\text{trail } S')$
 $= \text{rev } [1..<1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S'))]$
 $\langle \text{proof} \rangle$

We write $1 + \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$ instead of $backtrack_lvl S$ to avoid non termination of rewriting.

definition $cdcl_W\text{-M-level-inv} :: 'st \Rightarrow \text{bool}$ **where**

$cdcl_W\text{-M-level-inv } S \longleftrightarrow$
 $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$
 $\wedge \text{no-dup } (\text{trail } S)$
 $\wedge backtrack_lvl S = \text{length } (\text{get-all-levels-of-marked } (\text{trail } S))$
 $\wedge \text{get-all-levels-of-marked } (\text{trail } S)$
 $= \text{rev } [1..<1+\text{length } (\text{get-all-levels-of-marked } (\text{trail } S))]$

lemma $cdcl_W\text{-M-level-inv-decomp}$:

assumes $cdcl_W\text{-M-level-inv } S$
shows
 $\text{consistent-interp } (\text{lits-of-l } (\text{trail } S))$ and
 $\text{no-dup } (\text{trail } S)$
 $\langle \text{proof} \rangle$

lemma $cdcl_W\text{-consistent-inv}$:

fixes $S S' :: 'st$
assumes
 $cdcl_W S S'$ and
 $cdcl_W\text{-M-level-inv } S$
shows $cdcl_W\text{-M-level-inv } S'$
 $\langle \text{proof} \rangle$

lemma $rtrancp\text{-}cdcl_W\text{-consistent-inv}$:

assumes
 $cdcl_W^{**} S S'$ and
 $cdcl_W\text{-M-level-inv } S$
shows $cdcl_W\text{-M-level-inv } S'$
 $\langle \text{proof} \rangle$

lemma $trancp\text{-}cdcl_W\text{-consistent-inv}$:

assumes
 $cdcl_W^{++} S S'$ and
 $cdcl_W\text{-M-level-inv } S$

shows $cdcl_W\text{-}M\text{-level-inv } S'$
 $\langle \text{proof} \rangle$

lemma $cdcl_W\text{-}M\text{-level-inv-S0-cdcl}_W[simp]$:
 $cdcl_W\text{-}M\text{-level-inv } (\text{init-state } N)$
 $\langle \text{proof} \rangle$

lemma $cdcl_W\text{-}M\text{-level-inv-get-level-le-backtrack-lvl}$:
assumes inv : $cdcl_W\text{-}M\text{-level-inv } S$
shows $\text{get-level } (\text{trail } S) L \leq \text{backtrack-lvl } S$
 $\langle \text{proof} \rangle$

lemma $\text{backtrack-ex-decomp}$:
assumes
 $M\text{-l}$: $cdcl_W\text{-}M\text{-level-inv } S$ **and**
 $i\text{-S}$: $i < \text{backtrack-lvl } S$
shows $\exists K M1 M2. (\text{Marked } K (i + 1) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$
 $\langle \text{proof} \rangle$

19.3.2 Better-Suited Induction Principle

We generalise the induction principle defined previously: the induction case for *backtrack* now includes the assumption that *undefined-lit* $M1 L$. This helps the simplifier and thus the automation.

lemma $\text{backtrack-induction-lev}[\text{consumes } 1, \text{case-names } M\text{-level-inv backtrack}]$:
assumes
 bt : $\text{backtrack } S T$ **and**
 inv : $cdcl_W\text{-}M\text{-level-inv } S$ **and**
 backtrackH : $\bigwedge K i M1 M2 L D T.$
 $\text{raw-conflicting } S = \text{Some } D \implies$
 $L \in \# \text{ mset-ccls } D \implies$
 $(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S)) \implies$
 $\text{get-level } (\text{trail } S) L = \text{backtrack-lvl } S \implies$
 $\text{get-level } (\text{trail } S) L = \text{get-maximum-level } (\text{trail } S) (\text{mset-ccls } D) \implies$
 $\text{get-maximum-level } (\text{trail } S) (\text{remove1-mset } L (\text{mset-ccls } D)) \equiv i \implies$
 $\text{undefined-lit } M1 L \implies$
 $T \sim \text{cons-trail } (\text{Propagated } L (\text{cls-of-ccls } D))$
 $(\text{reduce-trail-to } M1$
 $(\text{add-learned-cls } (\text{cls-of-ccls } D)$
 $(\text{update-backtrack-lvl } i$
 $(\text{update-conflicting } \text{None } S)))) \implies$
 $P S T$
shows $P S T$
 $\langle \text{proof} \rangle$

lemmas $\text{backtrack-induction-lev2} = \text{backtrack-induction-lev}[\text{consumes } 2, \text{case-names } \text{backtrack}]$

lemma $cdcl_W\text{-all-induct-lev-full}$:
fixes $S :: 'st$
assumes
 $cdcl_W$: $cdcl_W S S'$ **and**
 $inv[simp]$: $cdcl_W\text{-}M\text{-level-inv } S$ **and**
 propagateH : $\bigwedge C L T. \text{conflicting } S = \text{None} \implies$
 $C !\in \text{raw-clauses } S \implies$
 $L \in \# \text{ mset-cls } C \implies$

$trail\ S \models_{as} CNot\ (remove1-mset\ L\ (mset-cl\ C)) \implies$
 $undefined-lit\ (trail\ S)\ L \implies$
 $T \sim cons-trail\ (Propagated\ L\ C)\ S \implies$
P S T and
 $conflictH: \bigwedge D\ T. \text{ conflicting } S = None \implies$
 $D !\in! raw-clauses\ S \implies$
 $trail\ S \models_{as} CNot\ (mset-cl\ D) \implies$
 $T \sim update-conflicting\ (Some\ (ccls-of-cl\ D))\ S \implies$
P S T and
 $forgetH: \bigwedge C\ T. \text{ conflicting } S = None \implies$
 $C !\in! raw-learned-clss\ S \implies$
 $\neg(trail\ S) \models_{asm} clauses\ S \implies$
 $mset-cl\ C \notin set\ (get-all-mark-of-propagated\ (trail\ S)) \implies$
 $mset-cl\ C \notin \# init-clss\ S \implies$
 $T \sim remove-cl\ C\ S \implies$
P S T and
 $restartH: \bigwedge T. \neg trail\ S \models_{asm} clauses\ S \implies$
 $\text{ conflicting } S = None \implies$
 $T \sim restart-state\ S \implies$
P S T and
 $decideH: \bigwedge L\ T. \text{ conflicting } S = None \implies$
 $undefined-lit\ (trail\ S)\ L \implies$
 $atm-of\ L \in atms-of-mm\ (init-clss\ S) \implies$
 $T \sim cons-trail\ (Marked\ L\ (backtrack-lvl\ S + 1))\ (incr-lvl\ S) \implies$
P S T and
 $skipH: \bigwedge L\ C'\ M\ E\ T.$
 $trail\ S = Propagated\ L\ C'\ \# M \implies$
 $raw-conflicting\ S = Some\ E \implies$
 $-L \notin \# mset-ccls\ E \implies mset-ccls\ E \neq \{\#\} \implies$
 $T \sim tl-trail\ S \implies$
P S T and
 $resolveH: \bigwedge L\ E\ M\ D\ T.$
 $trail\ S = Propagated\ L\ (mset-cl\ E)\ \# M \implies$
 $L \in \# mset-cl\ E \implies$
 $hd-raw-trail\ S = Propagated\ L\ E \implies$
 $raw-conflicting\ S = Some\ D \implies$
 $-L \in \# mset-ccls\ D \implies$
 $get-maximum-level\ (trail\ S)\ (mset-ccls\ (remove-clit\ (-L)\ D)) = backtrack-lvl\ S \implies$
 $T \sim update-conflicting$
 $(Some\ (union-ccls\ (remove-clit\ (-L)\ D)\ (ccls-of-cl\ (remove-lit\ L\ E))))\ (tl-trail\ S) \implies$
P S T and
 $backtrackH: \bigwedge K\ i\ M1\ M2\ L\ D\ T.$
 $raw-conflicting\ S = Some\ D \implies$
 $L \in \# mset-ccls\ D \implies$
 $(Marked\ K\ (Suc\ i)\ \# M1, M2) \in set\ (get-all-marked-decomposition\ (trail\ S)) \implies$
 $get-level\ (trail\ S)\ L = backtrack-lvl\ S \implies$
 $get-level\ (trail\ S)\ L = get-maximum-level\ (trail\ S)\ (mset-ccls\ D) \implies$
 $get-maximum-level\ (trail\ S)\ (remove1-mset\ L\ (mset-ccls\ D)) \equiv i \implies$
 $undefined-lit\ M1\ L \implies$
 $T \sim cons-trail\ (Propagated\ L\ (cls-of-ccls\ D))$
 $(reduce-trail-to\ M1$
 $(add-learned-cl\ (cls-of-ccls\ D)$
 $(update-backtrack-lvl\ i$
 $(update-conflicting\ None\ S)))) \implies$
P S T

shows $P S S'$

$\langle \text{proof} \rangle$

lemmas $cdcl_W\text{-all-induct-lev2} = cdcl_W\text{-all-induct-lev-full}[\text{consumes } 2, \text{case-names propagate conflict forget restart decide skip resolve backtrack}]$

lemmas $cdcl_W\text{-all-induct-lev} = cdcl_W\text{-all-induct-lev-full}[\text{consumes } 1, \text{case-names lev-inv propagate conflict forget restart decide skip resolve backtrack}]$

thm $cdcl_W\text{-o-induct}$

lemma $cdcl_W\text{-o-induct-lev}[\text{consumes } 1, \text{case-names } M\text{-lev decide skip resolve backtrack}]$:

fixes $S :: 'st$

assumes

$cdcl_W$: $cdcl_W\text{-o } S T$ **and**

$inv[simp]$: $cdcl_W\text{-M-level-inv } S$ **and**

$decideH$: $\bigwedge L T. \text{conflicting } S = \text{None} \implies$

$\text{undefined-lit } (\text{trail } S) L \implies$

$\text{atm-of } L \in \text{atms-of-mm } (\text{init-clss } S) \implies$

$T \sim \text{cons-trail } (\text{Marked } L (\text{backtrack-lvl } S + 1)) (\text{incr-lvl } S) \implies$

$P S T$ **and**

$skipH$: $\bigwedge L C' M E T.$

$\text{trail } S = \text{Propagated } L C' \# M \implies$

$\text{raw-conflicting } S = \text{Some } E \implies$

$-L \notin \# \text{mset-ccls } E \implies \text{mset-ccls } E \neq \{\#\} \implies$

$T \sim \text{tl-trail } S \implies$

$P S T$ **and**

$resolveH$: $\bigwedge L E M D T.$

$\text{trail } S = \text{Propagated } L (\text{mset-clss } E) \# M \implies$

$L \in \# \text{mset-clss } E \implies$

$\text{hd-raw-trail } S = \text{Propagated } L E \implies$

$\text{raw-conflicting } S = \text{Some } D \implies$

$-L \in \# \text{mset-ccls } D \implies$

$\text{get-maximum-level } (\text{trail } S) (\text{mset-ccls } (\text{remove-clit } (-L) D)) = \text{backtrack-lvl } S \implies$

$T \sim \text{update-conflicting}$

$(\text{Some } (\text{union-ccls } (\text{remove-clit } (-L) D) (\text{ccls-of-clss } (\text{remove-lit } L E)))) (\text{tl-trail } S) \implies$

$P S T$ **and**

backtrackH : $\bigwedge K i M1 M2 L D T.$

$\text{raw-conflicting } S = \text{Some } D \implies$

$L \in \# \text{mset-ccls } D \implies$

$(\text{Marked } K (\text{Suc } i) \# M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S)) \implies$

$\text{get-level } (\text{trail } S) L = \text{backtrack-lvl } S \implies$

$\text{get-level } (\text{trail } S) L = \text{get-maximum-level } (\text{trail } S) (\text{mset-ccls } D) \implies$

$\text{get-maximum-level } (\text{trail } S) (\text{remove1-mset } L (\text{mset-ccls } D)) \equiv i \implies$

$\text{undefined-lit } M1 L \implies$

$T \sim \text{cons-trail } (\text{Propagated } L (\text{cls-of-ccls } D))$

$(\text{reduce-trail-to } M1$

$(\text{add-learned-clss } (\text{cls-of-ccls } D)$

$(\text{update-backtrack-lvl } i$

$(\text{update-conflicting } \text{None } S)))) \implies$

$P S T$

shows $P S T$

$\langle \text{proof} \rangle$

lemmas $cdcl_W\text{-o-induct-lev2} = cdcl_W\text{-o-induct-lev}[\text{consumes } 2, \text{case-names decide skip resolve backtrack}]$

19.3.3 Compatibility with $op \sim$

lemma *propagate-state-eq-compatible:*

assumes

propa: *propagate* S T **and**

SS' : $S \sim S'$ **and**

TT' : $T \sim T'$

shows *propagate* S' T'

$\langle proof \rangle$

lemma *conflict-state-eq-compatible:*

assumes

conft: *conflict* S T **and**

TT' : $T \sim T'$ **and**

SS' : $S \sim S'$

shows *conflict* S' T'

$\langle proof \rangle$

lemma *backtrack-levE[consumes 2]:*

backtrack S $S' \implies cdcl_W$ -*M-level-inv* $S \implies$

$(\bigwedge K\ i\ M1\ M2\ L\ D.$

raw-conflicting $S = \text{Some } D \implies$

$L \in \# \text{ mset-ccls } D \implies$

$(\text{Marked } K\ (\text{Suc } i) \ \# \ M1, M2) \in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S)) \implies$

get-level $(\text{trail } S)\ L = \text{backtrack-lvl } S \implies$

get-level $(\text{trail } S)\ L = \text{get-maximum-level } (\text{trail } S)\ (\text{mset-ccls } D) \implies$

get-maximum-level $(\text{trail } S)\ (\text{remove1-mset } L\ (\text{mset-ccls } D)) \equiv i \implies$

undefined-lit $M1\ L \implies$

$S' \sim \text{cons-trail } (\text{Propagated } L\ (\text{cls-of-ccls } D))$

$(\text{reduce-trail-to } M1$

$(\text{add-learned-cls } (\text{cls-of-ccls } D)$

$(\text{update-backtrack-lvl } i$

$(\text{update-conflicting } \text{None } S)))) \implies P) \implies$

P

$\langle proof \rangle$

thm *allI*

lemma *backtrack-state-eq-compatible:*

assumes

bt: *backtrack* S T **and**

SS' : $S \sim S'$ **and**

TT' : $T \sim T'$ **and**

inv: *cdcl_W-M-level-inv* S

shows *backtrack* S' T'

$\langle proof \rangle$

lemma *decide-state-eq-compatible:*

assumes

decide S T **and**

$S \sim S'$ **and**

$T \sim T'$

shows *decide* S' T'

$\langle proof \rangle$

lemma *skip-state-eq-compatible:*

assumes
skip: *skip S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$
shows *skip S' T'*
 $\langle proof \rangle$

lemma *resolve-state-eq-compatible*:
assumes
res: *resolve S T* **and**
TT': $T \sim T'$ **and**
SS': $S \sim S'$
shows *resolve S' T'*
 $\langle proof \rangle$

lemma *forget-state-eq-compatible*:
assumes
forget: *forget S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$
shows *forget S' T'*
 $\langle proof \rangle$

lemma *cdcl_W-state-eq-compatible*:
assumes
cdcl_W S T **and** $\neg \text{restart } S \text{ } T$ **and**
 $S \sim S'$
 $T \sim T'$ **and**
cdcl_W-M-level-inv S
shows *cdcl_W S' T'*
 $\langle proof \rangle$

lemma *cdcl_W-bj-state-eq-compatible*:
assumes
cdcl_W-bj S T **and** *cdcl_W-M-level-inv S*
 $T \sim T'$
shows *cdcl_W-bj S T'*
 $\langle proof \rangle$

lemma *trancp-cdcl_W-bj-state-eq-compatible*:
assumes
cdcl_W-bj⁺⁺ S T **and** *inv: cdcl_W-M-level-inv S* **and**
 $S \sim S'$ **and**
 $T \sim T'$
shows *cdcl_W-bj⁺⁺ S' T'*
 $\langle proof \rangle$

19.3.4 Conservation of some Properties

lemma *cdcl_W-o-no-more-init-clss*:
assumes
cdcl_W-o S S' **and**
inv: cdcl_W-M-level-inv S
shows *init-clss S = init-clss S'*
 $\langle proof \rangle$

lemma *trancpl-cdcl_W-o-no-more-init-clss:*

assumes

cdcl_W-o⁺⁺ S S' and

inv: cdcl_W-M-level-inv S

shows *init-clss S = init-clss S'*

<proof>

lemma *rtrancpl-cdcl_W-o-no-more-init-clss:*

assumes

*cdcl_W-o^{**} S S' and*

inv: cdcl_W-M-level-inv S

shows *init-clss S = init-clss S'*

<proof>

lemma *cdcl_W-init-clss:*

assumes

cdcl_W S T and

inv: cdcl_W-M-level-inv S

shows *init-clss S = init-clss T*

<proof>

lemma *rtrancpl-cdcl_W-init-clss:*

*cdcl_W^{**} S T \implies cdcl_W-M-level-inv S \implies init-clss S = init-clss T*

<proof>

lemma *trancpl-cdcl_W-init-clss:*

cdcl_W⁺⁺ S T \implies cdcl_W-M-level-inv S \implies init-clss S = init-clss T

<proof>

19.3.5 Learned Clause

This invariant shows that:

- the learned clauses are entailed by the initial set of clauses.
- the conflicting clause is entailed by the initial set of clauses.
- the marks are entailed by the clauses. A more precise version would be to show that either these marked are learned or are in the set of clauses

definition *cdcl_W-learned-clause (S :: 'st) \longleftrightarrow*

(init-clss S \models_{psm} learned-clss S

$\wedge (\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{init-clss } S \models_{pm} T)$

$\wedge \text{set } (\text{get-all-mark-of-propagated } (\text{trail } S)) \subseteq \text{set-mset } (\text{clauses } S))$

lemma *cdcl_W-learned-clause-S0-cdcl_W[simp]:*

cdcl_W-learned-clause (init-state N)

<proof>

lemma *cdcl_W-learned-clss:*

assumes

cdcl_W S S' and

learned: cdcl_W-learned-clause S and

lev-inv: cdcl_W-M-level-inv S

shows $cdcl_W$ -learned-clause S'
 $\langle proof \rangle$

lemma $rtrancp$ - $cdcl_W$ -learned-clss:

assumes
 $cdcl_W^{**} S S'$ **and**
 $cdcl_W$ - M -level-inv S
 $cdcl_W$ -learned-clause S
shows $cdcl_W$ -learned-clause S'
 $\langle proof \rangle$

19.3.6 No alien atom in the state

This invariant means that all the literals are in the set of clauses.

definition $no\text{-}strange\text{-}atm S' \longleftrightarrow$ (
 $(\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S'))$
 $\wedge (\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S') \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-mm } (\text{init-clss } S'))$
 $\wedge \text{atms-of-mm } (\text{learned-clss } S') \subseteq \text{atms-of-mm } (\text{init-clss } S')$
 $\wedge \text{atm-of } ' (\text{lits-of-l } (\text{trail } S')) \subseteq \text{atms-of-mm } (\text{init-clss } S'))$

lemma $no\text{-}strange\text{-}atm\text{-}decomp$:

assumes $no\text{-}strange\text{-}atm S$
shows $\text{conflicting } S = \text{Some } T \implies \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S)$
and $(\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-mm } (\text{init-clss } S))$
and $\text{atms-of-mm } (\text{learned-clss } S) \subseteq \text{atms-of-mm } (\text{init-clss } S)$
and $\text{atm-of } ' (\text{lits-of-l } (\text{trail } S)) \subseteq \text{atms-of-mm } (\text{init-clss } S)$
 $\langle proof \rangle$

lemma $no\text{-}strange\text{-}atm\text{-}S0$ [simp]: $no\text{-}strange\text{-}atm (\text{init-state } N)$
 $\langle proof \rangle$

lemma $in\text{-}atms\text{-}of\text{-}implies\text{-}atm\text{-}of\text{-}on\text{-}atms\text{-}of\text{-}ms$:

$C + \{\#L\# \} \in \# A \implies x \in \text{atms-of } C \implies x \in \text{atms-of-mm } A$
 $\langle proof \rangle$

lemma $propagate\text{-}no\text{-}strange\text{-}atm\text{-}inv$:

assumes
 $propagate S T$ **and**
 $alien: no\text{-}strange\text{-}atm S$
shows $no\text{-}strange\text{-}atm T$
 $\langle proof \rangle$

lemma $in\text{-}atms\text{-}of\text{-}remove1\text{-}mset\text{-}in\text{-}atms\text{-}of$:

$x \in \text{atms-of } (\text{remove1-mset } L C) \implies x \in \text{atms-of } C$
 $\langle proof \rangle$

lemma $cdcl_W$ - $no\text{-}strange\text{-}atm\text{-}explicit$:

assumes
 $cdcl_W S S'$ **and**
 $lev: cdcl_W$ - M -level-inv S **and**
 $conf: \forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm } (\text{init-clss } S)$ **and**
 $marked: \forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{atms-of mark} \subseteq \text{atms-of-mm } (\text{init-clss } S)$ **and**

learned: $\text{atms-of-mm} (\text{learned-clss } S) \subseteq \text{atms-of-mm} (\text{init-clss } S)$ **and**
trail: $\text{atm-of } ' (\text{lits-of-l} (\text{trail } S)) \subseteq \text{atms-of-mm} (\text{init-clss } S)$

shows

$(\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{atms-of } T \subseteq \text{atms-of-mm} (\text{init-clss } S')) \wedge$
 $(\forall L \text{ mark. } \text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S') \longrightarrow \text{atms-of } (\text{mark}) \subseteq \text{atms-of-mm} (\text{init-clss } S')) \wedge$
 $\text{atms-of-mm} (\text{learned-clss } S') \subseteq \text{atms-of-mm} (\text{init-clss } S') \wedge$
 $\text{atm-of } ' (\text{lits-of-l} (\text{trail } S')) \subseteq \text{atms-of-mm} (\text{init-clss } S')$
 $(\text{is } ?C S' \wedge ?M S' \wedge ?U S' \wedge ?V S')$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-no-strange-atm-inv*:

assumes *cdcl_W S S'* **and** *no-strange-atm S* **and** *cdcl_W-M-level-inv S*

shows *no-strange-atm S'*

$\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-no-strange-atm-inv*:

assumes *cdcl_W** S S'* **and** *no-strange-atm S* **and** *cdcl_W-M-level-inv S*

shows *no-strange-atm S'*

$\langle \text{proof} \rangle$

19.3.7 No duplicates all around

This invariant shows that there is no duplicate (no literal appearing twice in the formula). The last part could be proven using the previous invariant moreover.

definition *distinct-cdcl_W-state (S :: 'st)*

$\longleftrightarrow ((\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{distinct-mset } T)$
 $\wedge \text{distinct-mset-mset} (\text{learned-clss } S)$
 $\wedge \text{distinct-mset-mset} (\text{init-clss } S)$
 $\wedge (\forall L \text{ mark. } (\text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset } (\text{mark}))))$

lemma *distinct-cdcl_W-state-decomp*:

assumes *distinct-cdcl_W-state (S :: 'st)*

shows $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{distinct-mset } T$

and *distinct-mset-mset (learned-clss S)*

and *distinct-mset-mset (init-clss S)*

and $\forall L \text{ mark. } (\text{Propagated } L \text{ mark} \in \text{set } (\text{trail } S) \longrightarrow \text{distinct-mset } (\text{mark}))$

$\langle \text{proof} \rangle$

lemma *distinct-cdcl_W-state-decomp-2*:

assumes *distinct-cdcl_W-state (S :: 'st)*

shows $\text{conflicting } S = \text{Some } T \implies \text{distinct-mset } T$

$\langle \text{proof} \rangle$

lemma *distinct-cdcl_W-state-S0-cdcl_W[simp]*:

distinct-mset-mset (mset-clss N) \implies distinct-cdcl_W-state (init-state N)

$\langle \text{proof} \rangle$

lemma *distinct-cdcl_W-state-inv*:

assumes

cdcl_W S S' **and**

lev-inv: cdcl_W-M-level-inv S **and**

distinct-cdcl_W-state S

shows *distinct-cdcl_W-state S'*

$\langle \text{proof} \rangle$

lemma *rtanclp-distinct-cdcl_W-state-inv*:

assumes

*cdcl_W** S S' and*

cdcl_W-M-level-inv S and

distinct-cdcl_W-state S

shows *distinct-cdcl_W-state S'*

<proof>

19.3.8 Conflicts and co

This invariant shows that each mark contains a contradiction only related to the previously defined variable.

abbreviation *every-mark-is-a-conflict* :: 'st \Rightarrow bool **where**

every-mark-is-a-conflict S \equiv

$\forall L \text{ mark } a \ b. \ a \ @ \ \text{Propagated } L \text{ mark} \ \# \ b = (\text{trail } S)$

$\longrightarrow (b \models_{as} \text{CNot } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$

definition *cdcl_W-conflicting S* \equiv

$(\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T)$

$\wedge \text{every-mark-is-a-conflict } S$

lemma *backtrack-atms-of-D-in-M1*:

fixes *M1* :: ('v, nat, 'v clause) marked-lits

assumes

inv: cdcl_W-M-level-inv S and

undef: undefined-lit M1 L and

i: get-maximum-level (trail S) (mset-ccls (remove-clit L D)) \equiv i and

decomp: (Marked K (Suc i) $\#$ M1, M2)

$\in \text{set } (\text{get-all-marked-decomposition } (\text{trail } S))$ and

S-lvl: backtrack-lvl S = get-maximum-level (trail S) (mset-ccls D) and

S-conf: raw-conflicting S = Some D and

undef: undefined-lit M1 L and

T: T \sim cons-trail (Propagated L (cls-of-ccls D))

(reduce-trail-to M1

(add-learned-cls (cls-of-ccls D)

(update-backtrack-lvl i

(update-conflicting None S)))) and

conf: $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} \text{CNot } T$

shows *atms-of (mset-ccls (remove-clit L D)) \subseteq atm-of ' lits-of-l (tl (trail T))*

<proof>

lemma *distinct-atms-of-incl-not-in-other*:

assumes

a1: no-dup (M @ M') and

a2: atms-of D \subseteq atm-of ' lits-of-l M' and

a3: x \in atms-of D

shows *x \notin atm-of ' lits-of-l M*

<proof>

lemma *cdcl_W-propagate-is-conclusion*:

assumes

cdcl_W S S' and

inv: cdcl_W-M-level-inv S and

decomp: all-decomposition-implies-m (init-cls S) (get-all-marked-decomposition (trail S)) and

learned: *cdcl_W-learned-clause 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*
shows *all-decomposition-implies-m (init-clss S') (get-all-marked-decomposition (trail S'))*
 <proof>

lemma *cdcl_W-propagate-is-false*:

assumes
cdcl_W S S' and
lev: *cdcl_W-M-level-inv S* **and**
learned: *cdcl_W-learned-clause S* **and**
decomp: *all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))* **and**
confl: $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot \ T$ **and**
alien: *no-strange-atm S* **and**
mark-confl: *every-mark-is-a-conflict S*
shows *every-mark-is-a-conflict S'*
 <proof>

lemma *cdcl_W-conflicting-is-false*:

assumes
cdcl_W S S' and
M-lev: *cdcl_W-M-level-inv S* **and**
confl-inv: $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot \ T$ **and**
marked-confl: $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S) \longrightarrow (b \models_{as} CNot \ (\text{mark} - \{\#L\}) \wedge L \in \# \ \text{mark})$ **and**
dist: *distinct-cdcl_W-state S*
shows $\forall T. \text{conflicting } S' = \text{Some } T \longrightarrow \text{trail } S' \models_{as} CNot \ T$
 <proof>

lemma *cdcl_W-conflicting-decomp*:

assumes *cdcl_W-conflicting S*
shows $\forall T. \text{conflicting } S = \text{Some } T \longrightarrow \text{trail } S \models_{as} CNot \ T$
and $\forall L \text{ mark } a \ b. a @ \text{Propagated } L \text{ mark } \# \ b = (\text{trail } S) \longrightarrow (b \models_{as} CNot \ (\text{mark} - \{\#L\}) \wedge L \in \# \ \text{mark})$
 <proof>

lemma *cdcl_W-conflicting-decomp2*:

assumes *cdcl_W-conflicting S* **and** *conflicting S = Some T*
shows *trail S \models_{as} CNot T*
 <proof>

lemma *cdcl_W-conflicting-S0-cdcl_W[simp]*:

cdcl_W-conflicting (init-state N)
 <proof>

19.3.9 Putting all the invariants together

lemma *cdcl_W-all-inv*:

assumes
cdcl_W: cdcl_W S S' and
1: *all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))* **and**
2: *cdcl_W-learned-clause S* **and**
4: *cdcl_W-M-level-inv S* **and**
5: *no-strange-atm S* **and**
7: *distinct-cdcl_W-state S* **and**
8: *cdcl_W-conflicting S*

shows

all-decomposition-implies-m (*init-clss* S') (*get-all-marked-decomposition* (*trail* S')) **and**
cdcl_W-learned-clause S' **and**
cdcl_W-M-level-inv S' **and**
no-strange-atm S' **and**
distinct-cdcl_W-state S' **and**
cdcl_W-conflicting S'

\langle *proof* \rangle

lemma *rtrancp-cdcl_W-all-inv*:

assumes

cdcl_W: *rtrancp cdcl_W S S'* **and**
1: *all-decomposition-implies-m* (*init-clss* S) (*get-all-marked-decomposition* (*trail* S)) **and**
2: *cdcl_W-learned-clause* S **and**
4: *cdcl_W-M-level-inv* S **and**
5: *no-strange-atm* S **and**
7: *distinct-cdcl_W-state* S **and**
8: *cdcl_W-conflicting* S

shows

all-decomposition-implies-m (*init-clss* S') (*get-all-marked-decomposition* (*trail* S')) **and**
cdcl_W-learned-clause S' **and**
cdcl_W-M-level-inv S' **and**
no-strange-atm S' **and**
distinct-cdcl_W-state S' **and**
cdcl_W-conflicting S'

\langle *proof* \rangle

lemma *all-invariant-S0-cdcl_W*:

assumes *distinct-mset-mset* (*mset-clss* N)

shows

all-decomposition-implies-m (*init-clss* (*init-state* N))
(*get-all-marked-decomposition* (*trail* (*init-state* N))) **and**
cdcl_W-learned-clause (*init-state* N) **and**
 $\forall T.$ *conflicting* (*init-state* N) = *Some* $T \longrightarrow$ (*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**
 $\forall L$ *mark* a . $a @$ *Propagated* L *mark* $\# b =$ *trail* (*init-state* N) \longrightarrow
 $(b \models_{as} \text{CNot } (\text{mark} - \{\#L\}) \wedge L \in \# \text{ mark})$ **and**
distinct-cdcl_W-state (*init-state* N)

\langle *proof* \rangle

lemma *cdcl_W-only-propagated-vars-unsat*:

assumes

marked: $\forall x \in \text{set } M. \neg \text{is-marked } x$ **and**
DN: $D \in \# \text{ clauses } S$ **and**
D: $M \models_{as} \text{CNot } D$ **and**
inv: *all-decomposition-implies-m* N (*get-all-marked-decomposition* M) **and**
state: *state* $S = (M, N, U, k, C)$ **and**
learned-cl: *cdcl_W-learned-clause* S **and**
atm-incl: *no-strange-atm* S

shows *unsatisfiable* (*set-mset* N)

\langle *proof* \rangle

We have actually a much stronger theorem, namely *all-decomposition-implies ?N* (*get-all-marked-decomposition* $?M$) \implies $?N \cup \{\text{unmark } L \mid L. \text{is-marked } L \wedge L \in \text{set } ?M\} \models_{ps} \text{unmark-l } ?M$, that show that

the only choices we made are marked in the formula

lemma

assumes *all-decomposition-implies-m* N (*get-all-marked-decomposition* M)
and $\forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\text{set-mset } N \models_{ps} \text{unmark-l } M$
 $\langle \text{proof} \rangle$

lemma *conflict-with-false-implies-unsat*:

assumes
 $\text{cdcl}_W: \text{cdcl}_W \ S \ S'$ **and**
 $\text{lev}: \text{cdcl}_W\text{-}M\text{-level-inv } S$ **and**
 $[\text{simp}]: \text{conflicting } S' = \text{Some } \{\#\}$ **and**
 $\text{learned}: \text{cdcl}_W\text{-learned-clause } S$
shows $\text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$
 $\langle \text{proof} \rangle$

lemma *conflict-with-false-implies-terminated*:

assumes $\text{cdcl}_W \ S \ S'$
and $\text{conflicting } S = \text{Some } \{\#\}$
shows False
 $\langle \text{proof} \rangle$

19.3.10 No tautology is learned

This is a simple consequence of all we have shown previously. It is not strictly necessary, but helps finding a better bound on the number of learned clauses.

lemma *learned-clss-are-not-tautologies*:

assumes
 $\text{cdcl}_W \ S \ S'$ **and**
 $\text{lev}: \text{cdcl}_W\text{-}M\text{-level-inv } S$ **and**
 $\text{conflicting}: \text{cdcl}_W\text{-conflicting } S$ **and**
 $\text{no-tauto}: \forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$
shows $\forall s \in \# \text{ learned-clss } S'. \neg \text{tautology } s$
 $\langle \text{proof} \rangle$

definition *final-cdcl_W-state* ($S :: 'st$)

$\longleftrightarrow (\text{trail } S \models_{asm} \text{init-clss } S$
 $\vee ((\forall L \in \text{set } (\text{trail } S). \neg \text{is-marked } L) \wedge$
 $(\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} \text{CNot } C)))$

definition *termination-cdcl_W-state* ($S :: 'st$)

$\longleftrightarrow (\text{trail } S \models_{asm} \text{init-clss } S$
 $\vee ((\forall L \in \text{atms-of-mm } (\text{init-clss } S). L \in \text{atm-of ' lits-of-l } (\text{trail } S))$
 $\wedge (\exists C \in \# \text{ init-clss } S. \text{trail } S \models_{as} \text{CNot } C)))$

19.4 CDCL Strong Completeness

fun *mapi* :: ($'a \Rightarrow \text{nat} \Rightarrow 'b \Rightarrow \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list}$ **where**

mapi - - $\square = \square \mid$

mapi $f \ n \ (x \ \# \ xs) = f \ x \ n \ \# \ \text{mapi } f \ (n - 1) \ xs$

lemma *mark-not-in-set-mapi*[*simp*]: $L \notin \text{set } M \implies \text{Marked } L \ k \notin \text{set } (\text{mapi } \text{Marked } i \ M)$

$\langle \text{proof} \rangle$

lemma *propagated-not-in-set-mapi[simp]*: $L \notin \text{set } M \implies \text{Propagated } L \notin \text{set } (\text{mapi Marked } i \ M)$
 <proof>

lemma *image-set-mapi*:
 $f \text{ ' set } (\text{mapi } g \ i \ M) = \text{set } (\text{mapi } (\lambda x \ i. f \ (g \ x \ i)) \ i \ M)$
 <proof>

lemma *mapi-map-convert*:
 $\forall x \ i \ j. f \ x \ i = f \ x \ j \implies \text{mapi } f \ i \ M = \text{map } (\lambda x. f \ x \ 0) \ M$
 <proof>

lemma *defined-lit-mapi*: $\text{defined-lit } (\text{mapi Marked } i \ M) \ L \longleftrightarrow \text{atm-of } L \in \text{atm-of ' set } M$
 <proof>

lemma *cdcl_W-can-do-step*:
assumes
 consistent-interp (set M) **and**
 distinct M **and**
 $\text{atm-of ' (set } M) \subseteq \text{atms-of-mm (mset-clss } N)$
shows $\exists S. \text{rtrancplp cdcl}_W \ (\text{init-state } N) \ S$
 $\wedge \text{state } S = (\text{mapi Marked (length } M) \ M, \text{mset-clss } N, \{\#\}, \text{length } M, \text{None})$
 <proof>

lemma *cdcl_W-strong-completeness*:
assumes
 MN: $\text{set } M \models_{sm} \text{mset-clss } N$ **and**
 cons: *consistent-interp* (set M) **and**
 dist: *distinct* M **and**
 atm: $\text{atm-of ' (set } M) \subseteq \text{atms-of-mm (mset-clss } N)$
obtains *S* **where**
 $\text{state } S = (\text{mapi Marked (length } M) \ M, \text{mset-clss } N, \{\#\}, \text{length } M, \text{None})$ **and**
 rtrancplp cdcl_W (init-state N) S **and**
 final-cdcl_W-state S
 <proof>

19.5 Higher level strategy

The rules described previously do not lead to a conclusive state. We have to add a strategy.

19.5.1 Definition

lemma *trancplp-conflict*:
 $\text{trancplp conflict } S \ S' \implies \text{conflict } S \ S'$
 <proof>

lemma *trancplp-conflict-iff[iff]*:
 $\text{full1 conflict } S \ S' \longleftrightarrow \text{conflict } S \ S'$
 <proof>

inductive *cdcl_W-cp* :: 'st \Rightarrow 'st \Rightarrow bool **where**
conflict'[intro]: $\text{conflict } S \ S' \implies \text{cdcl}_W\text{-cp } S \ S' \mid$
propagate': $\text{propagate } S \ S' \implies \text{cdcl}_W\text{-cp } S \ S'$

lemma *rtrancplp-cdcl_W-cp-rtrancplp-cdcl_W*:
 $\text{cdcl}_W\text{-cp}^{**} \ S \ T \implies \text{cdcl}_W^{**} \ S \ T$

$\langle \text{proof} \rangle$

lemma *cdcl_W-cp-state-eq-compatible:*

assumes

cdcl_W-cp S T and

S ~ S' and

T ~ T'

shows *cdcl_W-cp S' T'*

$\langle \text{proof} \rangle$

lemma *trancpl-cdcl_W-cp-state-eq-compatible:*

assumes

cdcl_W-cp⁺⁺ S T and

S ~ S' and

T ~ T'

shows *cdcl_W-cp⁺⁺ S' T'*

$\langle \text{proof} \rangle$

lemma *option-full-cdcl_W-cp:*

conflicting S ≠ None ⇒ full cdcl_W-cp S S

$\langle \text{proof} \rangle$

lemma *skip-unique:*

skip S T ⇒ skip S T' ⇒ T ~ T'

$\langle \text{proof} \rangle$

lemma *resolve-unique:*

resolve S T ⇒ resolve S T' ⇒ T ~ T'

$\langle \text{proof} \rangle$

lemma *cdcl_W-cp-no-more-clauses:*

assumes *cdcl_W-cp S S'*

shows *clauses S = clauses S'*

$\langle \text{proof} \rangle$

lemma *trancpl-cdcl_W-cp-no-more-clauses:*

assumes *cdcl_W-cp⁺⁺ S S'*

shows *clauses S = clauses S'*

$\langle \text{proof} \rangle$

lemma *rtrancpl-cdcl_W-cp-no-more-clauses:*

assumes *cdcl_W-cp^{**} S S'*

shows *clauses S = clauses S'*

$\langle \text{proof} \rangle$

lemma *no-conflict-after-conflict:*

conflict S T ⇒ ¬conflict T U

$\langle \text{proof} \rangle$

lemma *no-propagate-after-conflict:*

conflict S T ⇒ ¬propagate T U

$\langle \text{proof} \rangle$

lemma *trancpl-cdcl_W-cp-propagate-with-conflict-or-not:*

assumes *cdcl_W-cp⁺⁺ S U*

shows $(\text{propagate}^{++} S U \wedge \text{conflicting } U = \text{None})$
 $\vee (\exists T D. \text{propagate}^{**} S T \wedge \text{conflict } T U \wedge \text{conflicting } U = \text{Some } D)$
 $\langle \text{proof} \rangle$

lemma $\text{cdcl}_W\text{-cp-conflicting-not-empty}[\text{simp}]$: $\text{conflicting } S = \text{Some } D \implies \neg \text{cdcl}_W\text{-cp } S S'$
 $\langle \text{proof} \rangle$

lemma $\text{no-step-cdcl}_W\text{-cp-no-conflict-no-propagate}$:
assumes $\text{no-step cdcl}_W\text{-cp } S$
shows $\text{no-step conflict } S$ **and** $\text{no-step propagate } S$
 $\langle \text{proof} \rangle$

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 $\text{cdcl}_W\text{-stgy} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **for** $S :: 'st$ **where**
 $\text{conflict}': \text{full1 cdcl}_W\text{-cp } S S' \implies \text{cdcl}_W\text{-stgy } S S' \mid$
 $\text{other}': \text{cdcl}_W\text{-o } S S' \implies \text{no-step cdcl}_W\text{-cp } S \implies \text{full cdcl}_W\text{-cp } S' S'' \implies \text{cdcl}_W\text{-stgy } S S''$

19.5.2 Invariants

These are the same invariants as before, but lifted

lemma $\text{cdcl}_W\text{-cp-learned-clause-inv}$:
assumes $\text{cdcl}_W\text{-cp } S S'$
shows $\text{learned-clss } S = \text{learned-clss } S'$
 $\langle \text{proof} \rangle$

lemma $\text{rtrancpl-cdcl}_W\text{-cp-learned-clause-inv}$:
assumes $\text{cdcl}_W\text{-cp}^{**} S S'$
shows $\text{learned-clss } S = \text{learned-clss } S'$
 $\langle \text{proof} \rangle$

lemma $\text{trancpl-cdcl}_W\text{-cp-learned-clause-inv}$:
assumes $\text{cdcl}_W\text{-cp}^{++} S S'$
shows $\text{learned-clss } S = \text{learned-clss } S'$
 $\langle \text{proof} \rangle$

lemma $\text{cdcl}_W\text{-cp-backtrack-lvl}$:
assumes $\text{cdcl}_W\text{-cp } S S'$
shows $\text{backtrack-lvl } S = \text{backtrack-lvl } S'$
 $\langle \text{proof} \rangle$

lemma $\text{rtrancpl-cdcl}_W\text{-cp-backtrack-lvl}$:
assumes $\text{cdcl}_W\text{-cp}^{**} S S'$
shows $\text{backtrack-lvl } S = \text{backtrack-lvl } S'$
 $\langle \text{proof} \rangle$

lemma $\text{cdcl}_W\text{-cp-consistent-inv}$:
assumes $\text{cdcl}_W\text{-cp } S S'$
and $\text{cdcl}_W\text{-M-level-inv } S$
shows $\text{cdcl}_W\text{-M-level-inv } S'$
 $\langle \text{proof} \rangle$

lemma $\text{full1-cdcl}_W\text{-cp-consistent-inv}$:
assumes $\text{full1 cdcl}_W\text{-cp } S S'$

and $cdcl_W\text{-}M\text{-level-inv } S$
shows $cdcl_W\text{-}M\text{-level-inv } S'$
 $\langle proof \rangle$

lemma $rtrancpl\text{-}cdcl_W\text{-}cp\text{-consistent-inv$:
assumes $rtrancpl\text{-}cdcl_W\text{-}cp\text{-} S\ S'$
and $cdcl_W\text{-}M\text{-level-inv } S$
shows $cdcl_W\text{-}M\text{-level-inv } S'$
 $\langle proof \rangle$

lemma $cdcl_W\text{-}stgy\text{-consistent-inv$:
assumes $cdcl_W\text{-}stgy\text{-} S\ S'$
and $cdcl_W\text{-}M\text{-level-inv } S$
shows $cdcl_W\text{-}M\text{-level-inv } S'$
 $\langle proof \rangle$

lemma $rtrancpl\text{-}cdcl_W\text{-}stgy\text{-consistent-inv$:
assumes $cdcl_W\text{-}stgy^{**}\text{-} S\ S'$
and $cdcl_W\text{-}M\text{-level-inv } S$
shows $cdcl_W\text{-}M\text{-level-inv } S'$
 $\langle proof \rangle$

lemma $cdcl_W\text{-}cp\text{-no-more-init-clss$:
assumes $cdcl_W\text{-}cp\text{-} S\ S'$
shows $init\text{-clss } S = init\text{-clss } S'$
 $\langle proof \rangle$

lemma $trancpl\text{-}cdcl_W\text{-}cp\text{-no-more-init-clss$:
assumes $cdcl_W\text{-}cp^{++}\text{-} S\ S'$
shows $init\text{-clss } S = init\text{-clss } S'$
 $\langle proof \rangle$

lemma $cdcl_W\text{-}stgy\text{-no-more-init-clss$:
assumes $cdcl_W\text{-}stgy\text{-} S\ S'$ **and** $cdcl_W\text{-}M\text{-level-inv } S$
shows $init\text{-clss } S = init\text{-clss } S'$
 $\langle proof \rangle$

lemma $rtrancpl\text{-}cdcl_W\text{-}stgy\text{-no-more-init-clss$:
assumes $cdcl_W\text{-}stgy^{**}\text{-} S\ S'$ **and** $cdcl_W\text{-}M\text{-level-inv } S$
shows $init\text{-clss } S = init\text{-clss } S'$
 $\langle proof \rangle$

lemma $cdcl_W\text{-}cp\text{-dropWhile-trail'}$:
assumes $cdcl_W\text{-}cp\text{-} S\ S'$
obtains M **where** $trail\text{-} S' = M @ trail\text{-} S$ **and** $(\forall l \in set\text{-} M. \neg is\text{-}marked\text{-} l)$
 $\langle proof \rangle$

lemma $rtrancpl\text{-}cdcl_W\text{-}cp\text{-dropWhile-trail'}$:
assumes $cdcl_W\text{-}cp^{**}\text{-} S\ S'$
obtains $M :: ('v, nat, 'v\text{-}clause)\text{-}marked\text{-}lit\text{-}list$ **where**
 $trail\text{-} S' = M @ trail\text{-} S$ **and** $\forall l \in set\text{-} M. \neg is\text{-}marked\text{-} l$
 $\langle proof \rangle$

lemma $cdcl_W\text{-}cp\text{-dropWhile-trail}$:
assumes $cdcl_W\text{-}cp\text{-} S\ S'$

shows $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-cp-dropWhile-trail*:

assumes *cdcl_W-cp*** $S S'$

shows $\exists M. \text{trail } S' = M @ \text{trail } S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$
 $\langle \text{proof} \rangle$

This theorem can be seen as a termination theorem for *cdcl_W-cp*.

lemma *length-model-le-vars*:

assumes

no-strange-atm S **and**

no-d: *no-dup* (*trail* S) **and**

finite (*atms-of-mm* (*init-clss* S))

shows $\text{length } (\text{trail } S) \leq \text{card } (\text{atms-of-mm } (\text{init-clss } S))$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-decreasing-measure*:

assumes

cdcl_W: *cdcl_W-cp* $S T$ **and**

M-lev: *cdcl_W-M-level-inv* S **and**

alien: *no-strange-atm* S

shows $(\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$
 $+ (\text{if conflicting } S = \text{None then } 1 \text{ else } 0)) S$
 $> (\lambda S. \text{card } (\text{atms-of-mm } (\text{init-clss } S)) - \text{length } (\text{trail } S))$
 $+ (\text{if conflicting } S = \text{None then } 1 \text{ else } 0)) T$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-wf*: $\text{wf } \{(b, a). (\text{cdcl}_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a) \wedge \text{cdcl}_W\text{-cp } a b\}$
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-all-struct-inv-cdcl_W-cp-iff-rtrancp-cdcl_W-cp*:

assumes

lev: *cdcl_W-M-level-inv* S **and**

alien: *no-strange-atm* S

shows $(\lambda a b. (\text{cdcl}_W\text{-M-level-inv } a \wedge \text{no-strange-atm } a) \wedge \text{cdcl}_W\text{-cp } a b)^{**} S T$
 $\longleftrightarrow \text{cdcl}_W\text{-cp}^{**} S T$
 $(\text{is } ?I S T \longleftrightarrow ?C S T)$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-normalized-element*:

assumes

lev: *cdcl_W-M-level-inv* S **and**

no-strange-atm S

obtains T **where** *full cdcl_W-cp* $S T$
 $\langle \text{proof} \rangle$

lemma *always-exists-full-cdcl_W-cp-step*:

assumes *no-strange-atm* S

shows $\exists S''. \text{full cdcl}_W\text{-cp } S S''$
 $\langle \text{proof} \rangle$

19.5.3 Literal of highest level in conflicting clauses

One important property of the $cdcl_W$ with strategy is that, whenever a conflict takes place, there is at least a literal of level k involved (except if we have derived the false clause). The reason is that we apply conflicts before a decision is taken.

abbreviation $no_clause_is_false :: 'st \Rightarrow bool$ **where**

$no_clause_is_false \equiv$

$\lambda S. (conflicting\ S = None \longrightarrow (\forall D \in \# \text{ clauses } S. \neg trail\ S \models_{as} CNot\ D))$

abbreviation $conflict_is_false_with_level :: 'st \Rightarrow bool$ **where**

$conflict_is_false_with_level\ S \equiv \forall D. conflicting\ S = Some\ D \longrightarrow D \neq \{\#\}$
 $\longrightarrow (\exists L \in \# D. get_level\ (trail\ S)\ L = backtrack_lvl\ S)$

lemma $not_conflict_not_any_negated_init_clss$:

assumes $\forall S'. \neg conflict\ S\ S'$

shows $no_clause_is_false\ S$

$\langle proof \rangle$

lemma $full_cdcl_W_cp_not_any_negated_init_clss$:

assumes $full\ cdcl_W_cp\ S\ S'$

shows $no_clause_is_false\ S'$

$\langle proof \rangle$

lemma $full1_cdcl_W_cp_not_any_negated_init_clss$:

assumes $full1\ cdcl_W_cp\ S\ S'$

shows $no_clause_is_false\ S'$

$\langle proof \rangle$

lemma $cdcl_W_stgy_not_non_negated_init_clss$:

assumes $cdcl_W_stgy\ S\ S'$

shows $no_clause_is_false\ S'$

$\langle proof \rangle$

lemma $rtranclp_cdcl_W_stgy_not_non_negated_init_clss$:

assumes $cdcl_W_stgy^{**}\ S\ S'$ **and** $no_clause_is_false\ S$

shows $no_clause_is_false\ S'$

$\langle proof \rangle$

lemma $cdcl_W_stgy_conflict_ex_lit_of_max_level$:

assumes $cdcl_W_cp\ S\ S'$

and $no_clause_is_false\ S$

and $cdcl_W_M_level_inv\ S$

shows $conflict_is_false_with_level\ S'$

$\langle proof \rangle$

lemma $no_chained_conflict$:

assumes $conflict\ S\ S'$

and $conflict\ S'\ S''$

shows $False$

$\langle proof \rangle$

lemma $rtranclp_cdcl_W_cp_propa_or_propa_confl$:

assumes $cdcl_W_cp^{**}\ S\ U$

shows $propagate^{**}\ S\ U \vee (\exists T. propagate^{**}\ S\ T \wedge conflict\ T\ U)$

$\langle proof \rangle$

lemma *rtrancp-cdcl_W-co-conflict-ex-lit-of-max-level:*

assumes *full: full cdcl_W-cp S U*
and *cls-f: no-clause-is-false S*
and *conflict-is-false-with-level S*
and *lev: cdcl_W-M-level-inv S*
shows *conflict-is-false-with-level U*

<proof>

19.5.4 Literal of highest level in marked literals

definition *mark-is-false-with-level :: 'st \Rightarrow bool where*

mark-is-false-with-level S' \equiv

$\forall D M1 M2 L. M1 @ \text{Propagated } L D \# M2 = \text{trail } S' \longrightarrow D - \{\#L\} \neq \{\#\}$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{get-maximum-possible-level } M1)$

definition *no-more-propagation-to-do :: 'st \Rightarrow bool where*

no-more-propagation-to-do S \equiv

$\forall D M M' L. D + \{\#L\} \in \# \text{ clauses } S \longrightarrow \text{trail } S = M' @ M \longrightarrow M \models_{as} CNot D$
 $\longrightarrow \text{undefined-lit } M L \longrightarrow \text{get-maximum-possible-level } M < \text{backtrack-lvl } S$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S) L = \text{get-maximum-possible-level } M)$

lemma *propagate-no-more-propagation-to-do:*

assumes *propagate: propagate S S'*
and *H: no-more-propagation-to-do S*
and *lev-inv: cdcl_W-M-level-inv S*
shows *no-more-propagation-to-do S'*
<proof>

lemma *conflict-no-more-propagation-to-do:*

assumes
conflict: conflict S S' and
H: no-more-propagation-to-do S and
M: cdcl_W-M-level-inv S
shows *no-more-propagation-to-do S'*
<proof>

lemma *cdcl_W-cp-no-more-propagation-to-do:*

assumes
conflict: cdcl_W-cp S S' and
H: no-more-propagation-to-do S and
M: cdcl_W-M-level-inv S
shows *no-more-propagation-to-do S'*
<proof>

lemma *cdcl_W-then-exists-cdcl_W-stgy-step:*

assumes
o: cdcl_W-o S S' and
alien: no-strange-atm S and
lev: cdcl_W-M-level-inv S
shows $\exists S'. \text{cdcl}_W\text{-stgy } S S'$
<proof>

lemma *backtrack-no-decomp:*

assumes
S: raw-conflicting S = Some E and

LE: $L \in \#$ *mset-ccls* E **and**
L: *get-level* (*trail* S) $L = \text{backtrack-lvl } S$ **and**
D: *get-maximum-level* (*trail* S) (*remove1-mset* L (*mset-ccls* E)) $< \text{backtrack-lvl } S$ **and**
bt: *backtrack-lvl* $S = \text{get-maximum-level}$ (*trail* S) (*mset-ccls* E) **and**
M-L: *cdcl_W-M-level-inv* S
shows $\exists S'. \text{cdcl}_W\text{-o } S S'$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-final-state-conclusive*:
assumes
termi: $\forall S'. \neg \text{cdcl}_W\text{-stgy } S S'$ **and**
decomp: *all-decomposition-implies-m* (*init-clss* S) (*get-all-marked-decomposition* (*trail* S)) **and**
learned: *cdcl_W-learned-clause* S **and**
level-inv: *cdcl_W-M-level-inv* S **and**
alien: *no-strange-atm* S **and**
no-dup: *distinct-cdcl_W-state* S **and**
conf: *cdcl_W-conflicting* S **and**
conf-k: *conflict-is-false-with-level* S
shows (*conflicting* $S = \text{Some } \{\#\} \wedge \text{unsatisfiable } (\text{set-mset } (\text{init-clss } S))$)
 $\vee (\text{conflicting } S = \text{None} \wedge \text{trail } S \models_{\text{as}} \text{set-mset } (\text{init-clss } S))$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-tranclp-cdcl_W*:
 $\text{cdcl}_W\text{-cp } S S' \implies \text{cdcl}_W^{++} S S'$
 $\langle \text{proof} \rangle$

lemma *tranclp-cdcl_W-cp-tranclp-cdcl_W*:
 $\text{cdcl}_W\text{-cp}^{++} S S' \implies \text{cdcl}_W^{++} S S'$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-tranclp-cdcl_W*:
 $\text{cdcl}_W\text{-stgy } S S' \implies \text{cdcl}_W^{++} S S'$
 $\langle \text{proof} \rangle$

lemma *tranclp-cdcl_W-stgy-tranclp-cdcl_W*:
 $\text{cdcl}_W\text{-stgy}^{++} S S' \implies \text{cdcl}_W^{++} S S'$
 $\langle \text{proof} \rangle$

lemma *rtranclp-cdcl_W-stgy-rtranclp-cdcl_W*:
 $\text{cdcl}_W\text{-stgy}^{**} S S' \implies \text{cdcl}_W^{**} S S'$
 $\langle \text{proof} \rangle$

lemma *not-empty-get-maximum-level-exists-lit*:
assumes $n: D \neq \{\#\}$
and *max*: *get-maximum-level* $M D = n$
shows $\exists L \in \#D. \text{get-level } M L = n$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-o-conflict-is-false-with-level-inv*:
assumes
 $\text{cdcl}_W\text{-o } S S'$ **and**
lev: *cdcl_W-M-level-inv* S **and**
conf-inv: *conflict-is-false-with-level* S **and**
n-d: *distinct-cdcl_W-state* S **and**
conflicting: *cdcl_W-conflicting* S

shows *conflict-is-false-with-level* S'
 $\langle \text{proof} \rangle$

19.5.5 Strong completeness

lemma *cdcl_W-cp-propagate-conflict*:

assumes *cdcl_W-cp* S T
shows *propagate*** S $T \vee (\exists S'. \text{propagate** } S S' \wedge \text{conflict } S' T)$
 $\langle \text{proof} \rangle$

lemma *rtrancpl-cdcl_W-cp-propagate-conflict*:

assumes *cdcl_W-cp*** S T
shows *propagate*** S $T \vee (\exists S'. \text{propagate** } S S' \wedge \text{conflict } S' T)$
 $\langle \text{proof} \rangle$

lemma *propagate-high-levelE*:

assumes *propagate* S T
obtains $M' N' U k L C$ **where**
state $S = (M', N', U, k, \text{None})$ **and**
state $T = (\text{Propagated } L (C + \{\#L\}) \# M', N', U, k, \text{None})$ **and**
 $C + \{\#L\} \in \# \text{local.clauses } S$ **and**
 $M' \models_{\text{as}} C \text{Not } C$ **and**
undefined-lit (*trail* S) L
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-propagate-completeness*:

assumes MN : *set* $M \models_s \text{set-mset } N$ **and**
cons: *consistent-interp* (*set* M) **and**
tot: *total-over-m* (*set* M) (*set-mset* N) **and**
lits-of-l (*trail* S) $\subseteq \text{set } M$ **and**
init-clss $S = N$ **and**
*propagate*** $S S'$ **and**
learned-clss $S = \{\#\}$
shows $\text{length } (\text{trail } S) \leq \text{length } (\text{trail } S') \wedge \text{lits-of-l } (\text{trail } S') \subseteq \text{set } M$
 $\langle \text{proof} \rangle$

lemma

assumes *propagate*** S X
shows
rtrancpl-propagate-init-clss: *init-clss* $X = \text{init-clss } S$ **and**
rtrancpl-propagate-learned-clss: *learned-clss* $X = \text{learned-clss } S$
 $\langle \text{proof} \rangle$

lemma *completeness-is-a-full1-propagation*:

fixes $S :: 'st$ **and** $M :: 'v$ *literal list*
assumes MN : *set* $M \models_s \text{set-mset } N$
and *cons*: *consistent-interp* (*set* M)
and *tot*: *total-over-m* (*set* M) (*set-mset* N)
and *alien*: *no-strange-atm* S
and *learned*: *learned-clss* $S = \{\#\}$
and *clsS[simp]*: *init-clss* $S = N$
and *lits*: *lits-of-l* (*trail* S) $\subseteq \text{set } M$
shows $\exists S'. \text{propagate** } S S' \wedge \text{full } \text{cdcl}_W\text{-cp } S S'$
 $\langle \text{proof} \rangle$

See also *cdcl_W-cp*** $?S ?S' \implies \exists M. \text{trail } ?S' = M @ \text{trail } ?S \wedge (\forall l \in \text{set } M. \neg \text{is-marked } l)$

lemma *rtrancp-propagate-is-trail-append*:

*propagate** S T $\implies \exists c. \text{trail } T = c @ \text{trail } S$*
<proof>

lemma *rtrancp-propagate-is-update-trail*:

*propagate** S T $\implies \text{cdcl}_W\text{-M-level-inv } S \implies$*
init-clss S = init-clss T \wedge learned-clss S = learned-clss T \wedge backtrack-lvl S = backtrack-lvl T
 \wedge conflicting S = conflicting T
<proof>

lemma *cdcl_W-stgy-strong-completeness-n*:

assumes

MN: set M \models_s set-mset (mset-clss N) and
cons: consistent-interp (set M) and
tot: total-over-m (set M) (set-mset (mset-clss N)) and
atm-incl: atm-of ' (set M) \subseteq atms-of-mm (mset-clss N) and
distM: distinct M and
length: n \leq length M

shows

$\exists M' k S. \text{length } M' \geq n \wedge$
lits-of-l M' \subseteq set M \wedge
no-dup M' \wedge
state S = (M', mset-clss N, {#}, k, None) \wedge
*cdcl_W-stgy** (init-state N) S*

<proof>

lemma *cdcl_W-stgy-strong-completeness*:

assumes

MN: set M \models_s set-mset (mset-clss N) and
cons: consistent-interp (set M) and
tot: total-over-m (set M) (set-mset (mset-clss N)) and
atm-incl: atm-of ' (set M) \subseteq atms-of-mm (mset-clss N) and
distM: distinct M

shows

$\exists M' k S.$
lits-of-l M' = set M \wedge
state S = (M', mset-clss N, {#}, k, None) \wedge
*cdcl_W-stgy** (init-state N) S \wedge*
final-cdcl_W-state S

<proof>

19.5.6 No conflict with only variables of level less than backtrack level

This invariant is stronger than the previous argument in the sense that it is a property about all possible conflicts.

definition *no-smaller-confl* (S :: 'st) \equiv

$(\forall M K i M' D. M' @ \text{Marked } K i \# M = \text{trail } S \longrightarrow D \in \# \text{ clauses } S$
 $\longrightarrow \neg M \models_{as} \text{CNot } D)$

lemma *no-smaller-confl-init-sate[simp]*:

no-smaller-confl (init-state N) <proof>

lemma *cdcl_W-o-no-smaller-confl-inv*:

fixes S S' :: 'st

assumes

cdcl_W-o S S' **and**
lev: cdcl_W-M-level-inv S **and**
max-lev: conflict-is-false-with-level S **and**
smaller: no-smaller-conflict S **and**
no-f: no-clause-is-false S
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *conflict-no-smaller-conflict-inv:*
assumes *conflict S S'*
and *no-smaller-conflict S*
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *propagate-no-smaller-conflict-inv:*
assumes *propagate: propagate S S'*
and *n-l: no-smaller-conflict S*
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *cdcl_W-cp-no-smaller-conflict-inv:*
assumes *propagate: cdcl_W-cp S S'*
and *n-l: no-smaller-conflict S*
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *rtrancp-cdcl_W-cp-no-smaller-conflict-inv:*
assumes *propagate: cdcl_W-cp^{**} S S'*
and *n-l: no-smaller-conflict S*
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *trancp-cdcl_W-cp-no-smaller-conflict-inv:*
assumes *propagate: cdcl_W-cp⁺⁺ S S'*
and *n-l: no-smaller-conflict S*
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *full-cdcl_W-cp-no-smaller-conflict-inv:*
assumes *full cdcl_W-cp S S'*
and *n-l: no-smaller-conflict S*
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *full1-cdcl_W-cp-no-smaller-conflict-inv:*
assumes *full1 cdcl_W-cp S S'*
and *n-l: no-smaller-conflict S*
shows *no-smaller-conflict S'*
 ⟨proof⟩

lemma *cdcl_W-stgy-no-smaller-conflict-inv:*
assumes *cdcl_W-stgy S S'*
and *n-l: no-smaller-conflict S*
and *conflict-is-false-with-level S*
and *cdcl_W-M-level-inv S*

shows *no-smaller-confl* S'
 $\langle \text{proof} \rangle$

lemma *is-conflicting-exists-conflict*:

assumes $\neg(\forall D \in \# \text{init-clss } S' + \text{learned-clss } S'. \neg \text{trail } S' \models_{\text{as}} \text{CNot } D)$
and *conflicting* $S' = \text{None}$
shows $\exists S''. \text{conflict } S' S''$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-o-conflict-is-no-clause-is-false*:

fixes $S S' :: 'st$
assumes
cdcl_W-o $S S'$ **and**
lev: *cdcl_W-M-level-inv* S **and**
max-lev: *conflict-is-false-with-level* S **and**
no-f: *no-clause-is-false* S **and**
no-l: *no-smaller-confl* S
shows *no-clause-is-false* S'
 $\vee (\text{conflicting } S' = \text{None}$
 $\longrightarrow (\forall D \in \# \text{clauses } S'. \text{trail } S' \models_{\text{as}} \text{CNot } D$
 $\longrightarrow (\exists L. L \in \# D \wedge \text{get-level } (\text{trail } S') L = \text{backtrack-lvl } S')))$
 $\langle \text{proof} \rangle$

lemma *full1-cdcl_W-cp-exists-conflict-decompose*:

assumes
confl: $\exists D \in \# \text{clauses } S. \text{trail } S \models_{\text{as}} \text{CNot } D$ **and**
full: *full cdcl_W-cp* $S U$ **and**
no-confl: *conflicting* $S = \text{None}$ **and**
lev: *cdcl_W-M-level-inv* S
shows $\exists T. \text{propagate}^{**} S T \wedge \text{conflict } T U$
 $\langle \text{proof} \rangle$

lemma *full1-cdcl_W-cp-exists-conflict-full1-decompose*:

assumes
confl: $\exists D \in \# \text{clauses } S. \text{trail } S \models_{\text{as}} \text{CNot } D$ **and**
full: *full cdcl_W-cp* $S U$ **and**
no-confl: *conflicting* $S = \text{None}$ **and**
lev: *cdcl_W-M-level-inv* S
shows $\exists T D. \text{propagate}^{**} S T \wedge \text{conflict } T U$
 $\wedge \text{trail } T \models_{\text{as}} \text{CNot } D \wedge \text{conflicting } U = \text{Some } D \wedge D \in \# \text{clauses } S$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-no-smaller-confl*:

assumes
cdcl_W-stgy $S S'$ **and**
n-l: *no-smaller-confl* S **and**
conflict-is-false-with-level S **and**
cdcl_W-M-level-inv S **and**
no-clause-is-false S **and**
distinct-cdcl_W-state S **and**
cdcl_W-conflicting S
shows *no-smaller-confl* S'
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-ex-lit-of-max-level*:

assumes
cdcl_W-stgy S S' **and**
n-l: no-smaller-confl S **and**
conflict-is-false-with-level S **and**
cdcl_W-M-level-inv S **and**
no-clause-is-false S **and**
distinct-cdcl_W-state S **and**
cdcl_W-conflicting S
shows *conflict-is-false-with-level* S'
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-stgy-no-smaller-confl-inv:*

assumes
*cdcl_W-stgy*** S S' **and**
n-l: no-smaller-confl S **and**
cls-false: conflict-is-false-with-level S **and**
lev: cdcl_W-M-level-inv S **and**
no-f: no-clause-is-false S **and**
dist: distinct-cdcl_W-state S **and**
conflicting: cdcl_W-conflicting S **and**
decomp: all-decomposition-implies-m (*init-clss* S) (*get-all-marked-decomposition* (*trail* S)) **and**
learned: cdcl_W-learned-clause S **and**
alien: no-strange-atm S
shows *no-smaller-confl* $S' \wedge$ *conflict-is-false-with-level* S'
 $\langle \text{proof} \rangle$

19.5.7 Final States are Conclusive

lemma *full-cdcl_W-stgy-final-state-conclusive-non-false:*

fixes $S' :: 'st$
assumes *full: full cdcl_W-stgy* (*init-state* N) S'
and *no-d: distinct-mset-mset* (*mset-clss* N)
and *no-empty: $\forall D \in \#mset-clss$ $N. D \neq \{\#\}$*
shows (*conflicting* $S' = \text{Some } \{\#\} \wedge$ *unsatisfiable* (*set-mset* (*init-clss* S')))
 \vee (*conflicting* $S' = \text{None} \wedge$ *trail* $S' \models_{asm}$ *init-clss* S')
 $\langle \text{proof} \rangle$

lemma *conflict-is-full1-cdcl_W-cp:*

assumes *cp: conflict* S S'
shows *full1 cdcl_W-cp* S S'
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-fst-empty-conflicting-false:*

assumes
cdcl_W-cp S S' **and**
trail $S = []$ **and**
conflicting $S \neq \text{None}$
shows *False*
 $\langle \text{proof} \rangle$

lemma *cdcl_W-o-fst-empty-conflicting-false:*

assumes *cdcl_W-o* S S'
and *trail* $S = []$
and *conflicting* $S \neq \text{None}$
shows *False*

<proof>

lemma *cdcl_W-stgy-fst-empty-conflicting-false:*

assumes *cdcl_W-stgy S S'*

and *trail S = []*

and *conflicting S ≠ None*

shows *False*

<proof>

thm *cdcl_W-cp.induct[split-format(complete)]*

lemma *cdcl_W-cp-conflicting-is-false:*

cdcl_W-cp S S' ⇒ conflicting S = Some {#} ⇒ False

<proof>

lemma *rtrancp-cdcl_W-cp-conflicting-is-false:*

cdcl_W-cp⁺⁺ S S' ⇒ conflicting S = Some {#} ⇒ False

<proof>

lemma *cdcl_W-o-conflicting-is-false:*

cdcl_W-o S S' ⇒ conflicting S = Some {#} ⇒ False

<proof>

lemma *cdcl_W-stgy-conflicting-is-false:*

cdcl_W-stgy S S' ⇒ conflicting S = Some {#} ⇒ False

<proof>

lemma *rtrancp-cdcl_W-stgy-conflicting-is-false:*

*cdcl_W-stgy^{**} S S' ⇒ conflicting S = Some {#} ⇒ S' = S*

<proof>

lemma *full-cdcl_W-init-clss-with-false-normal-form:*

assumes

∀ m ∈ set M. ¬is-marked m and

E = Some D and

state S = (M, N, U, 0, E)

full cdcl_W-stgy S S' and

all-decomposition-implies-m (init-clss S) (get-all-marked-decomposition (trail S))

cdcl_W-learned-clause S

cdcl_W-M-level-inv S

no-strange-atm S

distinct-cdcl_W-state S

cdcl_W-conflicting S

shows *∃ M''. state S' = (M'', N, U, 0, Some {#})*

<proof>

lemma *full-cdcl_W-stgy-final-state-conclusive-is-one-false:*

fixes *S' :: 'st*

assumes *full: full cdcl_W-stgy (init-state N) S'*

and *no-d: distinct-mset-mset (mset-clss N)*

and *empty: {#} ∈# (mset-clss N)*

shows *conflicting S' = Some {#} ∧ unsatisfiable (set-mset (init-clss S'))*

<proof>

lemma *full-cdcl_W-stgy-final-state-conclusive:*

```

fixes  $S' :: 'st$ 
assumes  $full$ :  $full\ cdcl_W\text{-}stgy\ (init\text{-}state\ N)\ S'$  and  $no\text{-}d$ :  $distinct\text{-}mset\text{-}mset\ (mset\text{-}clss\ N)$ 
shows  $(conflicting\ S' = Some\ \{\#\} \wedge unsatisfiable\ (set\text{-}mset\ (init\text{-}clss\ S')))$ 
   $\vee (conflicting\ S' = None \wedge trail\ S' \models_{asm}\ init\text{-}clss\ S')$ 
 $\langle proof \rangle$ 

lemma  $full\text{-}cdcl_W\text{-}stgy\text{-}final\text{-}state\text{-}conclusive\text{-}from\text{-}init\text{-}state$ :
fixes  $S' :: 'st$ 
assumes  $full$ :  $full\ cdcl_W\text{-}stgy\ (init\text{-}state\ N)\ S'$ 
and  $no\text{-}d$ :  $distinct\text{-}mset\text{-}mset\ (mset\text{-}clss\ N)$ 
shows  $(conflicting\ S' = Some\ \{\#\} \wedge unsatisfiable\ (set\text{-}mset\ (mset\text{-}clss\ N)))$ 
   $\vee (conflicting\ S' = None \wedge trail\ S' \models_{asm}\ (mset\text{-}clss\ N) \wedge satisfiable\ (set\text{-}mset\ (mset\text{-}clss\ N)))$ 
 $\langle proof \rangle$ 
end
end
theory  $CDCL\text{-}W\text{-}Termination$ 
imports  $CDCL\text{-}W$ 
begin

context  $conflict\text{-}driven\text{-}clause\text{-}learning_W$ 
begin

```

19.6 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  $cdcl_W\text{-}all\text{-}struct\text{-}inv$  where
 $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \longleftrightarrow$ 
   $no\text{-}strange\text{-}atm\ S \wedge$ 
   $cdcl_W\text{-}M\text{-}level\text{-}inv\ S \wedge$ 
   $(\forall s \in \# \text{ learned-clss } S. \neg tautology\ s) \wedge$ 
   $distinct\text{-}cdcl_W\text{-}state\ S \wedge$ 
   $cdcl_W\text{-}conflicting\ S \wedge$ 
   $all\text{-}decomposition\text{-}implies\text{-}m\ (init\text{-}clss\ S)\ (get\text{-}all\text{-}marked\text{-}decomposition\ (trail\ S)) \wedge$ 
   $cdcl_W\text{-}learned\text{-}clause\ S$ 

```

```

lemma  $cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ :
assumes  $cdcl_W\ S\ S'$  and  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ 
shows  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S'$ 
 $\langle proof \rangle$ 

```

```

lemma  $rtrancp\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv\text{-}inv$ :
assumes  $cdcl_W^{**}\ S\ S'$  and  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$ 
shows  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S'$ 
 $\langle proof \rangle$ 

```

```

lemma  $cdcl_W\text{-}stgy\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv$ :
 $cdcl_W\text{-}stgy\ S\ T \implies cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \implies cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$ 
 $\langle proof \rangle$ 

```

```

lemma  $rtrancp\text{-}cdcl_W\text{-}stgy\text{-}cdcl_W\text{-}all\text{-}struct\text{-}inv$ :
 $cdcl_W\text{-}stgy^{**}\ S\ T \implies cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \implies cdcl_W\text{-}all\text{-}struct\text{-}inv\ T$ 
 $\langle proof \rangle$ 

```

19.7 No Relearning of a clause

lemma *cdcl_W-o-new-clause-learned-is-backtrack-step:*

assumes *learned: $D \in \# \text{ learned-clss } T$ and*
new: $D \notin \# \text{ learned-clss } S$ and
cdcl_W: cdcl_W-o $S \ T$ and
lev: cdcl_W-M-level-inv S
shows *backtrack $S \ T \wedge \text{conflicting } S = \text{Some } D$*
<proof>

lemma *cdcl_W-cp-new-clause-learned-has-backtrack-step:*

assumes *learned: $D \in \# \text{ learned-clss } T$ and*
new: $D \notin \# \text{ learned-clss } S$ and
cdcl_W: cdcl_W-stgy $S \ T$ and
lev: cdcl_W-M-level-inv S
shows *$\exists S'. \text{backtrack } S \ S' \wedge \text{cdcl}_W\text{-stgy}^{**} \ S' \ T \wedge \text{conflicting } S = \text{Some } D$*
<proof>

lemma *rtrancpl-cdcl_W-cp-new-clause-learned-has-backtrack-step:*

assumes *learned: $D \in \# \text{ learned-clss } T$ and*
new: $D \notin \# \text{ learned-clss } S$ and
*cdcl_W: cdcl_W-stgy^{**} $S \ T$ and*
lev: cdcl_W-M-level-inv S
shows *$\exists S' \ S''. \text{cdcl}_W\text{-stgy}^{**} \ S \ S' \wedge \text{backtrack } S' \ S'' \wedge \text{conflicting } S' = \text{Some } D \wedge$*
*cdcl_W-stgy^{**} $S'' \ T$*
<proof>

lemma *propagate-no-more-Marked-lit:*

assumes *propagate $S \ S'$*
shows *Marked $K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$*
<proof>

lemma *conflict-no-more-Marked-lit:*

assumes *conflict $S \ S'$*
shows *Marked $K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$*
<proof>

lemma *cdcl_W-cp-no-more-Marked-lit:*

assumes *cdcl_W-cp $S \ S'$*
shows *Marked $K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$*
<proof>

lemma *rtrancpl-cdcl_W-cp-no-more-Marked-lit:*

assumes *cdcl_W-cp^{**} $S \ S'$*
shows *Marked $K \ i \in \text{set } (\text{trail } S) \longleftrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S')$*
<proof>

lemma *cdcl_W-o-no-more-Marked-lit:*

assumes *cdcl_W-o $S \ S'$ and lev: cdcl_W-M-level-inv S and $\neg \text{decide } S \ S'$*
shows *Marked $K \ i \in \text{set } (\text{trail } S') \longrightarrow \text{Marked } K \ i \in \text{set } (\text{trail } S)$*
<proof>

lemma *cdcl_W-new-marked-at-beginning-is-decide:*

assumes *cdcl_W-stgy $S \ S'$ and*
lev: cdcl_W-M-level-inv S and
trail $S' = M' @ \text{Marked } L \ i \ \# \ M$ and

trail $S = M$
shows $\exists T. \text{decide } S \ T \wedge \text{no-step } \text{cdcl}_W\text{-cp } S$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-o-is-decide*:

assumes *cdcl_W-o* $S \ T$ **and** *lev*: *cdcl_W-M-level-inv* S
trail $T = \text{drop } (\text{length } M_0) \ M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**
 $\neg (\exists M'. \text{trail } S = M' @ \text{Marked } L \ i \ \# \ H @ M)$
shows *decide* $S \ T$
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-new-marked-at-beginning-is-decide*:

assumes *cdcl_W-stgy*** $R \ U$ **and**
trail $U = M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**
trail $R = M$ **and**
cdcl_W-M-level-inv R
shows
 $\exists S \ T \ T'. \text{cdcl}_W\text{-stgy}^{**} \ R \ S \wedge \text{decide } S \ T \wedge \text{cdcl}_W\text{-stgy}^{**} \ T \ U \wedge \text{cdcl}_W\text{-stgy}^{**} \ S \ U \wedge$
 $\text{no-step } \text{cdcl}_W\text{-cp } S \wedge \text{trail } T = \text{Marked } L \ i \ \# \ H @ M \wedge \text{trail } S = H @ M \wedge \text{cdcl}_W\text{-stgy } S \ T' \wedge$
 $\text{cdcl}_W\text{-stgy}^{**} \ T' \ U$
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-new-marked-at-beginning-is-decide'*:

assumes *cdcl_W-stgy*** $R \ U$ **and**
trail $U = M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**
trail $R = M$ **and**
cdcl_W-M-level-inv R
shows $\exists y \ y'. \text{cdcl}_W\text{-stgy}^{**} \ R \ y \wedge \text{cdcl}_W\text{-stgy } y \ y' \wedge \neg (\exists c. \text{trail } y = c @ \text{Marked } L \ i \ \# \ H @ M)$
 $\wedge (\lambda a \ b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L \ i \ \# \ H @ M))^{**} \ y' \ U$
 $\langle \text{proof} \rangle$

lemma *beginning-not-marked-invert*:

assumes $A: M @ A = M' @ \text{Marked } K \ i \ \# \ H$ **and**
 $\text{nm}: \forall m \in \text{set } M. \neg \text{is-marked } m$
shows $\exists M. A = M @ \text{Marked } K \ i \ \# \ H$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-trail-has-new-marked-is-decide-step*:

assumes *cdcl_W-stgy* $S \ T$
 $\neg (\exists c. \text{trail } S = c @ \text{Marked } L \ i \ \# \ H @ M)$ **and**
 $(\lambda a \ b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L \ i \ \# \ H @ M))^{**} \ T \ U$ **and**
 $\exists M'. \text{trail } U = M' @ \text{Marked } L \ i \ \# \ H @ M$ **and**
lev: *cdcl_W-M-level-inv* S
shows $\exists S'. \text{decide } S \ S' \wedge \text{full } \text{cdcl}_W\text{-cp } S' \ T \wedge \text{no-step } \text{cdcl}_W\text{-cp } S$
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-stgy-with-trail-end-has-trail-end*:

assumes $(\lambda a \ b. \text{cdcl}_W\text{-stgy } a \ b \wedge (\exists c. \text{trail } a = c @ \text{Marked } L \ i \ \# \ H @ M))^{**} \ T \ U$ **and**
 $\exists M'. \text{trail } U = M' @ \text{Marked } L \ i \ \# \ H @ M$
shows $\exists M'. \text{trail } T = M' @ \text{Marked } L \ i \ \# \ H @ M$
 $\langle \text{proof} \rangle$

lemma *remove1-mset-eq-remove1-mset-same*:

remove1-mset $L \ D = \text{remove1-mset } L' \ D \implies L \in \# \ D \implies L = L'$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-o-cannot-learn:*

assumes

cdcl_W-o y z and

lev: cdcl_W-M-level-inv y and

trM: trail y = c @ Marked Kh i # H and

DL: D ∉ # learned-clss y and

LD: L ∈ # D and

DH: atms-of (remove1-mset L D) ⊆ atm-of 'lits-of-l H and

LH: atm-of L ∉ atm-of 'lits-of-l H and

learned: ∀ T. conflicting y = Some T ⟶ trail y ⊨_{as} CNot T and

z: trail z = c' @ Marked Kh i # H

shows *D ∉ # learned-clss z*

<proof>

lemma *cdcl_W-stgy-with-trail-end-has-not-been-learned:*

assumes

cdcl_W-stgy y z and

cdcl_W-M-level-inv y and

trail y = c @ Marked Kh i # H and

D ∉ # learned-clss y and

LD: L ∈ # D and

DH: atms-of (remove1-mset L D) ⊆ atm-of 'lits-of-l H and

LH: atm-of L ∉ atm-of 'lits-of-l H and

∀ T. conflicting y = Some T ⟶ trail y ⊨_{as} CNot T and

trail z = c' @ Marked Kh i # H

shows *D ∉ # learned-clss z*

<proof>

lemma *rtrancp-cdcl_W-stgy-with-trail-end-has-not-been-learned:*

assumes

*(λa b. cdcl_W-stgy a b ∧ (∃ c. trail a = c @ Marked K i # H @ []))** S z and*

cdcl_W-all-struct-inv S and

trail S = c @ Marked K i # H and

D ∉ # learned-clss S and

LD: L ∈ # D and

DH: atms-of (remove1-mset L D) ⊆ atm-of 'lits-of-l H and

LH: atm-of L ∉ atm-of 'lits-of-l H and

∃ c'. trail z = c' @ Marked K i # H

shows *D ∉ # learned-clss z*

<proof>

lemma *cdcl_W-stgy-new-learned-clause:*

assumes *cdcl_W-stgy S T and*

lev: cdcl_W-M-level-inv S and

E ∉ # learned-clss S and

E ∈ # learned-clss T

shows *∃ S'. backtrack S S' ∧ conflicting S = Some E ∧ full cdcl_W-cp S' T*

<proof>

lemma *cdcl_W-stgy-no-relearned-clause:*

assumes

invR: cdcl_W-all-struct-inv R and

*st': cdcl_W-stgy** R S and*

bt: backtrack S T and

conflict: raw-conflicting $S = \text{Some } E$ **and**
already-learned: $\text{mset-clss } E \in \# \text{ clauses } S$ **and**
 R : trail $R = []$
shows *False*
 <proof>

lemma *rtrancpl-cdcl_W-stgy-distinct-mset-clauses*:
assumes
 invR: *cdcl_W-all-struct-inv* R **and**
 st: *cdcl_W-stgy*** R S **and**
 dist: *distinct-mset* (*clauses* R) **and**
 R : trail $R = []$
shows *distinct-mset* (*clauses* S)
 <proof>

lemma *cdcl_W-stgy-distinct-mset-clauses*:
assumes
 st: *cdcl_W-stgy*** (*init-state* N) S **and**
 no-duplicate-clause: *distinct-mset* (*mset-clss* N) **and**
 no-duplicate-in-clause: *distinct-mset-mset* (*mset-clss* N)
shows *distinct-mset* (*clauses* S)
 <proof>

19.8 Decrease of a measure

fun *cdcl_W-measure* **where**
cdcl_W-measure $S =$
 [($3::\text{nat}$) \wedge (*card* (*atms-of-mm* (*init-clss* S))) $-$ *card* (*set-mset* (*learned-clss* S))],
 if *conflicting* $S = \text{None}$ then 1 else 0,
 if *conflicting* $S = \text{None}$ then *card* (*atms-of-mm* (*init-clss* S)) $-$ *length* (*trail* S)
 else *length* (*trail* S)
]

lemma *length-model-le-vars-all-inv*:
assumes *cdcl_W-all-struct-inv* S
shows *length* (*trail* S) \leq *card* (*atms-of-mm* (*init-clss* S))
 <proof>
end

context *conflict-driven-clause-learning_W*
begin

lemma *learned-clss-less-upper-bound*:
fixes $S :: 'st$
assumes
 distinct-cdcl_W-state S **and**
 $\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$
shows *card*(*set-mset* (*learned-clss* S)) $\leq 3 \wedge$ *card* (*atms-of-mm* (*learned-clss* S))
 <proof>

lemma *cdcl_W-measure-decreasing*:
fixes $S :: 'st$
assumes
 cdcl_W S S' **and**
 no-restart:

$\neg(\text{learned-clss } S \subseteq \# \text{ learned-clss } S' \wedge [] = \text{trail } S' \wedge \text{conflicting } S' = \text{None})$
and
no-forget: $\text{learned-clss } S \subseteq \# \text{ learned-clss } S'$ **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: $\text{cdcl}_W\text{-M-level-inv } S$ **and**
no-taut: $\forall s \in \# \text{ learned-clss } S. \neg \text{tautology } s$ **and**
no-dup: *distinct-cdcl_W-state* S **and**
conf: $\text{cdcl}_W\text{-conflicting } S$
shows $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$
 $\langle \text{proof} \rangle$

lemma *propagate-measure-decreasing*:
fixes $S :: 'st$
assumes *propagate* $S S'$ **and** $\text{cdcl}_W\text{-all-struct-inv } S$
shows $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$
 $\langle \text{proof} \rangle$

lemma *conflict-measure-decreasing*:
fixes $S :: 'st$
assumes *conflict* $S S'$ **and** $\text{cdcl}_W\text{-all-struct-inv } S$
shows $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$
 $\langle \text{proof} \rangle$

lemma *decide-measure-decreasing*:
fixes $S :: 'st$
assumes *decide* $S S'$ **and** $\text{cdcl}_W\text{-all-struct-inv } S$
shows $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$
 $\langle \text{proof} \rangle$

lemma *trans-le*:
 $\text{trans } \{(a, (b::\text{nat})). a < b\}$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-measure-decreasing*:
fixes $S :: 'st$
assumes $\text{cdcl}_W\text{-cp } S S'$ **and** $\text{cdcl}_W\text{-all-struct-inv } S$
shows $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$
 $\langle \text{proof} \rangle$

lemma *tranclp-cdcl_W-cp-measure-decreasing*:
fixes $S :: 'st$
assumes $\text{cdcl}_W\text{-cp}^{++} S S'$ **and** $\text{cdcl}_W\text{-all-struct-inv } S$
shows $(\text{cdcl}_W\text{-measure } S', \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-step-decreasing*:
fixes $R S T :: 'st$
assumes $\text{cdcl}_W\text{-stgy } S T$ **and**
 $\text{cdcl}_W\text{-stgy}^{**} R S$
 $\text{trail } R = []$ **and**
 $\text{cdcl}_W\text{-all-struct-inv } R$
shows $(\text{cdcl}_W\text{-measure } T, \text{cdcl}_W\text{-measure } S) \in \text{lexn } \{(a, b). a < b\} \text{ } 3$
 $\langle \text{proof} \rangle$

```

lemma tranclp-cdclW-stgy-decreasing:
  fixes R S T :: 'st
  assumes cdclW-stgy++ R S
  trail R = [] and
  cdclW-all-struct-inv R
  shows  $(\text{cdcl}_W\text{-measure } S, \text{cdcl}_W\text{-measure } R) \in \text{lexn } \{(a, b). a < b\} \mathcal{P}$ 
  <proof>

lemma tranclp-cdclW-stgy-S0-decreasing:
  fixes R S T :: 'st
  assumes
    pl: cdclW-stgy++ (init-state N) S and
    no-dup: distinct-mset-mset (mset-cls N)
  shows  $(\text{cdcl}_W\text{-measure } S, \text{cdcl}_W\text{-measure } (\text{init-state } N)) \in \text{lexn } \{(a, b). a < b\} \mathcal{P}$ 
  <proof>

lemma wf-tranclp-cdclW-stgy:
  wf {(S::'st, init-state N) |
    S N. distinct-mset-mset (mset-cls N)  $\wedge$  cdclW-stgy++ (init-state N) S}
  <proof>

lemma cdclW-cp-wf-all-inv:
  wf {(S', S). cdclW-all-struct-inv S  $\wedge$  cdclW-cp S S'}
  (is wf ?R)
  <proof>

end

end
theory DPLL-CDCL-W-Implementation
imports Partial-Annotated-Clausal-Logic
begin

```

20 Simple Implementation of the DPLL and CDCL

20.1 Common Rules

20.1.1 Propagation

The following theorem holds:

```

lemma lits-of-l-unfold[iff]:
   $(\forall c \in \text{set } C. -c \in \text{lits-of-l } Ms) \longleftrightarrow Ms \models_{\text{as}} \text{CNot } (\text{mset } C)$ 
  <proof>

```

The right-hand version is written at a high-level, but only the left-hand side is executable.

definition *is-unit-clause* :: 'a literal list \Rightarrow ('a, 'b, 'c) marked-lit list \Rightarrow 'a literal option

where

```

is-unit-clause l M =
  (case List.filter ( $\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of-l } M$ ) l of
    a # []  $\Rightarrow$  if M  $\models_{\text{as}}$  CNot (mset l - {#a#}) then Some a else None
    | -  $\Rightarrow$  None)

```

definition *is-unit-clause-code* :: 'a literal list \Rightarrow ('a, 'b, 'c) marked-lit list \Rightarrow 'a literal option **where**

is-unit-clause-code $l\ M =$
 (case *List.filter* ($\lambda a. \text{atm-of } a \notin \text{atm-of ' lits-of-}l\ M$) l of
 $a \# [] \Rightarrow \text{if } (\forall c \in \text{set } (\text{remove1 } a\ l). -c \in \text{lits-of-}l\ M) \text{ then } \text{Some } a \text{ else } \text{None}$
 $| - \Rightarrow \text{None}$)

lemma *is-unit-clause-is-unit-clause-code*[code]:
is-unit-clause $l\ M = \text{is-unit-clause-code } l\ M$
 <proof>

lemma *is-unit-clause-some-undef*:
assumes *is-unit-clause* $l\ M = \text{Some } a$
shows *undefined-lit* $M\ a$
 <proof>

lemma *is-unit-clause-some-CNot*: *is-unit-clause* $l\ M = \text{Some } a \Rightarrow M \models_{as} \text{CNot } (\text{mset } l - \{\#a\# \})$
 <proof>

lemma *is-unit-clause-some-in*: *is-unit-clause* $l\ M = \text{Some } a \Rightarrow a \in \text{set } l$
 <proof>

lemma *is-unit-clause-nil*[simp]: *is-unit-clause* $[]\ M = \text{None}$
 <proof>

20.1.2 Unit propagation for all clauses

Finding the first clause to propagate

fun *find-first-unit-clause* :: 'a literal list list \Rightarrow ('a, 'b, 'c) marked-lit list
 \Rightarrow ('a literal \times 'a literal list) option **where**
find-first-unit-clause $(a \# l)\ M =$
 (case *is-unit-clause* $a\ M$ of
 None $\Rightarrow \text{find-first-unit-clause } l\ M$
 $| \text{Some } L \Rightarrow \text{Some } (L, a)) \mid$
find-first-unit-clause $[]\ - = \text{None}$

lemma *find-first-unit-clause-some*:
find-first-unit-clause $l\ M = \text{Some } (a, c)$
 $\Rightarrow c \in \text{set } l \wedge M \models_{as} \text{CNot } (\text{mset } c - \{\#a\# \}) \wedge \text{undefined-lit } M\ a \wedge a \in \text{set } c$
 <proof>

lemma *propagate-is-unit-clause-not-None*:
assumes *dist*: *distinct* c **and**
 $M: M \models_{as} \text{CNot } (\text{mset } c - \{\#a\# \})$ **and**
undef: *undefined-lit* $M\ a$ **and**
 $ac: a \in \text{set } c$
shows *is-unit-clause* $c\ M \neq \text{None}$
 <proof>

lemma *find-first-unit-clause-none*:
 $\text{distinct } c \Rightarrow c \in \text{set } l \Rightarrow M \models_{as} \text{CNot } (\text{mset } c - \{\#a\# \}) \Rightarrow \text{undefined-lit } M\ a \Rightarrow a \in \text{set } c$
 $\Rightarrow \text{find-first-unit-clause } l\ M \neq \text{None}$
 <proof>

20.1.3 Decide

fun *find-first-unused-var* :: 'a literal list list \Rightarrow 'a literal set \Rightarrow 'a literal option **where**

$find_first_unused_var (a \# l) M =$
 $(case List.find (\lambda lit. lit \notin M \wedge \neg lit \notin M) a of$
 $None \Rightarrow find_first_unused_var l M$
 $| Some a \Rightarrow Some a) |$
 $find_first_unused_var [] = None$

lemma *find-none[iff]*:
 $List.find (\lambda lit. lit \notin M \wedge \neg lit \notin M) a = None \longleftrightarrow atm-of ' set a \subseteq atm-of ' M$
 $\langle proof \rangle$

lemma *find-some*: $List.find (\lambda lit. lit \notin M \wedge \neg lit \notin M) a = Some b \implies b \in set a \wedge b \notin M \wedge \neg b \notin M$
 $\langle proof \rangle$

lemma *find-first-unused-var-None[iff]*:
 $find_first_unused_var l M = None \longleftrightarrow (\forall a \in set l. atm-of ' set a \subseteq atm-of ' M)$
 $\langle proof \rangle$

lemma *find-first-unused-var-Some-not-all-incl*:
assumes $find_first_unused_var l M = Some c$
shows $\neg(\forall a \in set l. atm-of ' set a \subseteq atm-of ' M)$
 $\langle proof \rangle$

lemma *find-first-unused-var-Some*:
 $find_first_unused_var l M = Some a \implies (\exists m \in set l. a \in set m \wedge a \notin M \wedge \neg a \notin M)$
 $\langle proof \rangle$

lemma *find-first-unused-var-undefined*:
 $find_first_unused_var l (lits-of-l Ms) = Some a \implies undefined-lit Ms a$
 $\langle proof \rangle$

end

theory *DPLL-W-Implementation*

imports *DPLL-CDCL-W-Implementation DPLL-W* $\sim\sim /src/HOL/Library/Code-Target-Numeral$
begin

20.2 Simple Implementation of DPLL

20.2.1 Combining the propagate and decide: a DPLL step

definition *DPLL-step* :: $int \text{ dpll}_W\text{-marked-lits} \times int \text{ literal list list}$
 $\Rightarrow int \text{ dpll}_W\text{-marked-lits} \times int \text{ literal list list}$ **where**

$DPLL\text{-step} = (\lambda(Ms, N).$
 $(case find_first_unit_clause N Ms of$
 $Some (L, -) \Rightarrow (Propagated L () \# Ms, N)$
 $| - \Rightarrow$
 $if \exists C \in set N. (\forall c \in set C. \neg c \in lits-of-l Ms)$
 $then$
 $(case backtrack_split Ms of$
 $(-, L \# M) \Rightarrow (Propagated (\neg (lit-of L)) () \# M, N)$
 $| (-, -) \Rightarrow (Ms, N)$
 $)$
 $else$
 $(case find_first_unused_var N (lits-of-l Ms) of$
 $Some a \Rightarrow (Marked a () \# Ms, N)$
 $| None \Rightarrow (Ms, N))))$

Example of propagation:

value *DPLL-step* ([*Marked* (*Neg* 1) ()], [[*Pos* (1::int), *Neg* 2]])

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

abbreviation *toS* $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list})$
 $(N:: \text{int literal list list}). (Ms, \text{mset} (\text{map mset } N))$

abbreviation *toS'* $\equiv \lambda(Ms::(\text{int}, \text{unit}, \text{unit}) \text{ marked-lit list},$
 $N:: \text{int literal list list}). (Ms, \text{mset} (\text{map mset } N))$

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-dpll_W-step*:

assumes *step*: (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*)

and *neg*: (*Ms*, *N*) \neq (*Ms'*, *N'*)

shows *dpll_W* (*toS Ms N*) (*toS Ms' N'*)

$\langle \text{proof} \rangle$

lemma *DPLL-step-stuck-final-state*:

assumes *step*: (*Ms*, *N*) = *DPLL-step* (*Ms*, *N*)

shows *conclusive-dpll_W-state* (*toS Ms N*)

$\langle \text{proof} \rangle$

20.2.2 Adding invariants

Invariant tested in the function **function** *DPLL-ci* :: *int dpll_W-marked-lits* \Rightarrow *int literal list list*

\Rightarrow *int dpll_W-marked-lits* \times *int literal list list* **where**

DPLL-ci Ms N =

(if $\neg \text{dpll}_W\text{-all-inv}$ (*Ms*, *mset* (*map mset N*))

then (*Ms*, *N*)

else

let (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*) in

if (*Ms'*, *N'*) = (*Ms*, *N*) then (*Ms*, *N*) else *DPLL-ci Ms' N*)

$\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

No invariant tested **function** (*domintros*) *DPLL-part*:: *int dpll_W-marked-lits* \Rightarrow *int literal list list*

\Rightarrow

int dpll_W-marked-lits \times *int literal list list* **where**

DPLL-part Ms N =

(let (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*) in

if (*Ms'*, *N'*) = (*Ms*, *N*) then (*Ms*, *N*) else *DPLL-part Ms' N*)

$\langle \text{proof} \rangle$

lemma *snd-DPLL-step[simp]*:

snd (*DPLL-step* (*Ms*, *N*)) = *N*

$\langle \text{proof} \rangle$

lemma *dpll_W-all-inv-implicS-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*)

$\langle \text{proof} \rangle$

lemma *DPLL-ci-dpll_W-rtrancp*:

assumes $DPLL\text{-}ci\ Ms\ N = (Ms', N')$
shows $dpll_W^{**}\ (toS\ Ms\ N)\ (toS\ Ms'\ N)$
 $\langle proof \rangle$

lemma $dpll_W\text{-}all\text{-}inv\text{-}dpll_W\text{-}tranclp\text{-}irrefl$:
assumes $dpll_W\text{-}all\text{-}inv\ (Ms, N)$
and $dpll_W^{++}\ (Ms, N)\ (Ms, N)$
shows $False$
 $\langle proof \rangle$

lemma $DPLL\text{-}ci\text{-}final\text{-}state$:
assumes $step: DPLL\text{-}ci\ Ms\ N = (Ms, N)$
and $inv: dpll_W\text{-}all\text{-}inv\ (toS\ Ms\ N)$
shows $conclusive\text{-}dpll_W\text{-}state\ (toS\ Ms\ N)$
 $\langle proof \rangle$

lemma $DPLL\text{-}step\text{-}obtains$:
obtains Ms' **where** $(Ms', N) = DPLL\text{-}step\ (Ms, N)$
 $\langle proof \rangle$

lemma $DPLL\text{-}ci\text{-}obtains$:
obtains Ms' **where** $(Ms', N) = DPLL\text{-}ci\ Ms\ N$
 $\langle proof \rangle$

lemma $DPLL\text{-}ci\text{-}no\text{-}more\text{-}step$:
assumes $step: DPLL\text{-}ci\ Ms\ N = (Ms', N')$
shows $DPLL\text{-}ci\ Ms'\ N' = (Ms', N')$
 $\langle proof \rangle$

lemma $DPLL\text{-}part\text{-}dpll_W\text{-}all\text{-}inv\text{-}final$:
fixes $M\ Ms':: (int, unit, unit)\ marked\text{-}lit\ list$ **and**
 $N :: int\ literal\ list\ list$
assumes $inv: dpll_W\text{-}all\text{-}inv\ (Ms, mset\ (map\ mset\ N))$
and $MsN: DPLL\text{-}part\ Ms\ N = (Ms', N)$
shows $conclusive\text{-}dpll_W\text{-}state\ (toS\ Ms'\ N) \wedge dpll_W^{**}\ (toS\ Ms\ N)\ (toS\ Ms'\ N)$
 $\langle proof \rangle$

Embedding the invariant into the type

Defining the type **typedef** $dpll_W\text{-}state =$
 $\{(M::(int, unit, unit)\ marked\text{-}lit\ list, N::int\ literal\ list\ list).$
 $dpll_W\text{-}all\text{-}inv\ (toS\ M\ N)\}$
morphisms $rough\text{-}state\text{-}of\ state\text{-}of$
 $\langle proof \rangle$

lemma
 $DPLL\text{-}part\text{-}dom\ ([], N)$
 $\langle proof \rangle$

Some type classes **instantiation** $dpll_W\text{-}state :: equal$
begin

definition $equal\text{-}dpll_W\text{-}state :: dpll_W\text{-}state \Rightarrow dpll_W\text{-}state \Rightarrow bool$ **where**
 $equal\text{-}dpll_W\text{-}state\ S\ S' = (rough\text{-}state\text{-}of\ S = rough\text{-}state\text{-}of\ S')$

instance
 $\langle proof \rangle$
end

DPLL **definition** $DPLL\text{-}step' :: dpll_W\text{-}state \Rightarrow dpll_W\text{-}state$ **where**
 $DPLL\text{-}step' S = state\text{-}of (DPLL\text{-}step (rough\text{-}state\text{-}of S))$

declare $rough\text{-}state\text{-}of\text{-}inverse[simp]$

lemma $DPLL\text{-}step\text{-}dpll_W\text{-}conc\text{-}inv$:
 $DPLL\text{-}step (rough\text{-}state\text{-}of S) \in \{(M, N). dpll_W\text{-}all\text{-}inv (toS M N)\}$
 $\langle proof \rangle$

lemma $rough\text{-}state\text{-}of\text{-}DPLL\text{-}step'\text{-}DPLL\text{-}step[simp]$:
 $rough\text{-}state\text{-}of (DPLL\text{-}step' S) = DPLL\text{-}step (rough\text{-}state\text{-}of S)$
 $\langle proof \rangle$

function $DPLL\text{-}tot :: dpll_W\text{-}state \Rightarrow dpll_W\text{-}state$ **where**
 $DPLL\text{-}tot S =$
 $(let S' = DPLL\text{-}step' S in$
 $if S' = S then S else DPLL\text{-}tot S')$
 $\langle proof \rangle$

termination
 $\langle proof \rangle$

lemma $[code]$:
 $DPLL\text{-}tot S =$
 $(let S' = DPLL\text{-}step' S in$
 $if S' = S then S else DPLL\text{-}tot S') \langle proof \rangle$

lemma $DPLL\text{-}tot\text{-}DPLL\text{-}step\text{-}DPLL\text{-}tot[simp]$: $DPLL\text{-}tot (DPLL\text{-}step' S) = DPLL\text{-}tot S$
 $\langle proof \rangle$

lemma $DOPLL\text{-}step'\text{-}DPLL\text{-}tot[simp]$:
 $DPLL\text{-}step' (DPLL\text{-}tot S) = DPLL\text{-}tot S$
 $\langle proof \rangle$

lemma $DPLL\text{-}tot\text{-}final\text{-}state$:
assumes $DPLL\text{-}tot S = S$
shows $conclusive\text{-}dpll_W\text{-}state (toS' (rough\text{-}state\text{-}of S))$
 $\langle proof \rangle$

lemma $DPLL\text{-}tot\text{-}star$:
assumes $rough\text{-}state\text{-}of (DPLL\text{-}tot S) = S'$
shows $dpll_W^{**} (toS' (rough\text{-}state\text{-}of S)) (toS' S')$
 $\langle proof \rangle$

lemma $rough\text{-}state\text{-}of\text{-}rough\text{-}state\text{-}of\text{-}nil[simp]$:
 $rough\text{-}state\text{-}of (state\text{-}of ([], N)) = ([], N)$
 $\langle proof \rangle$

Theorem of correctness

lemma $DPLL\text{-}tot\text{-}correct$:

assumes *rough-state-of* (*DPLL-tot* (*state-of* ($([], N)$))) = (*M*, *N'*)
and (*M'*, *N''*) = *toS'* (*M*, *N'*)
shows $M' \models_{asm} N'' \longleftrightarrow \text{satisfiable } (\text{set-mset } N'')$
 <proof>

20.2.3 Code export

A conversion to DPLL-W-Implementation. *dpll_W-state* **definition** *Con* :: (*int*, *unit*, *unit*) *marked-lit*
list × *int literal list list*

\Rightarrow *dpll_W-state* **where**

Con *xs* = *state-of* (*if* *dpll_W-all-inv* (*toS* (*fst* *xs*) (*snd* *xs*)) *then* *xs* *else* ($[], []$))

lemma [*code abstype*]:

Con (*rough-state-of* *S*) = *S*

<proof>

declare *rough-state-of-DPLL-step'-DPLL-step*[*code abstract*]

lemma *Con-DPLL-step-rough-state-of-state-of*[*simp*]:

Con (*DPLL-step* (*rough-state-of* *s*)) = *state-of* (*DPLL-step* (*rough-state-of* *s*))

<proof>

A slightly different version of *DPLL-tot* where the returned boolean indicates the result.

definition *DPLL-tot-rep* **where**

DPLL-tot-rep *S* =

(*let* (*M*, *N*) = (*rough-state-of* (*DPLL-tot* *S*)) *in* ($\forall A \in \text{set } N. (\exists a \in \text{set } A. a \in \text{lits-of-l } (M)), M$))

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

notation *image-mset* (**infixr** '*#* 90)

type-synonym '*a cdcl_W-mark* = '*a literal list*

type-synonym *cdcl_W-marked-level* = *nat*

type-synonym '*v cdcl_W-marked-lit* = ('*v*, *cdcl_W-marked-level*, '*v cdcl_W-mark*) *marked-lit*

type-synonym '*v cdcl_W-marked-lits* = ('*v*, *cdcl_W-marked-level*, '*v cdcl_W-mark*) *marked-lits*

type-synonym '*v cdcl_W-state* =

'*v cdcl_W-marked-lits* × '*v literal list list* × '*v literal list list* × *nat* ×

'*v literal list option*

abbreviation *raw-trail* :: '*a* × '*b* × '*c* × '*d* × '*e* \Rightarrow '*a* **where**

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 $\text{raw-tl-trail} :: 'a \text{ list} \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \text{ list} \times 'b \times 'c \times 'd \times 'e$ **where**
 $\text{raw-tl-trail} \equiv (\lambda (M, S). (\text{tl } M, S))$

abbreviation $\text{raw-init-clss} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b$ **where**
 $\text{raw-init-clss} \equiv \lambda (M, N, -). N$

abbreviation $\text{raw-learned-clss} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c$ **where**
 $\text{raw-learned-clss} \equiv \lambda (M, N, U, -). U$

abbreviation $\text{raw-backtrack-lvl} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'd$ **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 $\text{raw-conflicting} :: 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e$ **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 $\text{raw-add-learned-cls}$ **where**
 $\text{raw-add-learned-cls} \equiv \lambda C (M, N, U, S). (M, N, \{\#C\# \} + U, S)$

abbreviation raw-remove-cls **where**
 $\text{raw-remove-cls} \equiv \lambda C (M, N, U, S). (M, \text{removeAll-mset } C \ N, \text{removeAll-mset } C \ U, S)$

type-synonym $'v \text{ cdcl}_W\text{-state-inv-st} = ('v, \text{nat}, 'v \text{ literal list}) \text{ marked-lit list} \times$
 $'v \text{ literal list list} \times 'v \text{ literal list list} \times \text{nat} \times 'v \text{ literal list option}$

abbreviation $\text{raw-S0-cdcl}_W \ N \equiv (([], N, [], 0, \text{None}) :: 'v \text{ cdcl}_W\text{-state-inv-st})$

fun $\text{mmset-of-mlit}' :: ('v, \text{nat}, 'v \text{ literal list}) \text{ marked-lit} \Rightarrow ('v, \text{nat}, 'v \text{ clause}) \text{ marked-lit}$
where
 $\text{mmset-of-mlit}' (\text{Propagated } L \ C) = \text{Propagated } L \ (\text{mset } C) \mid$
 $\text{mmset-of-mlit}' (\text{Marked } L \ i) = \text{Marked } L \ i$

lemma $\text{lit-of-mmset-of-mlit}'[\text{simp}]$:
 $\text{lit-of } (\text{mmset-of-mlit}' \ x a) = \text{lit-of } x a$
 $\langle \text{proof} \rangle$

abbreviation trail **where**
 $\text{trail } S \equiv \text{map mmset-of-mlit}' (\text{raw-trail } S)$

abbreviation clauses-of-l **where**
 $\text{clauses-of-l} \equiv \lambda L. \text{mset } (\text{map mset } L)$

global-interpretation $\text{state}_W\text{-ops}$
 $\text{mset} :: 'v \text{ literal list} \Rightarrow 'v \text{ clause}$

op # remove1

clauses-of-l op @ λL C. L ∈ set C op # λC. remove1-cond (λL. mset L = mset C)

mset λxs ys. case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))
op # remove1

id id

λ(M, -). map mmset-of-mlit' M λ(M, -). hd M

λ(-, N, -). N

λ(-, -, U, -). U

λ(-, -, -, k, -). k

λ(-, -, -, -, C). C

λL (M, S). (L # M, S)

λ(M, S). (tl M, S)

λC (M, N, S). (M, C # N, S)

λC (M, N, U, S). (M, N, C # U, S)

λC (M, N, U, S). (M, filter (λL. mset L ≠ mset C) N, filter (λL. mset L ≠ 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)

λ(-, N, U, -). ([], N, U, 0, None)

⟨proof⟩

lemma *mmset-of-mlit'-mmset-of-mlit: mmset-of-mlit' l = mmset-of-mlit l*

⟨proof⟩

lemma *clauses-of-l-filter-removeAll:*

clauses-of-l [L←a . mset L ≠ mset C] = mset (removeAll (mset C) (map mset a))

⟨proof⟩

interpretation *statew*

mset::'v literal list ⇒ 'v clause

op # remove1

clauses-of-l op @ λL C. L ∈ set C op # λC. remove1-cond (λL. mset L = mset C)

mset λxs ys. case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))

op # remove1

id id

λ(M, -). map mmset-of-mlit' M λ(M, -). hd M

λ(-, N, -). N

λ(-, -, U, -). U

λ(-, -, -, k, -). k

λ(-, -, -, -, C). C

λL (M, S). (L # M, S)

λ(M, S). (tl M, S)

λC (M, N, S). (M, C # N, S)

λC (M, N, U, S). (M, N, C # U, S)

λC (M, N, U, S). (M, filter (λL. mset L ≠ mset C) N, filter (λL. mset L ≠ mset C) U, S)

$\lambda(k::nat) (M, N, U, -, D). (M, N, U, k, D)$
 $\lambda D (M, N, U, k, -). (M, N, U, k, D)$
 $\lambda N. ([], N, [], 0, None)$
 $\lambda(-, N, U, -). ([], N, U, 0, None)$
 $\langle proof \rangle$

global-interpretation *conflict-driven-clause-learning_W*

$mset::'v \text{ literal list} \Rightarrow 'v \text{ clause}$
 $op \# remove1$

$clauses-of-l \text{ op } @ \lambda L \ C. L \in \text{set } C \text{ op } \# \lambda C. remove1\text{-cond } (\lambda L. mset \ L = mset \ C)$

$mset \ \lambda xs \ ys. \text{case-prod append (fold } (\lambda x \ (ys, zs). (remove1 \ x \ ys, x \ \# \ zs)) \ xs \ (ys, []))$
 $op \# remove1$

$id \ id$

$\lambda(M, -). \text{map mmset-of-mlit}' \ M \ \lambda(M, -). \text{hd } M$
 $\lambda(-, N, -). \ N$
 $\lambda(-, -, U, -). \ U$
 $\lambda(-, -, -, k, -). \ k$
 $\lambda(-, -, -, -, C). \ C$

$\lambda L \ (M, S). (L \ \# \ M, S)$
 $\lambda(M, S). (tl \ M, S)$
 $\lambda C \ (M, N, S). (M, C \ \# \ N, S)$
 $\lambda C \ (M, N, U, S). (M, N, C \ \# \ U, S)$
 $\lambda C \ (M, N, U, S). (M, \text{filter } (\lambda L. mset \ L \neq mset \ C) \ N, \text{filter } (\lambda L. mset \ L \neq mset \ C) \ U, S)$
 $\lambda(k::nat) (M, N, U, -, D). (M, N, U, k, D)$
 $\lambda D (M, N, U, k, -). (M, N, U, k, D)$
 $\lambda N. ([], N, [], 0, None)$
 $\lambda(-, N, U, -). ([], N, U, 0, None)$
 $\langle proof \rangle$

declare $state\text{-simp}[simp \ del] \ raw\text{-clauses-def}[simp] \ state\text{-eq-def}[simp]$

notation $state\text{-eq}$ (**infix** ~ 50)

term $reduce\text{-trail-to}$

lemma $reduce\text{-trail-to-map}[simp]:$

$reduce\text{-trail-to} \ (map \ f \ M1) = reduce\text{-trail-to} \ M1$
 $\langle proof \rangle$

20.3 CDCL Implementation

20.3.1 Types and Additional Lemmas

lemma $true\text{-clss-remdups}[simp]:$

$I \models s \ (mset \circ \text{remdups}) \ 'N \longleftrightarrow I \models s \ mset \ 'N$
 $\langle proof \rangle$

lemma $satisfiable\text{-mset-remdups}[simp]:$

$satisfiable \ ((mset \circ \text{remdups}) \ 'N) \longleftrightarrow satisfiable \ (mset \ 'N)$
 $\langle proof \rangle$

We need some functions to convert between our abstract state $nat \ cdcl_W\text{-state}$ and the concrete state $'v \ cdcl_W\text{-state-inv-st}$.

abbreviation $\text{convert}C :: 'a \text{ list option} \Rightarrow 'a \text{ multiset option}$ **where**
 $\text{convert}C \equiv \text{map-option mset}$

lemma $\text{convert-Propagated}[\text{elim!}]$:
 $\text{mmset-of-mlit}' z = \text{Propagated } L \ C \Longrightarrow (\exists C'. z = \text{Propagated } L \ C' \wedge C = \text{mset } C')$
 $\langle \text{proof} \rangle$

lemma $\text{get-rev-level-map-convert}$:
 $\text{get-rev-level } (\text{map mmset-of-mlit}' M) \ n \ x = \text{get-rev-level } M \ n \ x$
 $\langle \text{proof} \rangle$

lemma $\text{get-level-map-convert}[\text{simp}]$:
 $\text{get-level } (\text{map mmset-of-mlit}' M) = \text{get-level } M$
 $\langle \text{proof} \rangle$

lemma $\text{get-rev-level-map-mmsetof-mlit}[\text{simp}]$:
 $\text{get-rev-level } (\text{map mmset-of-mlit } M) = \text{get-rev-level } M$
 $\langle \text{proof} \rangle$

lemma $\text{get-level-map-mmsetof-mlit}[\text{simp}]$:
 $\text{get-level } (\text{map mmset-of-mlit } M) = \text{get-level } M$
 $\langle \text{proof} \rangle$

lemma $\text{get-maximum-level-map-convert}[\text{simp}]$:
 $\text{get-maximum-level } (\text{map mmset-of-mlit}' M) \ D = \text{get-maximum-level } M \ D$
 $\langle \text{proof} \rangle$

lemma $\text{get-all-levels-of-marked-map-convert}[\text{simp}]$:
 $\text{get-all-levels-of-marked } (\text{map mmset-of-mlit}' M) = (\text{get-all-levels-of-marked } M)$
 $\langle \text{proof} \rangle$

lemma $\text{reduce-trail-to-empty-trail}[\text{simp}]$:
 $\text{reduce-trail-to } F \ (\[], aa, ab, ac, b) = (\[], aa, ab, ac, b)$
 $\langle \text{proof} \rangle$

lemma $\text{raw-trail-reduce-trail-to-length-le}$:
assumes $\text{length } F > \text{length } (\text{raw-trail } S)$
shows $\text{raw-trail } (\text{reduce-trail-to } F \ S) = []$
 $\langle \text{proof} \rangle$

lemma reduce-trail-to :
 $\text{reduce-trail-to } F \ S =$
 $((\text{if } \text{length } (\text{raw-trail } S) \geq \text{length } F$
 $\text{then } \text{drop } (\text{length } (\text{raw-trail } S) - \text{length } F) (\text{raw-trail } S)$
 $\text{else } []), \text{raw-init-clss } S, \text{raw-learned-clss } S, \text{raw-backtrack-lvl } S, \text{raw-conflicting } S)$
 $(\text{is } ?S = -)$
 $\langle \text{proof} \rangle$

Definition an abstract type

typedef $'v \text{ cdcl}_W\text{-state-inv} = \{S :: 'v \text{ cdcl}_W\text{-state-inv-st. cdcl}_W\text{-all-struct-inv } S\}$
morphisms $\text{rough-state-of state-of}$
 $\langle \text{proof} \rangle$

instantiation $\text{cdcl}_W\text{-state-inv} :: (\text{type}) \text{ equal}$
begin

definition *equal-cdcl_W-state-inv* :: 'v cdcl_W-state-inv \Rightarrow 'v cdcl_W-state-inv \Rightarrow bool **where**
equal-cdcl_W-state-inv *S S'* = (rough-state-of *S* = rough-state-of *S'*)

instance

<proof>

end

lemma *lits-of-map-convert[simp]*: *lits-of-l* (map mmset-of-mlit' *M*) = *lits-of-l* *M*

<proof>

lemma *undefined-lit-map-convert[iff]*:

undefined-lit (map mmset-of-mlit' *M*) *L* \longleftrightarrow *undefined-lit* *M* *L*

<proof>

lemma *true-annot-map-convert[simp]*: map mmset-of-mlit' *M* \models_a *N* \longleftrightarrow *M* \models_a *N*

<proof>

lemma *true-annots-map-convert[simp]*: map mmset-of-mlit' *M* \models_{as} *N* \longleftrightarrow *M* \models_{as} *N*

<proof>

lemmas *propagateE*

lemma *find-first-unit-clause-some-is-propagate*:

assumes *H*: *find-first-unit-clause* (*N* @ *U*) *M* = *Some* (*L*, *C*)

shows *propagate* (*M*, *N*, *U*, *k*, *None*) (*Propagated* *L* *C* # *M*, *N*, *U*, *k*, *None*)

<proof>

20.3.2 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* *S* (*do-propagate-step* *S*)

<proof>

lemma *do-propagate-step-option[simp]*:

conflicting *S* \neq *None* \implies *do-propagate-step* *S* = *S*

<proof>

thm *prod-cases*

lemma *do-propagate-step-no-step*:

assumes *dist*: $\forall c \in \text{set } (\text{raw-clauses } S). \text{distinct } c$ **and**

prop-step: *do-propagate-step* *S* = *S*

shows *no-step propagate* *S*

<proof>

Conflict fun *find-conflict* **where**

find-conflict *M* [] = *None* |

find-conflict *M* (*N* # *Ns*) = (if ($\forall c \in \text{set } N. -c \in \text{lits-of-l } M$) then *Some* *N* else *find-conflict* *M* *Ns*)

lemma *find-conflict-Some*:

find-conflict $M \ Ns = \text{Some } N \implies N \in \text{set } Ns \wedge M \models_{as} \text{CNot } (\text{mset } N)$
 $\langle \text{proof} \rangle$

lemma *find-conflict-None*:

find-conflict $M \ Ns = \text{None} \longleftrightarrow (\forall N \in \text{set } Ns. \neg M \models_{as} \text{CNot } (\text{mset } N))$
 $\langle \text{proof} \rangle$

lemma *find-conflict-None-no-conflict*:

find-conflict $M \ (N @ U) = \text{None} \longleftrightarrow \text{no-step conflict } (M, N, U, k, \text{None})$
 $\langle \text{proof} \rangle$

definition *do-conflict-step* **where**

do-conflict-step $S =$

(*case* S of
 (M, N, U, k, None) \Rightarrow
 (*case* *find-conflict* $M \ (N @ U)$ of
 Some $a \Rightarrow (M, N, U, k, \text{Some } a)$
 | *None* $\Rightarrow (M, N, U, k, \text{None})$)
 | $S \Rightarrow S$)

lemma *do-conflict-step*:

do-conflict-step $S \neq S \implies \text{conflict } S \ (\text{do-conflict-step } S)$
 $\langle \text{proof} \rangle$

lemma *do-conflict-step-no-step*:

do-conflict-step $S = S \implies \text{no-step conflict } S$
 $\langle \text{proof} \rangle$

lemma *do-conflict-step-option[simp]*:

conflicting $S \neq \text{None} \implies \text{do-conflict-step } S = S$
 $\langle \text{proof} \rangle$

lemma *do-conflict-step-conflicting[dest]*:

do-conflict-step $S \neq S \implies \text{conflicting } (\text{do-conflict-step } S) \neq \text{None}$
 $\langle \text{proof} \rangle$

definition *do-cp-step* **where**

do-cp-step $S =$

(*do-propagate-step* o *do-conflict-step*) S

lemma *cp-step-is-cdcl_W-cp*:

assumes H : *do-cp-step* $S \neq S$
shows *cdcl_W-cp* $S \ (\text{do-cp-step } S)$
 $\langle \text{proof} \rangle$

lemma *do-cp-step-eq-no-prop-no-conflict*:

do-cp-step $S = S \implies \text{do-conflict-step } S = S \wedge \text{do-propagate-step } S = S$
 $\langle \text{proof} \rangle$

lemma *no-cdcl_W-cp-iff-no-propagate-no-conflict*:

no-step cdcl_W-cp $S \longleftrightarrow \text{no-step propagate } S \wedge \text{no-step conflict } S$
 $\langle \text{proof} \rangle$

lemma *do-cp-step-eq-no-step*:

assumes

H: *do-cp-step* *S* = *S* **and**
 $\forall c \in \text{set } (\text{raw-init-clss } S \text{ @ raw-learned-clss } S). \text{ distinct } c$
shows *no-step* *cdcl_W-cp* *S*
 ⟨*proof*⟩

lemma *cdcl_W-cp-cdcl_W-st*: *cdcl_W-cp* *S S'* \implies *cdcl_W*** *S S'*
 ⟨*proof*⟩

lemma *cdcl_W-all-struct-inv-rough-state[simp]*: *cdcl_W-all-struct-inv* (*rough-state-of* *S*)
 ⟨*proof*⟩

lemma [*simp*]: *cdcl_W-all-struct-inv* *S* \implies *rough-state-of* (*state-of* *S*) = *S*
 ⟨*proof*⟩

lemma *rough-state-of-state-of-do-cp-step[simp]*:
rough-state-of (*state-of* (*do-cp-step* (*rough-state-of* *S*))) = *do-cp-step* (*rough-state-of* *S*)
 ⟨*proof*⟩

Skip fun *do-skip-step* :: '*v* *cdcl_W-state-inv-st* \Rightarrow '*v* *cdcl_W-state-inv-st* **where**
do-skip-step (*Propagated* *L C* # *Ls,N,U,k*, *Some D*) =
 (if $\neg L \notin \text{set } D \wedge D \neq []$
 then (*Ls*, *N*, *U*, *k*, *Some D*)
 else (*Propagated* *L C* # *Ls*, *N*, *U*, *k*, *Some D*)) |
do-skip-step *S* = *S*

lemma *do-skip-step*:
do-skip-step *S* \neq *S* \implies *skip* *S* (*do-skip-step* *S*)
 ⟨*proof*⟩

lemma *do-skip-step-no*:
do-skip-step *S* = *S* \implies *no-step* *skip* *S*
 ⟨*proof*⟩

lemma *do-skip-step-trail-is-None[iff]*:
do-skip-step *S* = (*a*, *b*, *c*, *d*, *None*) \longleftrightarrow *S* = (*a*, *b*, *c*, *d*, *None*)
 ⟨*proof*⟩

Resolve fun *maximum-level-code*:: '*a* *literal list* \Rightarrow ('*a*, *nat*, '*b*) *marked-lit list* \Rightarrow *nat*
where
maximum-level-code [] = 0 |
maximum-level-code (*L* # *Ls*) *M* = max (*get-level* *M L*) (*maximum-level-code* *Ls M*)

lemma *maximum-level-code-eq-get-maximum-level[simp]*:
maximum-level-code *D M* = *get-maximum-level* *M* (*mset* *D*)
 ⟨*proof*⟩

lemma [*code*]:
fixes *M* :: ('*a*::{*type*}, *nat*, '*b*) *marked-lit list*
shows *get-maximum-level* *M* (*mset* *D*) = *maximum-level-code* *D M*
 ⟨*proof*⟩

fun *do-resolve-step* :: '*v* *cdcl_W-state-inv-st* \Rightarrow '*v* *cdcl_W-state-inv-st* **where**
do-resolve-step (*Propagated* *L C* # *Ls*, *N*, *U*, *k*, *Some D*) =
 (if $\neg L \in \text{set } D \wedge \text{maximum-level-code } (\text{remove1 } (\neg L) D) (\text{Propagated } L C \# Ls) = k$
 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-all-struct-inv S \implies do-resolve-step S \neq S
 \implies resolve S (do-resolve-step S)

\langle proof \rangle

lemma do-resolve-step-no:

do-resolve-step S = S \implies no-step resolve S

\langle proof \rangle

lemma rough-state-of-state-of-resolve[simp]:

cdcl_W-all-struct-inv S \implies rough-state-of (state-of (do-resolve-step S)) = do-resolve-step S

\langle proof \rangle

lemma do-resolve-step-trail-is-None[iff]:

do-resolve-step S = (a, b, c, d, None) \longleftrightarrow S = (a, b, c, d, None)

\langle proof \rangle

Backjumping fun find-level-decomp where

find-level-decomp M [] D k = None |

find-level-decomp M (L # Ls) D k =

(case (get-level M L, maximum-level-code (D @ Ls) M) of
(i, j) \Rightarrow if i = k \wedge j < i then Some (L, j) else find-level-decomp M Ls (L # D) k
)

lemma find-level-decomp-some:

assumes find-level-decomp M Ls D k = Some (L, j)

shows L \in set Ls \wedge get-maximum-level M (mset (remove1 L (Ls @ D))) = j \wedge get-level M L = k

\langle proof \rangle

lemma find-level-decomp-none:

assumes find-level-decomp M Ls E k = None **and** mset (L # D) = mset (Ls @ E)

shows \neg (L \in set Ls \wedge get-maximum-level M (mset D) < k \wedge k = get-level M L)

\langle proof \rangle

fun bt-cut where

bt-cut i (Propagated - - # Ls) = bt-cut i Ls |

bt-cut i (Marked K k # Ls) = (if k = Suc i then Some (Marked K k # Ls) else bt-cut i Ls) |

bt-cut i [] = None

lemma bt-cut-some-decomp:

bt-cut i M = Some M' $\implies \exists K M2 M1. M = M2 @ M' \wedge M' = \text{Marked } K (i+1) \# M1$

\langle proof \rangle

lemma bt-cut-not-none: M = M2 @ Marked K (Suc i) # M' \implies bt-cut i M \neq None

\langle proof \rangle

lemma get-all-marked-decomposition-ex:

$\exists N. (\text{Marked } K (\text{Suc } i) \# M', N) \in \text{set } (\text{get-all-marked-decomposition } (M2 @ \text{Marked } K (\text{Suc } i) \# M'))$

\langle proof \rangle

lemma bt-cut-in-get-all-marked-decomposition:

bt-cut *i* *M* = *Some* *M'* $\implies \exists M2. (M', M2) \in \text{set } (\text{get-all-marked-decomposition } M)$
 <proof>

fun *do-backtrack-step* **where**

do-backtrack-step (*M*, *N*, *U*, *k*, *Some* *D*) =
 (case *find-level-decomp* *M* *D* [] *k* of
 None \Rightarrow (*M*, *N*, *U*, *k*, *Some* *D*)
 | *Some* (*L*, *j*) \Rightarrow
 (case *bt-cut* *j* *M* of
 Some (*Marked* - - # *Ls*) \Rightarrow (*Propagated* *L* *D* # *Ls*, *N*, *D* # *U*, *j*, *None*)
 | - \Rightarrow (*M*, *N*, *U*, *k*, *Some* *D*))
) |
do-backtrack-step *S* = *S*

lemma *get-all-marked-decomposition-map-convert*:

(*get-all-marked-decomposition* (*map* *mmset-of-mlit'* *M*)) =
map ($\lambda(a, b). (\text{map } \text{mmset-of-mlit}' a, \text{map } \text{mmset-of-mlit}' b))$) (*get-all-marked-decomposition* *M*)
 <proof>

lemma *do-backtrack-step*:

assumes
db: *do-backtrack-step* *S* \neq *S* **and**
inv: *cdcl_W-all-struct-inv* *S*
shows *backtrack* *S* (*do-backtrack-step* *S*)
 <proof>

lemma *map-eq-list-length*:

map *f* *L* = *L'* $\implies \text{length } L = \text{length } L'$
 <proof>

lemma *map-mmset-of-mlit-eq-cons*:

assumes *map* *mmset-of-mlit'* *M* = *a* @ *c*
obtains *a'* *c'* **where**
M = *a'* @ *c'* **and**
a = *map* *mmset-of-mlit'* *a'* **and**
c = *map* *mmset-of-mlit'* *c'*
 <proof>

lemma *do-backtrack-step-no*:

assumes
db: *do-backtrack-step* *S* = *S* **and**
inv: *cdcl_W-all-struct-inv* *S*
shows *no-step* *backtrack* *S*
 <proof>

lemma *rough-state-of-state-of-backtrack[simp]*:

assumes *inv*: *cdcl_W-all-struct-inv* *S*
shows *rough-state-of* (*state-of* (*do-backtrack-step* *S*)) = *do-backtrack-step* *S*
 <proof>

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 \Rightarrow (*M*, *N*, *U*, *k*, *None*)
 | *Some* *L* \Rightarrow (*Marked* *L* (*Suc* *k*) # *M*, *N*, *U*, *k+1*, *None*)) |

do-decide-step $S = S$

lemma *do-decide-step*:

fixes $S :: 'v \text{ cdcl}_W\text{-state-inv-st}$
assumes *do-decide-step* $S \neq S$
shows *decide* S (*do-decide-step* S)
 $\langle \text{proof} \rangle$

lemma *mmset-of-mlit'-eq-Marked*[*iff*]: $\text{mmset-of-mlit}' z = \text{Marked } x \ k \longleftrightarrow z = \text{Marked } x \ k$
 $\langle \text{proof} \rangle$

lemma *do-decide-step-no*:

do-decide-step $S = S \implies \text{no-step decide } S$
 $\langle \text{proof} \rangle$

lemma *rough-state-of-state-of-do-decide-step*[*simp*]:

$\text{cdcl}_W\text{-all-struct-inv } S \implies \text{rough-state-of } (\text{state-of } (\text{do-decide-step } S)) = \text{do-decide-step } S$
 $\langle \text{proof} \rangle$

lemma *rough-state-of-state-of-do-skip-step*[*simp*]:

$\text{cdcl}_W\text{-all-struct-inv } S \implies \text{rough-state-of } (\text{state-of } (\text{do-skip-step } S)) = \text{do-skip-step } S$
 $\langle \text{proof} \rangle$

20.3.3 Code generation

Type definition There are two invariants: one while applying conflict and propagate and one for the other rules

declare *rough-state-of-inverse*[*simp add*]

definition *Con* **where**

Con $xs = \text{state-of } (\text{if } \text{cdcl}_W\text{-all-struct-inv } xs \text{ then } xs \text{ else } ([], [], [], 0, \text{None}))$

lemma [*code abstype*]:

Con (*rough-state-of* S) = S
 $\langle \text{proof} \rangle$

definition *do-cp-step'* **where**

do-cp-step' $S = \text{state-of } (\text{do-cp-step } (\text{rough-state-of } S))$

typedef $'v \text{ cdcl}_W\text{-state-inv-from-init-state} = \{S :: 'v \text{ cdcl}_W\text{-state-inv-st. } \text{cdcl}_W\text{-all-struct-inv } S$
 $\wedge \text{cdcl}_W\text{-stgy}^{**} (\text{raw-S0-cdcl}_W (\text{raw-init-cls } S)) S\}$

morphisms *rough-state-from-init-state-of* *state-from-init-state-of*

$\langle \text{proof} \rangle$

instantiation *cdcl_W-state-inv-from-init-state* :: (*type*) *equal*

begin

definition *equal-cdcl_W-state-inv-from-init-state* :: $'v \text{ cdcl}_W\text{-state-inv-from-init-state} \Rightarrow$

$'v \text{ cdcl}_W\text{-state-inv-from-init-state} \Rightarrow \text{bool}$ **where**

equal-cdcl_W-state-inv-from-init-state $S \ S' \longleftrightarrow$

$(\text{rough-state-from-init-state-of } S = \text{rough-state-from-init-state-of } S')$

instance

$\langle \text{proof} \rangle$

end

definition *ConI* **where**

ConI $S = \text{state-from-init-state-of } (\text{if } \text{cdcl}_W\text{-all-struct-inv } S$

$\wedge \text{cdcl}_W\text{-stgy}^{**} (\text{raw-}S0\text{-cdcl}_W (\text{raw-init-clss } S)) \text{ } S \text{ then } S \text{ else } ([], [], [], 0, \text{None}))$

lemma [code abstype]:

$\text{ConI } (\text{rough-state-from-init-state-of } S) = S$
 $\langle \text{proof} \rangle$

definition $\text{id-of-}I\text{-to} :: 'v \text{ cdcl}_W\text{-state-inv-from-init-state} \Rightarrow 'v \text{ cdcl}_W\text{-state-inv}$ **where**
 $\text{id-of-}I\text{-to } S = \text{state-of } (\text{rough-state-from-init-state-of } S)$

lemma [code abstract]:

$\text{rough-state-of } (\text{id-of-}I\text{-to } S) = \text{rough-state-from-init-state-of } S$
 $\langle \text{proof} \rangle$

Conflict and Propagate function $\text{do-full1-cp-step} :: 'v \text{ cdcl}_W\text{-state-inv} \Rightarrow 'v \text{ cdcl}_W\text{-state-inv}$
where

$\text{do-full1-cp-step } S =$
 $(\text{let } S' = \text{do-cp-step}' S \text{ in}$
 $\text{if } S = S' \text{ then } S \text{ else } \text{do-full1-cp-step } S')$
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

lemma $\text{do-full1-cp-step-fix-point-of-do-full1-cp-step}$:

$\text{do-cp-step}(\text{rough-state-of } (\text{do-full1-cp-step } S)) = \text{rough-state-of } (\text{do-full1-cp-step } S)$
 $\langle \text{proof} \rangle$

lemma $\text{in-clauses-rough-state-of-is-distinct}$:

$c \in \text{set } (\text{raw-init-clss } (\text{rough-state-of } S) @ \text{raw-learned-clss } (\text{rough-state-of } S)) \implies \text{distinct } c$
 $\langle \text{proof} \rangle$

lemma $\text{do-full1-cp-step-full}$:

$\text{full cdcl}_W\text{-cp } (\text{rough-state-of } S)$
 $(\text{rough-state-of } (\text{do-full1-cp-step } S))$
 $\langle \text{proof} \rangle$

lemma [code abstract]:

$\text{rough-state-of } (\text{do-cp-step}' S) = \text{do-cp-step } (\text{rough-state-of } S)$
 $\langle \text{proof} \rangle$

The other rules **fun** do-other-step **where**

$\text{do-other-step } S =$
 $(\text{let } T = \text{do-skip-step } S \text{ in}$
 $\text{if } T \neq S$
 $\text{then } T$
 else
 $(\text{let } U = \text{do-resolve-step } T \text{ in}$
 $\text{if } U \neq T$
 $\text{then } U \text{ else}$
 $(\text{let } V = \text{do-backtrack-step } U \text{ in}$
 $\text{if } V \neq U \text{ then } V \text{ else } \text{do-decide-step } V)))$

lemma do-other-step :

assumes $\text{inv: cdcl}_W\text{-all-struct-inv } S$ **and**
 $\text{st: do-other-step } S \neq S$
shows $\text{cdcl}_W\text{-o } S (\text{do-other-step } S)$

$\langle \text{proof} \rangle$

lemma *do-other-step-no*:

assumes *inv*: $\text{cdcl}_W\text{-all-struct-inv } S$ **and**

st: $\text{do-other-step } S = S$

shows $\text{no-step cdcl}_W\text{-o } S$

$\langle \text{proof} \rangle$

lemma *rough-state-of-state-of-do-other-step[simp]*:

$\text{rough-state-of } (\text{state-of } (\text{do-other-step } (\text{rough-state-of } S))) = \text{do-other-step } (\text{rough-state-of } S)$

$\langle \text{proof} \rangle$

definition *do-other-step'* **where**

$\text{do-other-step}' S =$

$\text{state-of } (\text{do-other-step } (\text{rough-state-of } S))$

lemma *rough-state-of-do-other-step'[code abstract]*:

$\text{rough-state-of } (\text{do-other-step}' S) = \text{do-other-step } (\text{rough-state-of } S)$

$\langle \text{proof} \rangle$

definition *do-cdcl_W-stgy-step* **where**

$\text{do-cdcl}_W\text{-stgy-step } S =$

$(\text{let } T = \text{do-full1-cp-step } S \text{ in}$

$\text{if } T \neq S$

$\text{then } T$

else

$(\text{let } U = (\text{do-other-step}' T) \text{ in}$

$(\text{do-full1-cp-step } U)))$

definition *do-cdcl_W-stgy-step'* **where**

$\text{do-cdcl}_W\text{-stgy-step}' S = \text{state-from-init-state-of } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{id-of-I-to } S)))$

lemma *toS-do-full1-cp-step-not-eq*: $\text{do-full1-cp-step } S \neq S \implies$

$\text{rough-state-of } S \neq \text{rough-state-of } (\text{do-full1-cp-step } S)$

$\langle \text{proof} \rangle$

do-full1-cp-step should not be unfolded anymore:

declare *do-full1-cp-step.simps*[*simp del*]

Correction of the transformation **lemma** *do-cdcl_W-stgy-step*:

assumes $\text{do-cdcl}_W\text{-stgy-step } S \neq S$

shows $\text{cdcl}_W\text{-stgy } (\text{rough-state-of } S) (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } S))$

$\langle \text{proof} \rangle$

lemma *do-skip-step-trail-changed-or-conflict*:

assumes *d*: $\text{do-other-step } S \neq S$

and *inv*: $\text{cdcl}_W\text{-all-struct-inv } S$

shows $\text{trail } S \neq \text{trail } (\text{do-other-step } S)$

$\langle \text{proof} \rangle$

lemma *do-full1-cp-step-induct*:

$(\bigwedge S. (S \neq \text{do-cp-step}' S \implies P (\text{do-cp-step}' S)) \implies P S) \implies P a0$

$\langle \text{proof} \rangle$

lemma *do-cp-step-neq-trail-increase*:

$\exists c. \text{raw-trail } (\text{do-cp-step } S) = c @ \text{raw-trail } S \wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$
 $\langle \text{proof} \rangle$

lemma *do-full1-cp-step-neq-trail-increase:*

$\exists c. \text{raw-trail } (\text{rough-state-of } (\text{do-full1-cp-step } S)) = c @ \text{raw-trail } (\text{rough-state-of } S)$
 $\wedge (\forall m \in \text{set } c. \neg \text{is-marked } m)$
 $\langle \text{proof} \rangle$

lemma *do-cp-step-conflicting:*

$\text{conflicting } (\text{rough-state-of } S) \neq \text{None} \implies \text{do-cp-step}' S = S$
 $\langle \text{proof} \rangle$

lemma *do-full1-cp-step-conflicting:*

$\text{conflicting } (\text{rough-state-of } S) \neq \text{None} \implies \text{do-full1-cp-step } S = S$
 $\langle \text{proof} \rangle$

lemma *do-decide-step-not-conflicting-one-more-decide:*

assumes
 $\text{conflicting } S = \text{None}$ **and**
 $\text{do-decide-step } S \neq S$
shows $\text{Suc } (\text{length } (\text{filter is-marked } (\text{raw-trail } S)))$
 $= \text{length } (\text{filter is-marked } (\text{raw-trail } (\text{do-decide-step } S)))$
 $\langle \text{proof} \rangle$

lemma *do-decide-step-not-conflicting-one-more-decide-bt:*

assumes $\text{conflicting } S \neq \text{None}$ **and**
 $\text{do-decide-step } S \neq S$
shows $\text{length } (\text{filter is-marked } (\text{raw-trail } S)) <$
 $\text{length } (\text{filter is-marked } (\text{raw-trail } (\text{do-decide-step } S)))$
 $\langle \text{proof} \rangle$

lemma *do-other-step-not-conflicting-one-more-decide-bt:*

assumes
 $\text{conflicting } (\text{rough-state-of } S) \neq \text{None}$ **and**
 $\text{conflicting } (\text{rough-state-of } (\text{do-other-step}' S)) = \text{None}$ **and**
 $\text{do-other-step}' S \neq S$
shows $\text{length } (\text{filter is-marked } (\text{raw-trail } (\text{rough-state-of } S)))$
 $> \text{length } (\text{filter is-marked } (\text{raw-trail } (\text{rough-state-of } (\text{do-other-step}' S))))$
 $\langle \text{proof} \rangle$

lemma *do-other-step-not-conflicting-one-more-decide:*

assumes $\text{conflicting } (\text{rough-state-of } S) = \text{None}$ **and**
 $\text{do-other-step}' S \neq S$
shows $1 + \text{length } (\text{filter is-marked } (\text{raw-trail } (\text{rough-state-of } S)))$
 $= \text{length } (\text{filter is-marked } (\text{raw-trail } (\text{rough-state-of } (\text{do-other-step}' S))))$
 $\langle \text{proof} \rangle$

lemma *rough-state-of-state-of-do-skip-step-rough-state-of[simp]:*

$\text{rough-state-of } (\text{state-of } (\text{do-skip-step } (\text{rough-state-of } S))) = \text{do-skip-step } (\text{rough-state-of } S)$
 $\langle \text{proof} \rangle$

lemma *conflicting-do-resolve-step-iff[iff]:*

$\text{conflicting } (\text{do-resolve-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
 $\langle \text{proof} \rangle$

lemma *conflicting-do-skip-step-iff*[iff]:
 $\text{conflicting } (\text{do-skip-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
 $\langle \text{proof} \rangle$

lemma *conflicting-do-decide-step-iff*[iff]:
 $\text{conflicting } (\text{do-decide-step } S) = \text{None} \longleftrightarrow \text{conflicting } S = \text{None}$
 $\langle \text{proof} \rangle$

lemma *conflicting-do-backtrack-step-imp*[simp]:
 $\text{do-backtrack-step } S \neq S \implies \text{conflicting } (\text{do-backtrack-step } S) = \text{None}$
 $\langle \text{proof} \rangle$

lemma *do-skip-step-eq-iff-trail-eq*:
 $\text{do-skip-step } S = S \longleftrightarrow \text{trail } (\text{do-skip-step } S) = \text{trail } S$
 $\langle \text{proof} \rangle$

lemma *do-decide-step-eq-iff-trail-eq*:
 $\text{do-decide-step } S = S \longleftrightarrow \text{trail } (\text{do-decide-step } S) = \text{trail } S$
 $\langle \text{proof} \rangle$

lemma *do-backtrack-step-eq-iff-trail-eq*:
 $\text{do-backtrack-step } S = S \longleftrightarrow \text{raw-trail } (\text{do-backtrack-step } S) = \text{raw-trail } S$
 $\langle \text{proof} \rangle$

lemma *do-resolve-step-eq-iff-trail-eq*:
 $\text{do-resolve-step } S = S \longleftrightarrow \text{trail } (\text{do-resolve-step } S) = \text{trail } S$
 $\langle \text{proof} \rangle$

lemma *do-other-step-eq-iff-trail-eq*:
 $\text{do-other-step } S = S \longleftrightarrow \text{raw-trail } (\text{do-other-step } S) = \text{raw-trail } S$
 $\langle \text{proof} \rangle$

lemma *do-full1-cp-step-do-other-step'-normal-form*[dest!]:
assumes $H: \text{do-full1-cp-step } (\text{do-other-step}' S) = S$
shows $\text{do-other-step}' S = S \wedge \text{do-full1-cp-step } S = S$
 $\langle \text{proof} \rangle$

lemma *do-cdcl_W-stgy-step-no*:
assumes $S: \text{do-cdcl}_W\text{-stgy-step } S = S$
shows $\text{no-step } \text{cdcl}_W\text{-stgy } (\text{rough-state-of } S)$
 $\langle \text{proof} \rangle$

lemma *toS-rough-state-of-state-of-rough-state-from-init-state-of*[simp]:
 $\text{rough-state-of } (\text{state-of } (\text{rough-state-from-init-state-of } S))$
 $= \text{rough-state-from-init-state-of } S$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-is-rtrancp-cdcl_W*: $\text{cdcl}_W\text{-cp } S T \implies \text{cdcl}_W^{**} S T$
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-cp-is-rtrancp-cdcl_W*: $\text{cdcl}_W\text{-cp}^{**} S T \implies \text{cdcl}_W^{**} S T$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-is-rtrancp-cdcl_W*:

$cdcl_W\text{-stgy } S \ T \implies cdcl_W^{**} \ S \ T$
 $\langle \text{proof} \rangle$

lemma $cdcl_W\text{-stgy-init-clss}$: $cdcl_W\text{-stgy } S \ T \implies cdcl_W\text{-M-level-inv } S \implies \text{init-clss } S = \text{init-clss } T$
 $\langle \text{proof} \rangle$

lemma $\text{clauses-toS-rough-state-of-do-cdcl}_W\text{-stgy-step[simp]}$:
 $\text{init-clss } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{state-of } (\text{rough-state-from-init-state-of } S))))$
 $= \text{init-clss } (\text{rough-state-from-init-state-of } S) \ (\text{is } - = \text{init-clss } ?S)$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-do-cp-step[simp]}$:
 $\text{raw-init-clss } (\text{do-cp-step } S) = \text{raw-init-clss } S$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-do-cp-step'[simp]}$:
 $\text{raw-init-clss } (\text{rough-state-of } (\text{do-cp-step}' S)) = \text{raw-init-clss } (\text{rough-state-of } S)$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-rough-state-of-do-full1-cp-step[simp]}$:
 $\text{raw-init-clss } (\text{rough-state-of } (\text{do-full1-cp-step } S))$
 $= \text{raw-init-clss } (\text{rough-state-of } S)$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-do-skip-def[simp]}$:
 $\text{raw-init-clss } (\text{do-skip-step } S) = \text{raw-init-clss } S$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-do-resolve-def[simp]}$:
 $\text{raw-init-clss } (\text{do-resolve-step } S) = \text{raw-init-clss } S$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-do-backtrack-def[simp]}$:
 $\text{raw-init-clss } (\text{do-backtrack-step } S) = \text{raw-init-clss } S$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-do-decide-def[simp]}$:
 $\text{raw-init-clss } (\text{do-decide-step } S) = \text{raw-init-clss } S$
 $\langle \text{proof} \rangle$

lemma $\text{raw-init-clss-rough-state-of-do-other-step'[simp]}$:
 $\text{raw-init-clss } (\text{rough-state-of } (\text{do-other-step}' S))$
 $= \text{raw-init-clss } (\text{rough-state-of } S)$
 $\langle \text{proof} \rangle$

lemma $[\text{simp}]$:
 $\text{raw-init-clss } (\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{state-of } (\text{rough-state-from-init-state-of } S))))$
 $=$
 $\text{raw-init-clss } (\text{rough-state-from-init-state-of } S)$
 $\langle \text{proof} \rangle$

lemma $\text{rough-state-from-init-state-of-do-cdcl}_W\text{-stgy-step'[code abstract]}$:
 $\text{rough-state-from-init-state-of } (\text{do-cdcl}_W\text{-stgy-step}' S) =$
 $\text{rough-state-of } (\text{do-cdcl}_W\text{-stgy-step } (\text{id-of-I-to } S))$
 $\langle \text{proof} \rangle$

All rules together function *do-all-cdcl_W-stgy* where

do-all-cdcl_W-stgy *S* =
 (let *T* = *do-cdcl_W-stgy-step'* *S* in
 if *T* = *S* then *S* else *do-all-cdcl_W-stgy* *T*)

⟨proof⟩

termination

⟨proof⟩

thm *do-all-cdcl_W-stgy.induct*

lemma *do-all-cdcl_W-stgy-induct*:

($\bigwedge S. (do-cdcl_W-stgy-step' S \neq S \implies P (do-cdcl_W-stgy-step' S)) \implies P S) \implies P a0$)

⟨proof⟩

lemma [*simp*]: *raw-init-clss* (*rough-state-from-init-state-of* (*do-all-cdcl_W-stgy* *S*)) =
raw-init-clss (*rough-state-from-init-state-of* *S*)

⟨proof⟩

lemma *no-step-cdcl_W-stgy-cdcl_W-all*:

fixes *S* :: 'a *cdcl_W-state-inv-from-init-state*

shows *no-step cdcl_W-stgy* (*rough-state-from-init-state-of* (*do-all-cdcl_W-stgy* *S*))

⟨proof⟩

lemma *do-all-cdcl_W-stgy-is-rtrancpl-cdcl_W-stgy*:

cdcl_W-stgy^{**} (*rough-state-from-init-state-of* *S*)

(*rough-state-from-init-state-of* (*do-all-cdcl_W-stgy* *S*))

⟨proof⟩

Final theorem:

lemma *consistent-interp-mmset-of-mlit*[*simp*]:

consistent-interp (*lit-of* 'mmset-of-mlit' 'set *M*') \longleftrightarrow

consistent-interp (*lit-of* 'set *M*')

⟨proof⟩

lemma *DPLL-tot-correct*:

assumes

r: *rough-state-from-init-state-of* (*do-all-cdcl_W-stgy* (*state-from-init-state-of*
 (([], map *remdups* *N*, [], 0, None)))) = *S* **and**

S: (*M'*, *N'*, *U'*, *k*, *E*) = *S*

shows (*E* ≠ *Some* [] ∧ *satisfiable* (set (map *mset* *N*)))

∨ (*E* = *Some* [] ∧ *unsatisfiable* (set (map *mset* *N*)))

⟨proof⟩

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

21 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: *conflict-driven-clause-learning_W.conflict*, *conflict-driven-clause-learning_W.resolve*, *conflict-driven-clause-learning_W.skip*, and *conflict-driven-clause-learning_W.backtrack* have to be done in a single step since they have a single counterpart in NOT's CDCL.

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 state.

21.1 Inclusion of the states

```
context conflict-driven-clause-learningW
begin
declare cdclW.intros[intro] cdclW-bj.intros[intro] cdclW-o.intros[intro]
```

```
lemma backtrack-no-cdclW-bj:
  assumes cdcl: cdclW-bj T U and inv: cdclW-M-level-inv V
  shows ¬backtrack V T
  ⟨proof⟩
```

```
inductive skip-or-resolve :: 'st ⇒ 'st ⇒ bool where
s-or-r-skip[intro]: skip S T ⇒ skip-or-resolve S T |
s-or-r-resolve[intro]: resolve S T ⇒ skip-or-resolve S T
```

```
lemma rtrancpl-cdclW-bj-skip-or-resolve-backtrack:
  assumes cdclW-bj** S U and inv: cdclW-M-level-inv S
  shows skip-or-resolve** S U ∨ (∃ T. skip-or-resolve** S T ∧ backtrack T U)
  ⟨proof⟩
```

```
lemma rtrancpl-skip-or-resolve-rtrancpl-cdclW:
  skip-or-resolve** S T ⇒ cdclW** S T
  ⟨proof⟩
```

```
definition backjump-l-cond :: 'v clause ⇒ 'v clause ⇒ 'v literal ⇒ 'st ⇒ 'st ⇒ bool where
backjump-l-cond ≡ λC C' L' S T. True
```

```
definition invNOT :: 'st ⇒ bool where
invNOT ≡ λS. no-dup (trail S)
```

```
declare invNOT-def[simp]
end
```

```
context conflict-driven-clause-learningW
begin
```

21.2 More lemmas conflict-propagate and backjumping

21.2.1 Termination

```
lemma cdclW-cp-normalized-element-all-inv:
  assumes inv: cdclW-all-struct-inv S
  obtains T where full cdclW-cp S T
  ⟨proof⟩
thm backtrackE
```

lemma *cdcl_W-bj-measure*:
assumes *cdcl_W-bj S T* **and** *cdcl_W-M-level-inv S*
shows *length (trail S) + (if conflicting S = None then 0 else 1)*
> length (trail T) + (if conflicting T = None then 0 else 1)
 $\langle \text{proof} \rangle$

lemma *wf-cdcl_W-bj*:
wf {(b,a). cdcl_W-bj a b \wedge cdcl_W-M-level-inv a}
 $\langle \text{proof} \rangle$

lemma *cdcl_W-bj-exists-normal-form*:
assumes *lev: cdcl_W-M-level-inv S*
shows $\exists T. \text{full } \text{cdcl}_W\text{-bj } S \ T$
 $\langle \text{proof} \rangle$

lemma *rtrancp-skip-state-decomp*:
assumes *skip** S T* **and** *no-dup (trail S)*
shows
 $\exists M. \text{trail } S = M @ \text{trail } T \wedge (\forall m \in \text{set } M. \neg \text{is-marked } m)$
init-clss S = init-clss T
learned-clss S = learned-clss T
backtrack-lvl S = backtrack-lvl T
conflicting S = conflicting T
 $\langle \text{proof} \rangle$

21.2.2 More backjumping

Backjumping after skipping or jump directly **lemma** *rtrancp-skip-backtrack-backtrack*:
assumes
*skip** S T* **and**
backtrack T W **and**
cdcl_W-all-struct-inv S
shows *backtrack S W*
 $\langle \text{proof} \rangle$

lemma *fst-get-all-marked-decomposition-prepend-not-marked*:
assumes $\forall m \in \text{set } MS. \neg \text{is-marked } m$
shows $\text{set } (\text{map } \text{fst } (\text{get-all-marked-decomposition } M))$
 $= \text{set } (\text{map } \text{fst } (\text{get-all-marked-decomposition } (MS @ M)))$
 $\langle \text{proof} \rangle$

See also $\llbracket \text{skip** } ?S \ ?T; \text{backtrack } ?T \ ?W; \text{cdcl}_W\text{-all-struct-inv } ?S \rrbracket \implies \text{backtrack } ?S \ ?W$

lemma *rtrancp-skip-backtrack-backtrack-end*:
assumes
*skip: skip** S T* **and**
bt: backtrack S W **and**
inv: cdcl_W-all-struct-inv S
shows *backtrack T W*
 $\langle \text{proof} \rangle$

lemma *cdcl_W-bj-decomp-resolve-skip-and-bj*:
assumes *cdcl_W-bj** S T* **and** *inv: cdcl_W-M-level-inv S*
shows $(\text{skip-or-resolve** } S \ T$
 $\vee (\exists U. \text{skip-or-resolve** } S \ U \wedge \text{backtrack } U \ T))$
 $\langle \text{proof} \rangle$

lemma *resolve-skip-deterministic*:

$resolve\ S\ T \implies skip\ S\ U \implies False$
 $\langle proof \rangle$

lemma *backtrack-unique:*

assumes
 $bt-T: backtrack\ S\ T$ **and**
 $bt-U: backtrack\ S\ U$ **and**
 $inv: cdcl_W\text{-all-struct-inv}\ S$
shows $T \sim U$
 $\langle proof \rangle$

lemma *if-can-apply-backtrack-no-more-resolve:*

assumes
 $skip: skip^{**}\ S\ U$ **and**
 $bt: backtrack\ S\ T$ **and**
 $inv: cdcl_W\text{-all-struct-inv}\ S$
shows $\neg resolve\ U\ V$
 $\langle proof \rangle$

lemma *if-can-apply-resolve-no-more-backtrack:*

assumes
 $skip: skip^{**}\ S\ U$ **and**
 $resolve: resolve\ S\ T$ **and**
 $inv: cdcl_W\text{-all-struct-inv}\ S$
shows $\neg backtrack\ U\ V$
 $\langle proof \rangle$

lemma *if-can-apply-backtrack-skip-or-resolve-is-skip:*

assumes
 $bt: backtrack\ S\ T$ **and**
 $skip: skip\text{-or-resolve}^{**}\ S\ U$ **and**
 $inv: cdcl_W\text{-all-struct-inv}\ S$
shows $skip^{**}\ S\ U$
 $\langle proof \rangle$

lemma *cdcl_W-bj-bj-decomp:*

assumes $cdcl_W\text{-bj}^{**}\ S\ W$ **and** $cdcl_W\text{-all-struct-inv}\ S$
shows
 $(\exists T\ U\ V. (\lambda S\ T. skip\text{-or-resolve}\ S\ T \wedge no\text{-step}\ backtrack\ S)^{**}\ S\ T$
 $\wedge (\lambda T\ U. resolve\ T\ U \wedge no\text{-step}\ backtrack\ T)\ T\ U$
 $\wedge skip^{**}\ U\ V \wedge backtrack\ V\ W)$
 $\vee (\exists T\ U. (\lambda S\ T. skip\text{-or-resolve}\ S\ T \wedge no\text{-step}\ backtrack\ S)^{**}\ S\ T$
 $\wedge (\lambda T\ U. resolve\ T\ U \wedge no\text{-step}\ backtrack\ T)\ T\ U \wedge skip^{**}\ U\ W)$
 $\vee (\exists T. skip^{**}\ S\ T \wedge backtrack\ T\ W)$
 $\vee skip^{**}\ S\ W$ (**is** $?RB\ S\ W \vee ?R\ S\ W \vee ?SB\ S\ W \vee ?S\ S\ W$)
 $\langle proof \rangle$

The case distinction is needed, since $T \sim V$ does not imply that $R^{**}\ T\ V$.

lemma *cdcl_W-bj-strongly-confluent:*

assumes
 $cdcl_W\text{-bj}^{**}\ S\ V$ **and**
 $cdcl_W\text{-bj}^{**}\ S\ T$ **and**
 $n\text{-s: no-step}\ cdcl_W\text{-bj}\ V$ **and**
 $inv: cdcl_W\text{-all-struct-inv}\ S$
shows $T \sim V \vee cdcl_W\text{-bj}^{**}\ T\ V$

$\langle proof \rangle$

lemma *cdcl_W-bj-unique-normal-form*:

assumes

ST: *cdcl_W-bj** S T* **and** *SU*: *cdcl_W-bj** S U* **and**

n-s-U: *no-step cdcl_W-bj U* **and**

n-s-T: *no-step cdcl_W-bj T* **and**

inv: *cdcl_W-all-struct-inv S*

shows *T ~ U*

$\langle proof \rangle$

lemma *full-cdcl_W-bj-unique-normal-form*:

assumes *full cdcl_W-bj S T* **and** *full cdcl_W-bj S U* **and**

inv: *cdcl_W-all-struct-inv S*

shows *T ~ U*

$\langle proof \rangle$

21.3 CDCL FW

inductive *cdcl_W-merge-restart* :: *'st \Rightarrow 'st \Rightarrow bool* **where**

fw-r-propagate: *propagate S S' \Rightarrow cdcl_W-merge-restart S S'* |

fw-r-conflict: *conflict S T \Rightarrow full cdcl_W-bj T U \Rightarrow cdcl_W-merge-restart S U* |

fw-r-decide: *decide S S' \Rightarrow cdcl_W-merge-restart S S'* |

fw-r-rf: *cdcl_W-rf S S' \Rightarrow cdcl_W-merge-restart S S'*

lemma *rtrancpl-cdcl_W-bj-rtrancpl-cdcl_W*:

*cdcl_W-bj** S T \Rightarrow cdcl_W** S T*

$\langle proof \rangle$

lemma *cdcl_W-merge-restart-cdcl_W*:

assumes *cdcl_W-merge-restart S T*

shows *cdcl_W** S T*

$\langle proof \rangle$

lemma *cdcl_W-merge-restart-conflicting-true-or-no-step*:

assumes *cdcl_W-merge-restart S T*

shows *conflicting T = None \vee no-step cdcl_W T*

$\langle proof \rangle$

inductive *cdcl_W-merge* :: *'st \Rightarrow 'st \Rightarrow bool* **where**

fw-propagate: *propagate S S' \Rightarrow cdcl_W-merge S S'* |

fw-conflict: *conflict S T \Rightarrow full cdcl_W-bj T U \Rightarrow cdcl_W-merge S U* |

fw-decide: *decide S S' \Rightarrow cdcl_W-merge S S'* |

fw-forget: *forget S S' \Rightarrow cdcl_W-merge S S'*

lemma *cdcl_W-merge-cdcl_W-merge-restart*:

cdcl_W-merge S T \Rightarrow cdcl_W-merge-restart S T

$\langle proof \rangle$

lemma *rtrancpl-cdcl_W-merge-trancpl-cdcl_W-merge-restart*:

*cdcl_W-merge** S T \Rightarrow cdcl_W-merge-restart** S T*

$\langle proof \rangle$

lemma *cdcl_W-merge-rtrancpl-cdcl_W*:

*cdcl_W-merge S T \Rightarrow cdcl_W** S T*

$\langle proof \rangle$

lemma *rtrancp-cdcl_W-merge-rtrancp-cdcl_W*:
 *cdcl_W-merge^{**} S T \implies cdcl_W^{**} S T*
 $\langle proof \rangle$

lemmas *rulesE* =
 skipE resolveE backtrackE propagateE conflictE decideE restartE forgetE

lemma *cdcl_W-all-struct-inv-trancp-cdcl_W-merge-trancp-cdcl_W-merge-cdcl_W-all-struct-inv*:
 assumes
 inv: cdcl_W-all-struct-inv b
 cdcl_W-merge⁺⁺ b a
 shows $(\lambda S T. \text{cdcl}_W\text{-all-struct-inv } S \wedge \text{cdcl}_W\text{-merge } S T)^{++} b a$
 $\langle proof \rangle$

lemma *backtrack-is-full1-cdcl_W-bj*:
 assumes *bt: backtrack S T and inv: cdcl_W-M-level-inv S*
 shows *full1 cdcl_W-bj S T*
 $\langle proof \rangle$

lemma *rtranc-cdcl_W-conflicting-true-cdcl_W-merge-restart*:
 assumes *cdcl_W^{**} S V and inv: cdcl_W-M-level-inv S and conflicting S = None*
 shows $(\text{cdcl}_W\text{-merge-restart}^{**} S V \wedge \text{conflicting } V = \text{None})$
 $\vee (\exists T U. \text{cdcl}_W\text{-merge-restart}^{**} S T \wedge \text{conflicting } V \neq \text{None} \wedge \text{conflict } T U \wedge \text{cdcl}_W\text{-bj}^{**} U V)$
 $\langle proof \rangle$

lemma *no-step-cdcl_W-no-step-cdcl_W-merge-restart*: *no-step cdcl_W S \implies no-step cdcl_W-merge-restart S*
 $\langle proof \rangle$

lemma *no-step-cdcl_W-merge-restart-no-step-cdcl_W*:
 assumes
 conflicting S = None and
 cdcl_W-M-level-inv S and
 no-step cdcl_W-merge-restart S
 shows *no-step cdcl_W S*
 $\langle proof \rangle$

lemma *cdcl_W-merge-restart-no-step-cdcl_W-bj*:
 assumes
 cdcl_W-merge-restart S T
 shows *no-step cdcl_W-bj T*
 $\langle proof \rangle$

lemma *rtrancp-cdcl_W-merge-restart-no-step-cdcl_W-bj*:
 assumes
 *cdcl_W-merge-restart^{**} S T and*
 conflicting S = None
 shows *no-step cdcl_W-bj T*
 $\langle proof \rangle$

If *conflicting S \neq None*, we cannot say anything.

Remark that this theorem does not say anything about well-foundedness: even if you know that one relation is well-founded, it only states that the normal forms are shared.

lemma *conflicting-true-full-cdcl_W-iff-full-cdcl_W-merge:*
assumes *conf!*: *conflicting* $S = \text{None}$ **and** *lev*: *cdcl_W-M-level-inv* S
shows $\text{full cdcl}_W S V \longleftrightarrow \text{full cdcl}_W\text{-merge-restart } S V$
 $\langle \text{proof} \rangle$

lemma *init-state-true-full-cdcl_W-iff-full-cdcl_W-merge:*
shows $\text{full cdcl}_W (\text{init-state } N) V \longleftrightarrow \text{full cdcl}_W\text{-merge-restart } (\text{init-state } N) V$
 $\langle \text{proof} \rangle$

21.4 FW with strategy

21.4.1 The intermediate step

inductive $\text{cdcl}_W\text{-s}' :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**
conflict': $\text{full1 cdcl}_W\text{-cp } S S' \Longrightarrow \text{cdcl}_W\text{-s}' S S' \mid$
decide': $\text{decide } S S' \Longrightarrow \text{no-step cdcl}_W\text{-cp } S \Longrightarrow \text{full cdcl}_W\text{-cp } S' S'' \Longrightarrow \text{cdcl}_W\text{-s}' S S'' \mid$
bj': $\text{full1 cdcl}_W\text{-bj } S S' \Longrightarrow \text{no-step cdcl}_W\text{-cp } S \Longrightarrow \text{full cdcl}_W\text{-cp } S' S'' \Longrightarrow \text{cdcl}_W\text{-s}' S S''$

inductive-cases $\text{cdcl}_W\text{-s}'E$: $\text{cdcl}_W\text{-s}' S T$

lemma *rtrancp-cdcl_W-bj-full1-cdclp-cdcl_W-stgy:*
 $\text{cdcl}_W\text{-bj}^{**} S S' \Longrightarrow \text{full cdcl}_W\text{-cp } S' S'' \Longrightarrow \text{cdcl}_W\text{-stgy}^{**} S S''$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-s'-is-rtrancp-cdcl_W-stgy:*
 $\text{cdcl}_W\text{-s}' S T \Longrightarrow \text{cdcl}_W\text{-stgy}^{**} S T$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-cdcl_W-bj-bissimulation:*
assumes
 $\text{full cdcl}_W\text{-cp } T U$ **and**
 $\text{cdcl}_W\text{-bj}^{**} T T'$ **and**
 $\text{cdcl}_W\text{-all-struct-inv } T$ **and**
 $\text{no-step cdcl}_W\text{-bj } T'$
shows $\text{full cdcl}_W\text{-cp } T' U$
 $\vee (\exists U' U''. \text{full cdcl}_W\text{-cp } T' U'' \wedge \text{full1 cdcl}_W\text{-bj } U U' \wedge \text{full cdcl}_W\text{-cp } U' U''$
 $\wedge \text{cdcl}_W\text{-s}'^{**} U U'')$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-cp-cdcl_W-bj-bissimulation':*
assumes
 $\text{full cdcl}_W\text{-cp } T U$ **and**
 $\text{cdcl}_W\text{-bj}^{**} T T'$ **and**
 $\text{cdcl}_W\text{-all-struct-inv } T$ **and**
 $\text{no-step cdcl}_W\text{-bj } T'$
shows $\text{full cdcl}_W\text{-cp } T' U$
 $\vee (\exists U'. \text{full1 cdcl}_W\text{-bj } U U' \wedge (\forall U''. \text{full cdcl}_W\text{-cp } U' U'' \longrightarrow \text{full cdcl}_W\text{-cp } T' U''$
 $\wedge \text{cdcl}_W\text{-s}'^{**} U U'))$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-cdcl_W-s'-connected:*
assumes $\text{cdcl}_W\text{-stgy } S U$ **and** $\text{cdcl}_W\text{-all-struct-inv } S$
shows $\text{cdcl}_W\text{-s}' S U$
 $\vee (\exists U'. \text{full1 cdcl}_W\text{-bj } U U' \wedge (\forall U''. \text{full cdcl}_W\text{-cp } U' U'' \longrightarrow \text{cdcl}_W\text{-s}' S U''))$
 $\langle \text{proof} \rangle$

lemma *cdcl_W-stgy-cdcl_W-s'-connected'*:

assumes *cdcl_W-stgy S U* **and** *cdcl_W-all-struct-inv S*

shows *cdcl_W-s' S U*

$\vee (\exists U' U''. \text{cdcl}_W\text{-s}' S U'' \wedge \text{full1 } \text{cdcl}_W\text{-bj } U U' \wedge \text{full } \text{cdcl}_W\text{-cp } U' U'')$

<proof>

lemma *cdcl_W-stgy-cdcl_W-s'-no-step*:

assumes *cdcl_W-stgy S U* **and** *cdcl_W-all-struct-inv S* **and** *no-step cdcl_W-bj U*

shows *cdcl_W-s' S U*

<proof>

lemma *rtrancpl-cdcl_W-stgy-connected-to-rtrancpl-cdcl_W-s'*:

assumes *cdcl_W-stgy** S U* **and** *inv: cdcl_W-M-level-inv S*

shows *cdcl_W-s'*** S U* $\vee (\exists T. \text{cdcl}_W\text{-s}'*** S T \wedge \text{cdcl}_W\text{-bj}^{++} T U \wedge \text{conflicting } U \neq \text{None})$

<proof>

lemma *n-step-cdcl_W-stgy-iff-no-step-cdcl_W-cl-cdcl_W-o*:

assumes *inv: cdcl_W-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>

lemma *cdcl_W-s'-trancpl-cdcl_W*:

cdcl_W-s' S S' \implies cdcl_W⁺⁺ S S'

<proof>

lemma *trancpl-cdcl_W-s'-trancpl-cdcl_W*:

cdcl_W-s'⁺⁺ S S' \implies cdcl_W⁺⁺ S S'

<proof>

lemma *rtrancpl-cdcl_W-s'-rtrancpl-cdcl_W*:

*cdcl_W-s'*** S S' \implies cdcl_W^{**} S S'*

<proof>

lemma *full-cdcl_W-stgy-iff-full-cdcl_W-s'*:

assumes *inv: cdcl_W-all-struct-inv S*

shows *full cdcl_W-stgy S T* \longleftrightarrow *full cdcl_W-s' S T* (**is** *?S* \longleftrightarrow *?S'*)

<proof>

lemma *conflict-step-cdcl_W-stgy-step*:

assumes

conflict S T

cdcl_W-all-struct-inv S

shows $\exists T. \text{cdcl}_W\text{-stgy } S T$

<proof>

lemma *decide-step-cdcl_W-stgy-step*:

assumes

decide S T

cdcl_W-all-struct-inv S

shows $\exists T. \text{cdcl}_W\text{-stgy } S T$

<proof>

lemma *rtrancpl-cdcl_W-cp-conflicting-Some*:

*cdcl_W-cp** S T \implies conflicting S = Some D \implies S = T*

<proof>

inductive $cdcl_W\text{-merge-cp} :: 'st \Rightarrow 'st \Rightarrow \text{bool}$ **where**
conflict': $\text{conflict } S \ T \Longrightarrow \text{full } cdcl_W\text{-bj } T \ U \Longrightarrow cdcl_W\text{-merge-cp } S \ U \mid$
propagate': $\text{propagate}^{++} S \ S' \Longrightarrow cdcl_W\text{-merge-cp } S \ S'$

lemma $cdcl_W\text{-merge-restart-cases}$ [consumes 1, case-names *conflict propagate*]:

assumes
 $cdcl_W\text{-merge-cp } S \ U$ **and**
 $\bigwedge T. \text{conflict } S \ T \Longrightarrow \text{full } cdcl_W\text{-bj } T \ U \Longrightarrow P$ **and**
 $\text{propagate}^{++} S \ U \Longrightarrow P$
shows P
 $\langle \text{proof} \rangle$

lemma $cdcl_W\text{-merge-cp-tranclp-cdcl}_W\text{-merge}$:
 $cdcl_W\text{-merge-cp } S \ T \Longrightarrow cdcl_W\text{-merge}^{++} S \ T$
 $\langle \text{proof} \rangle$

lemma $rtranclp\text{-cdcl}_W\text{-merge-cp-rtranclp-cdcl}_W$:
 $cdcl_W\text{-merge-cp}^{**} S \ T \Longrightarrow cdcl_W^{**} S \ T$
 $\langle \text{proof} \rangle$

lemma $full1\text{-cdcl}_W\text{-bj-no-step-cdcl}_W\text{-bj}$:
 $full1 \ cdcl_W\text{-bj } S \ T \Longrightarrow \text{no-step } cdcl_W\text{-cp } S$
 $\langle \text{proof} \rangle$

21.4.2 Full Transformation

inductive $cdcl_W\text{-s'-without-decide}$ **where**
conflict'-without-decide[intro]: $full1 \ cdcl_W\text{-cp } S \ S' \Longrightarrow cdcl_W\text{-s'-without-decide } S \ S' \mid$
bj'-without-decide[intro]: $full1 \ cdcl_W\text{-bj } S \ S' \Longrightarrow \text{no-step } cdcl_W\text{-cp } S \Longrightarrow \text{full } cdcl_W\text{-cp } S' \ S'' \Longrightarrow cdcl_W\text{-s'-without-decide } S \ S''$

lemma $rtranclp\text{-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W$:
 $cdcl_W\text{-s'-without-decide}^{**} S \ T \Longrightarrow cdcl_W^{**} S \ T$
 $\langle \text{proof} \rangle$

lemma $rtranclp\text{-cdcl}_W\text{-s'-without-decide-rtranclp-cdcl}_W\text{-s'}$:
 $cdcl_W\text{-s'-without-decide}^{**} S \ T \Longrightarrow cdcl_W\text{-s'}^{**} S \ T$
 $\langle \text{proof} \rangle$

lemma $rtranclp\text{-cdcl}_W\text{-merge-cp-is-rtranclp-cdcl}_W\text{-s'-without-decide}$:

assumes
 $cdcl_W\text{-merge-cp}^{**} S \ V$
 $\text{conflicting } S = \text{None}$
shows
 $(cdcl_W\text{-s'-without-decide}^{**} S \ V)$
 $\vee (\exists T. \text{cdcl}_W\text{-s'-without-decide}^{**} S \ T \wedge \text{propagate}^{++} T \ V)$
 $\vee (\exists T \ U. \text{cdcl}_W\text{-s'-without-decide}^{**} S \ T \wedge \text{full1 } cdcl_W\text{-bj } T \ U \wedge \text{propagate}^{**} U \ V)$
 $\langle \text{proof} \rangle$

lemma $rtranclp\text{-cdcl}_W\text{-s'-without-decide-is-rtranclp-cdcl}_W\text{-merge-cp}$:

assumes
 $cdcl_W\text{-s'-without-decide}^{**} S \ V$ **and**
 $\text{confl: conflicting } S = \text{None}$
shows
 $(cdcl_W\text{-merge-cp}^{**} S \ V \wedge \text{conflicting } V = \text{None})$

$\vee (\text{cdcl}_W\text{-merge-cp}^{**} S V \wedge \text{conflicting } V \neq \text{None} \wedge \text{no-step cdcl}_W\text{-cp } V \wedge \text{no-step cdcl}_W\text{-bj } V)$
 $\vee (\exists T. \text{cdcl}_W\text{-merge-cp}^{**} S T \wedge \text{conflict } T V)$
 <proof>

lemma *no-step-cdcl_W-s'-no-ste-cdcl_W-merge-cp:*

assumes
cdcl_W-all-struct-inv S
conflicting S = None
no-step cdcl_W-s' S
shows *no-step cdcl_W-merge-cp S*
 <proof>

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
conft: conflicting S = None and
inv: cdcl_W-M-level-inv S and
n-s: no-step cdcl_W-merge-cp S
shows *no-step cdcl_W-s'-without-decide S*
 <proof>

lemma *conflicting-true-no-step-s'-without-decide-no-step-cdcl_W-merge-cp:*

assumes
inv: cdcl_W-all-struct-inv S and
n-s: no-step cdcl_W-s'-without-decide S
shows *no-step cdcl_W-merge-cp S*
 <proof>

lemma *no-step-cdcl_W-merge-cp-no-step-cdcl_W-cp:*

no-step cdcl_W-merge-cp S \implies cdcl_W-M-level-inv S \implies no-step cdcl_W-cp S
 <proof>

lemma *conflicting-not-true-rtrancp-cdcl_W-merge-cp-no-step-cdcl_W-bj:*

assumes
conflicting S = None and
*cdcl_W-merge-cp^{**} S T*
shows *no-step cdcl_W-bj T*
 <proof>

lemma *conflicting-true-full-cdcl_W-merge-cp-iff-full-cdcl_W-s'-without-decode:*

assumes
conft: conflicting S = None and
inv: cdcl_W-all-struct-inv S
shows
full cdcl_W-merge-cp S V \longleftrightarrow full cdcl_W-s'-without-decide S V (is ?fw \longleftrightarrow ?s')
 <proof>

lemma *conflicting-true-full1-cdcl_W-merge-cp-iff-full1-cdcl_W-s'-without-decode:*

assumes
conft: conflicting S = None and
inv: cdcl_W-all-struct-inv S
shows
full1 cdcl_W-merge-cp S V \longleftrightarrow full1 cdcl_W-s'-without-decide S V
 <proof>

lemma *conflicting-true-full1-cdcl_W-merge-cp-imp-full1-cdcl_W-s'-without-decode:*

assumes

fw: full1 cdcl_W-merge-cp S V and

inv: cdcl_W-all-struct-inv S

shows

full1 cdcl_W-s'-without-decode S V

<proof>

inductive *cdcl_W-merge-stgy where*

fw-s-cp[intro]: full1 cdcl_W-merge-cp S T \implies cdcl_W-merge-stgy S T |

fw-s-decide[intro]: decide S T \implies no-step cdcl_W-merge-cp S \implies full cdcl_W-merge-cp T U \implies cdcl_W-merge-stgy S U

lemma *cdcl_W-merge-stgy-tranclp-cdcl_W-merge:*

assumes *fw: cdcl_W-merge-stgy S T*

shows *cdcl_W-merge⁺⁺ S T*

<proof>

lemma *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W-merge:*

assumes *fw: cdcl_W-merge-stgy^{**} S T*

shows *cdcl_W-merge^{**} S T*

<proof>

lemma *cdcl_W-merge-stgy-rtranclp-cdcl_W:*

*cdcl_W-merge-stgy S T \implies cdcl_W^{**} S T*

<proof>

lemma *rtranclp-cdcl_W-merge-stgy-rtranclp-cdcl_W:*

*cdcl_W-merge-stgy^{**} S T \implies cdcl_W^{**} S T*

<proof>

lemma *cdcl_W-merge-stgy-cases[consumes 1, case-names fw-s-cp fw-s-decide]:*

assumes

cdcl_W-merge-stgy S U

full1 cdcl_W-merge-cp S U \implies P

$\bigwedge T. \text{decide } S \ T \implies \text{no-step } \text{cdcl}_W\text{-merge-cp } S \implies \text{full } \text{cdcl}_W\text{-merge-cp } T \ U \implies P$

shows *P*

<proof>

inductive *cdcl_W-s'-w :: 'st \Rightarrow 'st \Rightarrow bool where*

conflict': full1 cdcl_W-s'-without-decode S S' \implies cdcl_W-s'-w S S' |

decide': decide S S' \implies no-step cdcl_W-s'-without-decode S \implies full cdcl_W-s'-without-decode S' S'' \implies cdcl_W-s'-w S S''

lemma *cdcl_W-s'-w-rtranclp-cdcl_W:*

*cdcl_W-s'-w S T \implies cdcl_W^{**} S T*

<proof>

lemma *rtranclp-cdcl_W-s'-w-rtranclp-cdcl_W:*

*cdcl_W-s'-w^{**} S T \implies cdcl_W^{**} S T*

<proof>

lemma *no-step-cdcl_W-cp-no-step-cdcl_W-s'-without-decode:*

assumes *no-step cdcl_W-cp S and conflicting S = None and inv: cdcl_W-M-level-inv S*

shows *no-step cdcl_W-s'-without-decode S*

$\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-cp:*
cdcl_W-all-struct-inv S \implies no-step cdcl_W-s'-without-decide S \implies no-step cdcl_W-cp S
 $\langle \text{proof} \rangle$

lemma *after-cdcl_W-s'-w-no-step-cdcl_W-cp:*
assumes *cdcl_W-s'-w S T and cdcl_W-all-struct-inv S*
shows *no-step cdcl_W-cp T*
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-s'-w-no-step-cdcl_W-cp-or-eq:*
assumes *cdcl_W-s'-w** S T and cdcl_W-all-struct-inv S*
shows *S = T \vee no-step cdcl_W-cp T*
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-merge-stgy'-no-step-cdcl_W-cp-or-eq:*
assumes *cdcl_W-merge-stgy** S T and inv: cdcl_W-all-struct-inv S*
shows *S = T \vee no-step cdcl_W-cp T*
 $\langle \text{proof} \rangle$

lemma *no-step-cdcl_W-s'-without-decide-no-step-cdcl_W-bj:*
assumes *no-step cdcl_W-s'-without-decide S and inv: cdcl_W-all-struct-inv S*
shows *no-step cdcl_W-bj S*
 $\langle \text{proof} \rangle$

lemma *cdcl_W-s'-w-no-step-cdcl_W-bj:*
assumes *cdcl_W-s'-w S T and cdcl_W-all-struct-inv S*
shows *no-step cdcl_W-bj T*
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-s'-w-no-step-cdcl_W-bj-or-eq:*
assumes *cdcl_W-s'-w** S T and cdcl_W-all-struct-inv S*
shows *S = T \vee no-step cdcl_W-bj T*
 $\langle \text{proof} \rangle$

lemma *rtrancp-cdcl_W-s'-no-step-cdcl_W-s'-without-decide-decomp-into-cdcl_W-merge:*
assumes
 *cdcl_W-s'*** R V and*
 conflicting R = None and
 inv: cdcl_W-all-struct-inv R
shows *(cdcl_W-merge-stgy** R V \wedge conflicting V = None)*
 *\vee (cdcl_W-merge-stgy** R V \wedge conflicting V \neq None \wedge no-step cdcl_W-bj V)*
 *\vee ($\exists S T U. \text{cdcl}_W\text{-merge-stgy}^{**} R S \wedge \text{no-step cdcl}_W\text{-merge-cp } S \wedge \text{decide } S T$*
 *$\wedge \text{cdcl}_W\text{-merge-cp}^{**} T U \wedge \text{conflict } U V$)*
 *\vee ($\exists S T. \text{cdcl}_W\text{-merge-stgy}^{**} R S \wedge \text{no-step cdcl}_W\text{-merge-cp } S \wedge \text{decide } S T$*

$\wedge \text{cdcl}_W\text{-merge-cp}^{**} T V$
 $\wedge \text{conflicting } V = \text{None})$
 $\vee (\text{cdcl}_W\text{-merge-cp}^{**} R V \wedge \text{conflicting } V = \text{None})$
 $\vee (\exists U. \text{cdcl}_W\text{-merge-cp}^{**} R U \wedge \text{conflict } U V)$
 $\langle \text{proof} \rangle$

lemma *decide-rtrancpl-cdcl_W-s'-rtrancpl-cdcl_W-s'*:

assumes
dec: decide S T and
*cdcl_W-s'^{**} T U and*
n-s-S: no-step cdcl_W-cp S and
no-step cdcl_W-cp U
shows *cdcl_W-s'^{**} S U*
 $\langle \text{proof} \rangle$

lemma *rtrancpl-cdcl_W-merge-stgy-rtrancpl-cdcl_W-s'*:

assumes
*cdcl_W-merge-stgy^{**} R V and*
inv: cdcl_W-all-struct-inv R
shows *cdcl_W-s'^{**} R V*
 $\langle \text{proof} \rangle$

lemma *rtrancpl-cdcl_W-merge-stgy-distinct-mset-clauses*:

assumes *invR: cdcl_W-all-struct-inv R and*
*st: cdcl_W-merge-stgy^{**} R S and*
dist: distinct-mset (clauses R) and
R: trail R = []
shows *distinct-mset (clauses S)*
 $\langle \text{proof} \rangle$

lemma *no-step-cdcl_W-s'-no-step-cdcl_W-merge-stgy*:

assumes
inv: cdcl_W-all-struct-inv R and s': no-step cdcl_W-s' R
shows *no-step cdcl_W-merge-stgy R*
 $\langle \text{proof} \rangle$
end

21.4.3 Termination and full Equivalence

We will discharge the assumption later using NOT's proof of termination.

locale *conflict-driven-clause-learning_W-termination* =

conflict-driven-clause-learning_W +
assumes *wf-cdcl_W-merge-inv: wf {(T, S). cdcl_W-all-struct-inv S \wedge cdcl_W-merge S T}*
begin

lemma *wf-trancpl-cdcl_W-merge: wf {(T, S). cdcl_W-all-struct-inv S \wedge cdcl_W-merge⁺⁺ S T}*
 $\langle \text{proof} \rangle$

lemma *wf-cdcl_W-merge-cp*:

wf {(T, S). cdcl_W-all-struct-inv S \wedge cdcl_W-merge-cp S T}
 $\langle \text{proof} \rangle$

lemma *wf-cdcl_W-merge-stgy*:

wf {(T, S). cdcl_W-all-struct-inv S \wedge cdcl_W-merge-stgy S T}
 $\langle \text{proof} \rangle$

lemma *cdcl_W-merge-cp-obtain-normal-form*:
assumes *inv*: *cdcl_W-all-struct-inv* *R*
obtains *S* **where** *full cdcl_W-merge-cp* *R S*
<proof>

lemma *no-step-cdcl_W-merge-stgy-no-step-cdcl_W-s'*:
assumes
inv: *cdcl_W-all-struct-inv* *R* **and**
confl: *conflicting* *R* = *None* **and**
n-s: *no-step cdcl_W-merge-stgy* *R*
shows *no-step cdcl_W-s'* *R*
<proof>

lemma *rtrancp-cdcl_W-merge-cp-no-step-cdcl_W-bj*:
assumes *conflicting* *R* = *None* **and** *cdcl_W-merge-cp*^{**} *R S*
shows *no-step cdcl_W-bj* *S*
<proof>

lemma *rtrancp-cdcl_W-merge-stgy-no-step-cdcl_W-bj*:
assumes *confl*: *conflicting* *R* = *None* **and** *cdcl_W-merge-stgy*^{**} *R S*
shows *no-step cdcl_W-bj* *S*
<proof>

end

end

theory *CDCL-W-Restart*
imports *CDCL-W-Merge*
begin

21.5 Adding Restarts

locale *cdcl_W-restart* =
conflict-driven-clause-learning_W
— functions for clauses:
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:
mset-ccls union-ccls insert-ccls remove-clit

— conversion
ccls-of-cls cls-of-ccls

— functions for the state:
— access functions:
trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
— changing state:
cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— get state:
init-state
restart-state
for

*mset-cl*s :: 'cls \Rightarrow 'v clause **and**
*insert-cl*s :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**
remove-lit :: 'v literal \Rightarrow 'cls \Rightarrow 'cls **and**

*mset-cl*ss :: 'clss \Rightarrow 'v clauses **and**
*union-cl*ss :: 'clss \Rightarrow 'clss \Rightarrow 'clss **and**
*in-cl*ss :: 'cls \Rightarrow 'clss \Rightarrow bool **and**
*insert-cl*ss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**
*remove-from-cl*ss :: 'cls \Rightarrow 'clss \Rightarrow 'clss **and**

*mset-c*cls :: 'ccls \Rightarrow 'v clause **and**
*union-c*cls :: 'ccls \Rightarrow 'ccls \Rightarrow 'ccls **and**
*insert-c*cls :: 'v literal \Rightarrow 'ccls \Rightarrow 'ccls **and**
remove-clit :: 'v literal \Rightarrow 'ccls \Rightarrow 'ccls **and**

*c*cls-of-cls :: 'cls \Rightarrow 'ccls **and**
*cls-of-c*cls :: 'ccls \Rightarrow 'cls **and**

trail :: 'st \Rightarrow ('v, nat, 'v clause) marked-lits **and**
hd-raw-trail :: 'st \Rightarrow ('v, nat, 'cls) marked-lit **and**
*raw-init-cl*ss :: 'st \Rightarrow 'clss **and**
*raw-learned-cl*ss :: 'st \Rightarrow 'clss **and**
backtrack-lvl :: 'st \Rightarrow nat **and**
raw-conflicting :: 'st \Rightarrow 'ccls option **and**

cons-trail :: ('v, nat, 'cls) marked-lit \Rightarrow 'st \Rightarrow 'st **and**
tl-trail :: 'st \Rightarrow 'st **and**
*add-init-cl*s :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
*add-learned-cl*s :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
*remove-cl*s :: 'cls \Rightarrow 'st \Rightarrow 'st **and**
update-backtrack-lvl :: nat \Rightarrow 'st \Rightarrow 'st **and**
update-conflicting :: 'ccls option \Rightarrow 'st \Rightarrow 'st **and**

init-state :: 'clss \Rightarrow 'st **and**
restart-state :: 'st \Rightarrow 'st +

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:

$(\text{cdcl}_W\text{-merge-stgy} \sim (\text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss } S)))) S T$
 $\implies \text{card} (\text{set-mset} (\text{learned-clss } T)) - \text{card} (\text{set-mset} (\text{learned-clss } S)) > f n$
 $\implies \text{restart } T U \implies \text{cdcl}_W\text{-merge-with-restart } (S, n) (U, \text{Suc } n) \mid$

restart-full: *full1 cdcl_W-merge-stgy* *S T* $\implies \text{cdcl}_W\text{-merge-with-restart } (S, n) (T, \text{Suc } n)$

lemma *cdcl_W-merge-with-restart* *S T* $\implies \text{cdcl}_W\text{-merge-restart}^{**} (\text{fst } S) (\text{fst } T)$

<proof>

lemma *cdcl_W-merge-with-restart-rtrancpl-cdcl_W*:

cdcl_W-merge-with-restart *S T* $\implies \text{cdcl}_W^{**} (\text{fst } S) (\text{fst } T)$

<proof>

lemma *cdcl_W-merge-with-restart-increasing-number:*
cdcl_W-merge-with-restart S T \implies snd T = 1 + snd S
<proof>

lemma *full1 cdcl_W-merge-stgy S T \implies cdcl_W-merge-with-restart (S, n) (T, Suc n)*
<proof>

lemma *cdcl_W-all-struct-inv-learned-clss-bound:*
assumes *inv: cdcl_W-all-struct-inv S*
shows *set-mset (learned-clss S) \subseteq simple-clss (atms-of-mm (init-clss S))*
<proof>

lemma *cdcl_W-merge-with-restart-init-clss:*
cdcl_W-merge-with-restart S T \implies cdcl_W-M-level-inv (fst S) \implies
init-clss (fst S) = init-clss (fst T)
<proof>

lemma
wf {(T, S). cdcl_W-all-struct-inv (fst S) \wedge cdcl_W-merge-with-restart S T}
<proof>

lemma *cdcl_W-merge-with-restart-distinct-mset-clauses:*
assumes *invR: cdcl_W-all-struct-inv (fst R) and*
st: cdcl_W-merge-with-restart R S and
dist: distinct-mset (clauses (fst R)) and
R: trail (fst R) = []
shows *distinct-mset (clauses (fst S))*
<proof>

inductive *cdcl_W-with-restart where*

restart-step:

(cdcl_W-stgy \sim (card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)))) S T \implies
card (set-mset (learned-clss T)) - card (set-mset (learned-clss S)) > f n \implies
restart T U \implies

cdcl_W-with-restart (S, n) (U, Suc n) |

restart-full: full1 cdcl_W-stgy S T \implies cdcl_W-with-restart (S, n) (T, Suc n)

lemma *cdcl_W-with-restart-rtrancp-cdcl_W:*
*cdcl_W-with-restart S T \implies cdcl_W** (fst S) (fst T)*
<proof>

lemma *cdcl_W-with-restart-increasing-number:*
cdcl_W-with-restart S T \implies snd T = 1 + snd S
<proof>

lemma *full1 cdcl_W-stgy S T \implies cdcl_W-with-restart (S, n) (T, Suc n)*
<proof>

lemma *cdcl_W-with-restart-init-clss:*
cdcl_W-with-restart S T \implies cdcl_W-M-level-inv (fst S) \implies init-clss (fst S) = init-clss (fst T)
<proof>

lemma
wf {(T, S). cdcl_W-all-struct-inv (fst S) \wedge cdcl_W-with-restart S T}

⟨proof⟩

lemma *cdcl_W-with-restart-distinct-mset-clauses*:
assumes *invR*: *cdcl_W-all-struct-inv* (*fst R*) **and**
st: *cdcl_W-with-restart* *R S* **and**
dist: *distinct-mset* (*clauses* (*fst R*)) **and**
R: *trail* (*fst R*) = []
shows *distinct-mset* (*clauses* (*fst S*))
 ⟨proof⟩
end

locale *luby-sequence* =
fixes *ur* :: *nat*
assumes *ur* > 0
begin

lemma *exists-luby-decomp*:
fixes *i* :: *nat*
shows $\exists k :: \text{nat}. (2^{\wedge} (k - 1) \leq i \wedge i < 2^{\wedge} k - 1) \vee i = 2^{\wedge} k - 1$
 ⟨proof⟩

Luby sequences are defined by:

- $2^k - 1$, if $i = (2 :: 'a)^k - (1 :: 'a)$
- *luby-sequence-core* ($i - 2^{k-1} + 1$), if $(2 :: 'a)^{k-1} \leq i$ and $i \leq (2 :: 'a)^k - (1 :: 'a)$

Then the sequence is then scaled by a constant unit run (called *ur* here), strictly positive.

function *luby-sequence-core* :: *nat* \Rightarrow *nat* **where**
luby-sequence-core *i* =
 (if $\exists k. i = 2^{\wedge} k - 1$
 then $2^{\wedge} ((\text{SOME } k. i = 2^{\wedge} k - 1) - 1)$
 else *luby-sequence-core* ($i - 2^{\wedge} ((\text{SOME } k. 2^{\wedge} (k-1) \leq i \wedge i < 2^{\wedge} k - 1) - 1) + 1$))
 ⟨proof⟩
termination
 ⟨proof⟩

function *natlog2* :: *nat* \Rightarrow *nat* **where**
natlog2 *n* = (if *n* = 0 then 0 else 1 + *natlog2* (*n* div 2))
 ⟨proof⟩
termination ⟨proof⟩

declare *natlog2.simps*[*simp del*]

declare *luby-sequence-core.simps*[*simp del*]

lemma *two-pover-n-eq-two-power-n'-eq*:
assumes *H*: $(2 :: \text{nat})^{\wedge} (k :: \text{nat}) - 1 = 2^{\wedge} k' - 1$
shows $k' = k$
 ⟨proof⟩

lemma *luby-sequence-core-two-power-minus-one*:
luby-sequence-core ($2^{\wedge} k - 1$) = $2^{\wedge} (k-1)$ (**is** ?*L* = ?*K*)
 ⟨proof⟩

lemma *different-luby-decomposition-false:*

assumes

$H: 2^{\wedge} (k - \text{Suc } 0) \leq i$ **and**

$k': i < 2^{\wedge} k' - \text{Suc } 0$ **and**

$k-k': k > k'$

shows *False*

<proof>

lemma *luby-sequence-core-not-two-power-minus-one:*

assumes

$k-i: 2^{\wedge} (k - 1) \leq i$ **and**

$i-k: i < 2^{\wedge} k - 1$

shows *luby-sequence-core i = luby-sequence-core (i - 2^{\wedge} (k - 1) + 1)*

<proof>

lemma *unbounded-luby-sequence-core: unbounded luby-sequence-core*

<proof>

abbreviation *luby-sequence* :: *nat* \Rightarrow *nat* **where**

luby-sequence n \equiv *ur * luby-sequence-core n*

lemma *bounded-luby-sequence: unbounded luby-sequence*

<proof>

lemma *luby-sequence-core-0: luby-sequence-core 0 = 1*

<proof>

lemma *luby-sequence-core n \geq 1*

<proof>

end

locale *luby-sequence-restart* =

luby-sequence ur +

conflict-driven-clause-learning_W — functions for clauses:

mset-cls insert-cls remove-lit

mset-clss union-clss in-clss insert-clss remove-from-clss

— functions for the conflicting clause:

mset-ccls union-ccls insert-ccls remove-clit

— conversion

ccls-of-cls cls-of-ccls

— functions for the state:

— access functions:

trail hd-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting

— changing state:

cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl

update-conflicting

— get state:

init-state

restart-state

for

ur :: *nat* **and**

```

mset-cls:: 'cls ⇒ 'v clause and
insert-cls :: 'v literal ⇒ 'cls ⇒ 'cls and
remove-lit :: 'v literal ⇒ 'cls ⇒ 'cls and

mset-clss:: 'clss ⇒ 'v clauses and
union-clss :: 'clss ⇒ 'clss ⇒ 'clss and
in-clss :: 'cls ⇒ 'clss ⇒ bool and
insert-clss :: 'cls ⇒ 'clss ⇒ 'clss and
remove-from-clss :: 'cls ⇒ 'clss ⇒ 'clss and

mset-ccls:: 'ccls ⇒ 'v clause and
union-ccls :: 'ccls ⇒ 'ccls ⇒ 'ccls and
insert-ccls :: 'v literal ⇒ 'ccls ⇒ 'ccls and
remove-clit :: 'v literal ⇒ 'ccls ⇒ 'ccls and

ccls-of-cls :: 'cls ⇒ 'ccls and
cls-of-ccls :: 'ccls ⇒ 'cls and

trail :: 'st ⇒ ('v, nat, 'v clause) marked-lits and
hd-raw-trail :: 'st ⇒ ('v, nat, 'cls) marked-lit and
raw-init-clss :: 'st ⇒ 'clss and
raw-learned-clss :: 'st ⇒ 'clss and
backtrack-lvl :: 'st ⇒ nat and
raw-conflicting :: 'st ⇒ 'ccls option and

cons-trail :: ('v, nat, 'cls) marked-lit ⇒ 'st ⇒ 'st and
tl-trail :: 'st ⇒ 'st and
add-init-cls :: 'cls ⇒ 'st ⇒ 'st and
add-learned-cls :: 'cls ⇒ 'st ⇒ 'st and
remove-cls :: 'cls ⇒ 'st ⇒ 'st and
update-backtrack-lvl :: nat ⇒ 'st ⇒ 'st and
update-conflicting :: 'ccls option ⇒ 'st ⇒ 'st and

init-state :: 'clss ⇒ 'st and
restart-state :: 'st ⇒ 'st
begin

sublocale cdclW-restart - - - - - luby-sequence
⟨proof⟩

end
end
theory CDCL-WNOT
imports CDCL-NOT CDCL-W-Termination CDCL-W-Merge
begin

```

22 Link between Weidenbach's and NOT's CDCL

22.1 Inclusion of the states

```

declare upt.simps(2)[simp del]

fun convert-marked-lit-from-W where
  convert-marked-lit-from-W (Propagated L -) = Propagated L () |
  convert-marked-lit-from-W (Marked L -) = Marked L ()

```

abbreviation *convert-trail-from-W* ::
 ('v, 'lvl, 'a) marked-lit list
 \Rightarrow ('v, unit, unit) marked-lit list **where**
convert-trail-from-W \equiv map *convert-marked-lit-from-W*

lemma *lits-of-l-convert-trail-from-W[simp]*:
lits-of-l (*convert-trail-from-W* M) = *lits-of-l* M
 <proof>

lemma *lit-of-convert-trail-from-W[simp]*:
lit-of (*convert-marked-lit-from-W* L) = *lit-of* L
 <proof>

lemma *no-dup-convert-from-W[simp]*:
no-dup (*convert-trail-from-W* M) \longleftrightarrow *no-dup* M
 <proof>

lemma *convert-trail-from-W-true-annots[simp]*:
convert-trail-from-W M \models_{as} C \longleftrightarrow M \models_{as} C
 <proof>

lemma *defined-lit-convert-trail-from-W[simp]*:
defined-lit (*convert-trail-from-W* S) L \longleftrightarrow *defined-lit* S L
 <proof>

The values 0 and {#} are dummy values.

consts *dummy-cls* :: 'cls
fun *convert-marked-lit-from-NOT*
 :: ('a, 'e, 'b) marked-lit \Rightarrow ('a, nat, 'cls) marked-lit **where**
convert-marked-lit-from-NOT (*Propagated* L -) = *Propagated* L *dummy-cls* |
convert-marked-lit-from-NOT (*Marked* L -) = *Marked* L 0

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT \equiv map *convert-marked-lit-from-NOT*

lemma *undefined-lit-convert-trail-from-NOT[simp]*:
undefined-lit (*convert-trail-from-NOT* F) L \longleftrightarrow *undefined-lit* F L
 <proof>

lemma *lits-of-l-convert-trail-from-NOT*:
lits-of-l (*convert-trail-from-NOT* F) = *lits-of-l* F
 <proof>

lemma *convert-trail-from-W-from-NOT[simp]*:
convert-trail-from-W (*convert-trail-from-NOT* M) = M
 <proof>

lemma *convert-trail-from-W-convert-lit-from-NOT[simp]*:
convert-marked-lit-from-W (*convert-marked-lit-from-NOT* L) = L
 <proof>

abbreviation *trail_{NOT}* **where**
trail_{NOT} S \equiv *convert-trail-from-W* (*fst* S)

lemma *undefined-lit-convert-trail-from-W*[*iff*]:
 $\text{undefined-lit } (\text{convert-trail-from-W } M) L \longleftrightarrow \text{undefined-lit } M L$
 ⟨*proof*⟩

lemma *lit-of-convert-marked-lit-from-NOT*[*iff*]:
 $\text{lit-of } (\text{convert-marked-lit-from-NOT } L) = \text{lit-of } L$
 ⟨*proof*⟩

sublocale $\text{state}_W \subseteq \text{dpll-state-ops}$
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
 $\lambda S. \text{convert-trail-from-W } (\text{trail } S)$
raw-clauses
 $\lambda L S. \text{cons-trail } (\text{convert-marked-lit-from-NOT } L) S$
 $\lambda S. \text{tl-trail } S$
 $\lambda C S. \text{add-learned-cls } C S$
 $\lambda C S. \text{remove-cls } C S$
 ⟨*proof*⟩

context state_W

begin

lemma *convert-marked-lit-from-W-convert-marked-lit-from-NOT*[*simp*]:
 $\text{convert-marked-lit-from-W } (\text{mset-of-mlit } (\text{convert-marked-lit-from-NOT } L)) = L$
 ⟨*proof*⟩

end

sublocale $\text{state}_W \subseteq \text{dpll-state}$
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
 $\lambda S. \text{convert-trail-from-W } (\text{trail } S)$
raw-clauses
 $\lambda L S. \text{cons-trail } (\text{convert-marked-lit-from-NOT } L) S$
 $\lambda S. \text{tl-trail } S$
 $\lambda C S. \text{add-learned-cls } C S$
 $\lambda C S. \text{remove-cls } C S$
 ⟨*proof*⟩

context state_W

begin

declare $\text{state-simp}_{\text{NOT}}[\text{simp del}]$

end

sublocale *conflict-driven-clause-learning* $_W \subseteq \text{cdcl}_{\text{NOT-merge-bj-learn-ops}}$
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
 $\lambda S. \text{convert-trail-from-W } (\text{trail } S)$
raw-clauses
 $\lambda L S. \text{cons-trail } (\text{convert-marked-lit-from-NOT } L) S$
 $\lambda S. \text{tl-trail } S$
 $\lambda C S. \text{add-learned-cls } C S$
 $\lambda C S. \text{remove-cls } C S$
 $\lambda - . \text{True}$

λ - S . raw-conflicting $S = \text{None}$
 $\lambda C C' L' S T$. backjump-l-cond $C C' L' S T$
 $\wedge \text{distinct-mset } (C' + \{\#L'\#\}) \wedge \neg \text{tautology } (C' + \{\#L'\#\})$
 $\langle \text{proof} \rangle$

thm $\text{cdcl}_{\text{NOT-merge-bj-learn-proxy}}.\text{axioms}$

sublocale $\text{conflict-driven-clause-learning}_W \subseteq \text{cdcl}_{\text{NOT-merge-bj-learn-proxy}}$

$\text{mset-cls insert-cls remove-lit}$
 $\text{mset-clss union-clss in-clss insert-clss remove-from-clss}$
 λS . convert-trail-from- W (trail S)
 raw-clauses
 $\lambda L S$. cons-trail (convert-marked-lit-from-NOT L) S
 λS . tl-trail S
 $\lambda C S$. add-learned-cls $C S$
 $\lambda C S$. remove-cls $C S$

λ - -. True
 λ - S . raw-conflicting $S = \text{None}$
 backjump-l-cond
 inv_{NOT}
 $\langle \text{proof} \rangle$

sublocale $\text{conflict-driven-clause-learning}_W \subseteq \text{cdcl}_{\text{NOT-merge-bj-learn-proxy2}} \text{ - - - - -}$

λS . convert-trail-from- W (trail S)
 raw-clauses
 $\lambda L S$. cons-trail (convert-marked-lit-from-NOT L) S
 λS . tl-trail S
 $\lambda C S$. add-learned-cls $C S$
 $\lambda C S$. remove-cls $C S$
 λ - -. True
 λ - S . raw-conflicting $S = \text{None}$ $\text{backjump-l-cond inv}_{\text{NOT}}$
 $\langle \text{proof} \rangle$

sublocale $\text{conflict-driven-clause-learning}_W \subseteq \text{cdcl}_{\text{NOT-merge-bj-learn}} \text{ - - - - -}$

λS . convert-trail-from- W (trail S)
 raw-clauses
 $\lambda L S$. cons-trail (convert-marked-lit-from-NOT L) S
 λS . tl-trail S
 $\lambda C S$. add-learned-cls $C S$
 $\lambda C S$. remove-cls $C S$
 backjump-l-cond
 λ - -. True
 λ - S . raw-conflicting $S = \text{None}$ inv_{NOT}
 $\langle \text{proof} \rangle$

context $\text{conflict-driven-clause-learning}_W$
begin

Notations are lost while proving locale inclusion:

notation $\text{state-eq}_{\text{NOT}}$ (**infix** \sim_{NOT} 50)

22.2 Additional Lemmas between NOT and W states

lemma $\text{trail}_W\text{-eq-reduce-trail-to}_{\text{NOT-eq}}$:

$\text{trail } S = \text{trail } T \implies \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F S) = \text{trail } (\text{reduce-trail-to}_{\text{NOT}} F T)$

$\langle \text{proof} \rangle$

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)

\langle proof \rangle

lemma *reduce-trail-to_{NOT}-reduce-trail-convert*:

reduce-trail-to_{NOT} C S = reduce-trail-to (convert-trail-from-NOT C) S

\langle proof \rangle

lemma *reduce-trail-to-map[simp]*:

reduce-trail-to (map f M) S = reduce-trail-to M S

\langle proof \rangle

lemma *reduce-trail-to_{NOT}-map[simp]*:

reduce-trail-to_{NOT} (map f M) S = reduce-trail-to_{NOT} M S

\langle proof \rangle

lemma *skip-or-resolve-state-change*:

assumes skip-or-resolve** S T

shows

$\exists M. \text{trail } S = M @ \text{trail } T \wedge (\forall m \in \text{set } M. \neg \text{is-marked } m)$

$\text{clauses } S = \text{clauses } T$

$\text{backtrack-lvl } S = \text{backtrack-lvl } T$

\langle proof \rangle

22.3 More lemmas conflict-propagate and backjumping

22.4 CDCL FW

lemma *cdcl_W-merge-is-cdcl_{NOT}-merged-bj-learn*:

assumes

inv: cdcl_W-all-struct-inv S **and**

cdcl_W:cdcl_W-merge S T

shows cdcl_{NOT}-merged-bj-learn S T

$\vee (\text{no-step cdcl}_W\text{-merge } T \wedge \text{conflicting } T \neq \text{None})$

\langle proof \rangle

abbreviation cdcl_{NOT}-restart **where**

cdcl_{NOT}-restart \equiv restart-ops.cdcl_{NOT}-raw-restart cdcl_{NOT} restart

lemma *cdcl_W-merge-restart-is-cdcl_{NOT}-merged-bj-learn-restart-no-step*:

assumes

inv: cdcl_W-all-struct-inv S **and**

cdcl_W:cdcl_W-merge-restart S T

shows cdcl_{NOT}-restart** S $T \vee (\text{no-step cdcl}_W\text{-merge } T \wedge \text{conflicting } T \neq \text{None})$

\langle proof \rangle

abbreviation $\mu_{FW} :: 'st \Rightarrow nat$ **where**

$\mu_{FW} S \equiv (\text{if no-step cdcl}_W\text{-merge } S \text{ then } 0 \text{ else } 1 + \mu_{CDCL} \text{'merged (set-mset (init-clss } S)) } S)$

lemma *cdcl_W-merge- μ_{FW} -decreasing*:

assumes

inv: cdcl_W-all-struct-inv S **and**

fw: cdcl_W-merge S T

shows $\mu_{FW} T < \mu_{FW} S$

<proof>

lemma *wf-cdcl_W-merge*: wf {(T, S). cdcl_W-all-struct-inv S ∧ cdcl_W-merge S T}
<proof>

sublocale *conflict-driven-clause-learning_W-termination*
<proof>

lemma *full-cdcl_W-s'-full-cdcl_W-merge-restart*:
assumes
 conflicting R = None and
 inv: cdcl_W-all-struct-inv R
shows *full cdcl_W-s' R V ⟷ full cdcl_W-merge-stgy R V (is ?s' ⟷ ?fw)*
<proof>

lemma *full-cdcl_W-stgy-full-cdcl_W-merge*:
assumes
 conflicting R = None and
 inv: cdcl_W-all-struct-inv R
shows *full cdcl_W-stgy R V ⟷ full cdcl_W-merge-stgy R V*
<proof>

lemma *full-cdcl_W-merge-stgy-final-state-conclusive'*:
fixes *S' :: 'st*
assumes *full: full cdcl_W-merge-stgy (init-state N) S'*
and *no-d: distinct-mset-mset (mset-cls N)*
shows *(conflicting S' = Some {#} ∧ unsatisfiable (set-mset (mset-cls N)))*
 ∨ (conflicting S' = None ∧ trail S' ⊨_{asm} mset-cls N ∧ satisfiable (set-mset (mset-cls N)))
<proof>
end

end
theory *CDCL-W-Incremental*
imports *CDCL-W-Termination*
begin

23 Incremental SAT solving

context *conflict-driven-clause-learning_W*
begin

This invariant holds all the invariant related to the strategy. See the structural invariant in *cdcl_W-all-struct-inv*

definition *cdcl_W-stgy-invariant* **where**
cdcl_W-stgy-invariant S ⟷
 conflict-is-false-with-level S
 ∧ no-clause-is-false S
 ∧ no-smaller-confl S
 ∧ no-clause-is-false S

lemma *cdcl_W-stgy-cdcl_W-stgy-invariant*:
assumes
 cdcl_W: cdcl_W-stgy S T and
 inv-s: cdcl_W-stgy-invariant S and
 inv: cdcl_W-all-struct-inv S

shows

$cdcl_W$ -stgy-invariant T

$\langle proof \rangle$

lemma $rtrancp$ - $cdcl_W$ -stgy- $cdcl_W$ -stgy-invariant:

assumes

$cdcl_W$: $cdcl_W$ -stgy** S T **and**

inv -s: $cdcl_W$ -stgy-invariant S **and**

inv : $cdcl_W$ -all-struct- inv S

shows

$cdcl_W$ -stgy-invariant T

$\langle proof \rangle$

abbreviation $decr$ -bt-lvl **where**

$decr$ -bt-lvl $S \equiv update$ -backtrack-lvl ($backtrack$ -lvl $S - 1$) S

When we add a new clause, we reduce the trail until we get to the first literal included in C . Then we can mark the conflict.

fun cut -trail-wrt-clause **where**

cut -trail-wrt-clause C [] $S = S$ |

cut -trail-wrt-clause C ($Marked$ $L - \#$ M) $S =$

(if $-L \in \#$ C then S

else cut -trail-wrt-clause C M ($decr$ -bt-lvl (tl -trail S))) |

cut -trail-wrt-clause C ($Propagated$ $L - \#$ M) $S =$

(if $-L \in \#$ C then S

else cut -trail-wrt-clause C M (tl -trail S))

definition add -new-clause-and-update :: $'ccls \Rightarrow 'st \Rightarrow 'st$ **where**

add -new-clause-and-update C $S =$

(if trail $S \models_{as}$ $CNot$ ($mset$ -ccls C)

then $update$ -conflicting ($Some$ C) (add -init-cls (cls -of-ccls C)

(cut -trail-wrt-clause ($mset$ -ccls C) (trail S) S))

else add -init-cls (cls -of-ccls C) S)

thm cut -trail-wrt-clause.induct

lemma $init$ -clss- cut -trail-wrt-clause[simp]:

$init$ -clss (cut -trail-wrt-clause C M S) = $init$ -clss S

$\langle proof \rangle$

lemma $learned$ -clss- cut -trail-wrt-clause[simp]:

$learned$ -clss (cut -trail-wrt-clause C M S) = $learned$ -clss S

$\langle proof \rangle$

lemma $conflicting$ -clss- cut -trail-wrt-clause[simp]:

$conflicting$ (cut -trail-wrt-clause C M S) = $conflicting$ S

$\langle proof \rangle$

lemma $trail$ -cut- $trail$ -wrt-clause:

$\exists M. trail$ $S = M @ trail$ (cut -trail-wrt-clause C (trail S) S)

$\langle proof \rangle$

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))

$\langle proof \rangle$

lemma *cut-trail-wrt-clause-backtrack-lvl-length-marked:*

assumes

$\text{backtrack-lvl } T = \text{length } (\text{get-all-levels-of-marked } (\text{trail } T))$

shows

$\text{backtrack-lvl } (\text{cut-trail-wrt-clause } C (\text{trail } T) T) =$

$\text{length } (\text{get-all-levels-of-marked } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)))$

$\langle \text{proof} \rangle$

lemma *cut-trail-wrt-clause-get-all-levels-of-marked:*

assumes $\text{get-all-levels-of-marked } (\text{trail } T) = \text{rev } [\text{Suc } 0..<$

$\text{Suc } (\text{length } (\text{get-all-levels-of-marked } (\text{trail } T)))]$

shows

$\text{get-all-levels-of-marked } (\text{trail } ((\text{cut-trail-wrt-clause } C (\text{trail } T) T))) = \text{rev } [\text{Suc } 0..<$

$\text{Suc } (\text{length } (\text{get-all-levels-of-marked } (\text{trail } ((\text{cut-trail-wrt-clause } C (\text{trail } T) T))))]$

$\langle \text{proof} \rangle$

lemma *cut-trail-wrt-clause-CNot-trail:*

assumes $\text{trail } T \models_{\text{as}} \text{CNot } C$

shows

$(\text{trail } ((\text{cut-trail-wrt-clause } C (\text{trail } T) T))) \models_{\text{as}} \text{CNot } C$

$\langle \text{proof} \rangle$

lemma *cut-trail-wrt-clause-hd-trail-in-or-empty-trail:*

$((\forall L \in \#C. \neg L \in \text{lits-of-l } (\text{trail } T)) \wedge \text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T) = [])$

$\vee (\neg \text{lit-of } (\text{hd } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T))) \in \#C$

$\wedge \text{length } (\text{trail } (\text{cut-trail-wrt-clause } C (\text{trail } T) T)) \geq 1)$

$\langle \text{proof} \rangle$

We can fully run *cdcl_W*-s or add a clause. Remark that we use *cdcl_W*-s to avoid an explicit *skip*, *resolve*, and *backtrack* normalisation to get rid of the conflict *C* if possible.

inductive *incremental-cdcl_W* :: '*st* \Rightarrow '*st* \Rightarrow bool **for** *S* **where**

add-conflict:

$\text{trail } S \models_{\text{asm}} \text{init-clss } S \implies \text{distinct-mset } (\text{mset-ccls } C) \implies \text{conflicting } S = \text{None} \implies$

$\text{trail } S \models_{\text{as}} \text{CNot } (\text{mset-ccls } C) \implies$

full cdcl_W-stgy

$(\text{update-conflicting } (\text{Some } C))$

$(\text{add-init-cls } (\text{cls-of-ccls } C) (\text{cut-trail-wrt-clause } (\text{mset-ccls } C) (\text{trail } S) S))) T \implies$

incremental-cdcl_W *S* *T* |

add-no-conflict:

$\text{trail } S \models_{\text{asm}} \text{init-clss } S \implies \text{distinct-mset } (\text{mset-ccls } C) \implies \text{conflicting } S = \text{None} \implies$

$\neg \text{trail } S \models_{\text{as}} \text{CNot } (\text{mset-ccls } C) \implies$

full cdcl_W-stgy $(\text{add-init-cls } (\text{cls-of-ccls } C) S) T \implies$

incremental-cdcl_W *S* *T*

lemma *cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-all-struct-inv:*

assumes

inv-T: *cdcl_W-all-struct-inv* *T* **and**

tr-T-N[simp]: $\text{trail } T \models_{\text{asm}} N$ **and**

tr-C[simp]: $\text{trail } T \models_{\text{as}} \text{CNot } (\text{mset-ccls } C)$ **and**

[simp]: $\text{distinct-mset } (\text{mset-ccls } C)$

shows *cdcl_W-all-struct-inv* $(\text{add-new-clause-and-update } C T)$ **(is** *cdcl_W-all-struct-inv* *?T')*

$\langle \text{proof} \rangle$

lemma *cdcl_W-all-struct-inv-add-new-clause-and-update-cdcl_W-stgy-inv:*

assumes
inv-s: *cdcl_W-stgy-invariant* *T* **and**
inv: *cdcl_W-all-struct-inv* *T* **and**
tr-T-N[simp]: *trail* *T* \models_{asm} *N* **and**
tr-C[simp]: *trail* *T* \models_{as} *CNot* (*mset-ccls* *C*) **and**
[simp]: *distinct-mset* (*mset-ccls* *C*)
shows *cdcl_W-stgy-invariant* (*add-new-clause-and-update* *C* *T*)
(is *cdcl_W-stgy-invariant* ?*T'*)
⟨*proof*⟩

lemma *full-cdcl_W-stgy-inv-normal-form*:
assumes
full: *full cdcl_W-stgy* *S* *T* **and**
inv-s: *cdcl_W-stgy-invariant* *S* **and**
inv: *cdcl_W-all-struct-inv* *S*
shows *conflicting* *T* = *Some* {#} \wedge *unsatisfiable* (*set-mset* (*init-clss* *S*))
 \vee *conflicting* *T* = *None* \wedge *trail* *T* \models_{asm} *init-clss* *S* \wedge *satisfiable* (*set-mset* (*init-clss* *S*))
⟨*proof*⟩

lemma *incremental-cdcl_W-inv*:
assumes
inc: *incremental-cdcl_W* *S* *T* **and**
inv: *cdcl_W-all-struct-inv* *S* **and**
s-inv: *cdcl_W-stgy-invariant* *S*
shows
cdcl_W-all-struct-inv *T* **and**
cdcl_W-stgy-invariant *T*
⟨*proof*⟩

lemma *rtrancpl-incremental-cdcl_W-inv*:
assumes
inc: *incremental-cdcl_W*** *S* *T* **and**
inv: *cdcl_W-all-struct-inv* *S* **and**
s-inv: *cdcl_W-stgy-invariant* *S*
shows
cdcl_W-all-struct-inv *T* **and**
cdcl_W-stgy-invariant *T*
⟨*proof*⟩

lemma *incremental-conclusive-state*:
assumes
inc: *incremental-cdcl_W* *S* *T* **and**
inv: *cdcl_W-all-struct-inv* *S* **and**
s-inv: *cdcl_W-stgy-invariant* *S*
shows *conflicting* *T* = *Some* {#} \wedge *unsatisfiable* (*set-mset* (*init-clss* *T*))
 \vee *conflicting* *T* = *None* \wedge *trail* *T* \models_{asm} *init-clss* *T* \wedge *satisfiable* (*set-mset* (*init-clss* *T*))
⟨*proof*⟩

lemma *trancpl-incremental-correct*:
assumes
inc: *incremental-cdcl_W++* *S* *T* **and**
inv: *cdcl_W-all-struct-inv* *S* **and**
s-inv: *cdcl_W-stgy-invariant* *S*
shows *conflicting* *T* = *Some* {#} \wedge *unsatisfiable* (*set-mset* (*init-clss* *T*))
 \vee *conflicting* *T* = *None* \wedge *trail* *T* \models_{asm} *init-clss* *T* \wedge *satisfiable* (*set-mset* (*init-clss* *T*))

$\langle proof \rangle$

end

end

24 2-Watched-Literal

```
theory CDCL-Two-Watched-Literals
imports CDCL-WNOT
begin
```

First we define here the core of the two-watched literal datastructure:

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 datastructure.

We will directly on the two-watched literals datastructure 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.

24.1 Essence of 2-WL

24.1.1 Datastructure and Access Functions

Only the 2-watched literals have to be verified here: the backtrack level and the trail that appear in the state are not related to the 2-watched algorithm.

```
datatype 'v twl-clause =
  TWL-Clause (watched: 'v literal list) (unwatched: 'v literal list)
```

```
datatype 'v twl-state =
  TWL-State (raw-trail: ('v, nat, 'v twl-clause) marked-lit list)
    (raw-init-clss: 'v twl-clause list)
    (raw-learned-clss: 'v twl-clause list) (backtrack-lvl: nat)
    (raw-conflicting: 'v literal list option)
```

```
fun mmsset-of-mlit' :: ('v, nat, 'v twl-clause) marked-lit  $\Rightarrow$  ('v, nat, 'v clause) marked-lit
where
  mmsset-of-mlit' (Propagated L C) = Propagated L (mset (watched C @ unwatched C)) |
  mmsset-of-mlit' (Marked L i) = Marked L i
```

```
lemma lit-of-mmsset-of-mlit'[simp]: lit-of (mmsset-of-mlit' x) = lit-of x
 $\langle proof \rangle$ 
```

```
lemma lits-of-mmsset-of-mlit'[simp]: lits-of (mmsset-of-mlit' ' S) = lits-of S
 $\langle proof \rangle$ 
```

```
abbreviation trail where
  trail S  $\equiv$  map mmsset-of-mlit' (raw-trail S)
```

abbreviation *clauses-of-l where*

clauses-of-l $\equiv \lambda L. \text{mset } (\text{map } \text{mset } L)$

definition *raw-clause* $:: 'v \text{ twl-clause} \Rightarrow 'v \text{ literal list where}$

raw-clause $C \equiv \text{watched } C @ \text{unwatched } C$

abbreviation *raw-clss* $:: 'v \text{ twl-state} \Rightarrow 'v \text{ clauses where}$

raw-clss $S \equiv \text{clauses-of-l } (\text{map } \text{raw-clause } (\text{raw-init-clss } S @ \text{raw-learned-clss } S))$

interpretation *raw-cl*

$\lambda C. \text{mset } (\text{raw-clause } C)$

$\lambda L C. \text{TWL-Clause } (\text{watched } C) (L \# \text{unwatched } C)$

$\lambda L C. \text{TWL-Clause } [] (\text{remove1 } L (\text{raw-clause } C))$

$\langle \text{proof} \rangle$

lemma *mset-map-clause-remove1-cond:*

$\text{mset } (\text{map } (\lambda x. \text{mset } (\text{unwatched } x) + \text{mset } (\text{watched } x))$
 $(\text{remove1-cond } (\lambda D. \text{mset } (\text{raw-clause } D) = \text{mset } (\text{raw-clause } a)) \text{ } Cs)) =$
 $\text{remove1-mset } (\text{mset } (\text{raw-clause } a)) (\text{mset } (\text{map } (\lambda x. \text{mset } (\text{raw-clause } x)) \text{ } Cs))$
 $\langle \text{proof} \rangle$

interpretation *raw-clss*

$\lambda C. \text{mset } (\text{raw-clause } C)$

$\lambda L C. \text{TWL-Clause } (\text{watched } C) (L \# \text{unwatched } C)$

$\lambda L C. \text{TWL-Clause } [] (\text{remove1 } L (\text{raw-clause } C))$

$\lambda C. \text{clauses-of-l } (\text{map } \text{raw-clause } C) \text{ op } @$

$\lambda L C. L \in \text{set } C \text{ op } \# \lambda C. \text{remove1-cond } (\lambda D. \text{mset } (\text{raw-clause } D) = \text{mset } (\text{raw-clause } C))$

$\langle \text{proof} \rangle$

lemma *ex-mset-unwatched-watched:*

$\exists a. \text{mset } (\text{unwatched } a) + \text{mset } (\text{watched } a) = E$

$\langle \text{proof} \rangle$

thm *CDCL-Two-Watched-Literals.raw-cl-axioms*

interpretation *twl: state_w-ops*

$\lambda C. \text{mset } (\text{raw-clause } C)$

$\lambda L C. \text{TWL-Clause } (\text{watched } C) (L \# \text{unwatched } C)$

$\lambda L C. \text{TWL-Clause } [] (\text{remove1 } L (\text{raw-clause } C))$

$\lambda C. \text{clauses-of-l } (\text{map } \text{raw-clause } C) \text{ op } @$

$\lambda L C. L \in \text{set } C \text{ op } \# \lambda C. \text{remove1-cond } (\lambda D. \text{mset } (\text{raw-clause } D) = \text{mset } (\text{raw-clause } C))$

$\text{mset } \lambda xs \text{ } ys. \text{case-prod } \text{append } (\text{fold } (\lambda x \text{ } (ys, zs). (\text{remove1 } x \text{ } ys, x \# zs)) \text{ } xs \text{ } (ys, []))$
 $\text{op } \# \text{remove1}$

raw-clause $\lambda C. \text{TWL-Clause } [] C$

trail $\lambda S. \text{hd } (\text{raw-trail } S)$

raw-init-clss *raw-learned-clss* *backtrack-lvl* *raw-conflicting*

$\langle \text{proof} \rangle$

declare *CDCL-Two-Watched-Literals.twl.mset-ccls-ccls-of-cl* $[simp \text{ del}]$

lemma *mmset-of-mlit'-mmset-of-mlit* $[simp]$:

twl.mmset-of-mlit $L = \text{mmset-of-mlit}' L$
 $\langle \text{proof} \rangle$

definition

candidates-propagate $:: 'v \text{ twl-state} \Rightarrow ('v \text{ literal} \times 'v \text{ twl-clause}) \text{ set}$
where
candidates-propagate $S =$
 $\{(L, C) \mid L \text{ } C.$
 $C \in \text{set } (\text{twl.raw-clauses } S) \wedge$
 $\text{set } (\text{watched } C) - (\text{uminus } ' \text{ lits-of-l } (\text{trail } S)) = \{L\} \wedge$
 $\text{undefined-lit } (\text{raw-trail } S) L\}$

definition *candidates-conflict* $:: 'v \text{ twl-state} \Rightarrow 'v \text{ twl-clause set}$ **where**

candidates-conflict $S =$
 $\{C. C \in \text{set } (\text{twl.raw-clauses } S) \wedge$
 $\text{set } (\text{watched } C) \subseteq \text{uminus } ' \text{ lits-of-l } (\text{raw-trail } S)\}$

primrec (*nonexhaustive*) *index* $:: 'a \text{ list} \Rightarrow 'a \Rightarrow \text{nat}$ **where**
index $(a \# l) \ c = (\text{if } a = c \text{ then } 0 \text{ else } 1 + \text{index } l \ c)$

lemma *index-nth*:

$a \in \text{set } l \implies l ! (\text{index } l \ a) = a$
 $\langle \text{proof} \rangle$

24.1.2 Invariants

We need the following property about updates: if there is a literal L with $-L$ in the trail, and L is not watched, then it stays unwatched; i.e., while updating with *rewatch* it does not get swap with a watched literal L' such that $-L'$ is in the trail.

primrec *watched-decided-most-recently* $:: ('v, 'wl, 'mark) \text{ marked-lit list} \Rightarrow$
 $'v \text{ twl-clause} \Rightarrow \text{bool}$

where

watched-decided-most-recently $M \ (\text{TWL-Clause } W \ UW) \longleftrightarrow$
 $(\forall L' \in \text{set } W. \forall L \in \text{set } UW.$
 $-L' \in \text{lits-of-l } M \longrightarrow -L \in \text{lits-of-l } M \longrightarrow L \notin \# \text{ mset } W \longrightarrow$
 $\text{index } (\text{map lit-of } M) \ (-L') \leq \text{index } (\text{map lit-of } M) \ (-L))$

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-tw-cls* $:: ('v, 'wl, 'mark) \text{ marked-lit list} \Rightarrow 'v \text{ twl-clause} \Rightarrow \text{bool}$ **where**
wf-tw-cls $M \ (\text{TWL-Clause } W \ UW) \longleftrightarrow$
 $\text{distinct } W \wedge \text{length } W \leq 2 \wedge (\text{length } W < 2 \longrightarrow \text{set } UW \subseteq \text{set } W) \wedge$
 $(\forall L \in \text{set } W. -L \in \text{lits-of-l } M \longrightarrow (\forall L' \in \text{set } UW. L' \notin \text{set } W \longrightarrow -L' \in \text{lits-of-l } M)) \wedge$
watched-decided-most-recently $M \ (\text{TWL-Clause } W \ UW)$

lemma *size-mset-2*: $\text{size } x1 = 2 \longleftrightarrow (\exists a \ b. x1 = \{\#a, b\# \})$
 $\langle \text{proof} \rangle$

lemma *distinct-mset-size-2*: $\text{distinct-mset } \{\#a, b\# \} \longleftrightarrow a \neq b$
 $\langle \text{proof} \rangle$

lemma *wf-tw-cls-annotation-independant*:

assumes $M: \text{map lit-of } M = \text{map lit-of } M'$

shows $\text{wf-tw-cls } M \ (\text{TWL-Clause } W \ UW) \longleftrightarrow \text{wf-tw-cls } M' \ (\text{TWL-Clause } W \ UW)$

$\langle \text{proof} \rangle$

lemma *wf-twl-cls-wf-twl-cls-tl*:

assumes *wf*: *wf-twl-cls* *M* *C* **and** *n-d*: *no-dup* *M*
shows *wf-twl-cls* (*tl* *M*) *C*

<proof>

lemma *wf-twl-cls-append*:

assumes
n-d: *no-dup* (*M'* @ *M*) **and**
wf: *wf-twl-cls* (*M'* @ *M*) *C*
shows *wf-twl-cls* *M* *C*

<proof>

definition *wf-twl-state* :: '*v* *twl-state* \Rightarrow *bool* **where**

wf-twl-state *S* \longleftrightarrow
 $(\forall C \in \text{set } (\text{twl.raw-clauses } S). \text{wf-twl-cls } (\text{raw-trail } S) \ C) \wedge \text{no-dup } (\text{raw-trail } S)$

lemma *wf-candidates-propagate-sound*:

assumes *wf*: *wf-twl-state* *S* **and**
cand: (*L*, *C*) \in *candidates-propagate* *S*
shows *raw-trail* *S* \models_{as} *CNot* (*mset* (*removeAll* *L* (*raw-clause* *C*))) \wedge *undefined-lit* (*raw-trail* *S*) *L*
(is ?Not \wedge ?undef)

<proof>

lemma *wf-candidates-propagate-complete*:

assumes *wf*: *wf-twl-state* *S* **and**
c-mem: *C* \in *set* (*twl.raw-clauses* *S*) **and**
l-mem: *L* \in *set* (*raw-clause* *C*) **and**
unsat: *trail* *S* \models_{as} *CNot* (*mset-set* (*set* (*raw-clause* *C*) - {*L*})) **and**
undef: *undefined-lit* (*raw-trail* *S*) *L*
shows (*L*, *C*) \in *candidates-propagate* *S*

<proof>

lemma *wf-candidates-conflict-sound*:

assumes *wf*: *wf-twl-state* *S* **and**
cand: *C* \in *candidates-conflict* *S*
shows *trail* *S* \models_{as} *CNot* (*mset* (*raw-clause* *C*)) \wedge *C* \in *set* (*twl.raw-clauses* *S*)

<proof>

lemma *wf-candidates-conflict-complete*:

assumes *wf*: *wf-twl-state* *S* **and**
c-mem: *C* \in *set* (*twl.raw-clauses* *S*) **and**
unsat: *trail* *S* \models_{as} *CNot* (*mset* (*raw-clause* *C*))
shows *C* \in *candidates-conflict* *S*

<proof>

typedef '*v* *wf-twl* = {*S*::'*v* *twl-state*. *wf-twl-state* *S*}

morphisms *rough-state-of-twl twl-of-rough-state*

<proof>

lemma [*code abstype*]:

twl-of-rough-state (*rough-state-of-twl* *S*) = *S*
<proof>

lemma *wf-twl-state-rough-state-of-twl[simp]*: *wf-twl-state* (*rough-state-of-twl* *S*)

<proof>

abbreviation *candidates-conflict-twl* :: 'v wf-twl \Rightarrow 'v twl-clause set **where**
candidates-conflict-twl *S* \equiv *candidates-conflict* (*rough-state-of-twl* *S*)

abbreviation *candidates-propagate-twl* :: 'v wf-twl \Rightarrow ('v literal \times 'v 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, nat, 'a twl-clause) marked-lit list **where**
raw-trail-twl *S* \equiv *raw-trail* (*rough-state-of-twl* *S*)

abbreviation *trail-twl* :: 'a wf-twl \Rightarrow ('a, nat, 'a literal multiset) marked-lit list **where**
trail-twl *S* \equiv *trail* (*rough-state-of-twl* *S*)

abbreviation *raw-clauses-twl* :: 'a wf-twl \Rightarrow 'a twl-clause list **where**
raw-clauses-twl *S* \equiv *twl.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 *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* (*mset* (*raw-clause* *C*))
shows *C* \in *candidates-conflict-twl* *S*
 <proof>

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*

24.1.3 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: $\text{no-dup } (\text{raw-trail } S) \implies \text{mset } (\text{raw-clause } (\text{watch } S \ C)) = \text{mset } C$ **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: $\text{mset } (\text{raw-clause } (\text{rewatch } L' \ S \ C')) = \text{mset } (\text{raw-clause } C')$ **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 :: $('v, \text{nat}, 'v \ \text{twl-clause}) \text{ marked-lit} \Rightarrow 'v \ \text{twl-state} \Rightarrow 'v \ \text{twl-state}$

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 $C \ S =$

$\text{TWL-State } (\text{raw-trail } S)$

$(\text{removeAll-cond } (\lambda D. \text{mset } (\text{raw-clause } D) = \text{mset } C) (\text{raw-init-clss } S))$

$(\text{removeAll-cond } (\lambda D. \text{mset } (\text{raw-clause } D) = \text{mset } C) (\text{raw-learned-clss } S))$

$(\text{backtrack-lvl } S)$

$(\text{raw-conflicting } S)$

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*:

$\text{raw-trail } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = M$

$\text{raw-learned-clss } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = U$

$\text{backtrack-lvl } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = k$

$\text{raw-conflicting } (\text{fold add-init-cl } Cs \ (\text{TWL-State } M \ N \ U \ k \ C)) = C$

$\langle \text{proof} \rangle$

lemma *unchanged-init-state[simp]*:

$raw-trail \ (init-state \ N) = []$
 $raw-learned-clss \ (init-state \ N) = []$
 $backtrack-lvl \ (init-state \ N) = 0$
 $raw-conflicting \ (init-state \ N) = None$
 $\langle proof \rangle$

lemma *clauses-init-fold-add-init*:

$no-dup \ M \implies$
 $twl.init-clss \ (fold \ add-init-cls \ Cs \ (TWL-State \ M \ N \ U \ k \ C)) =$
 $clauses-of-l \ Cs + clauses-of-l \ (map \ raw-clause \ N)$
 $\langle proof \rangle$

lemma *init-clss-init-state[simp]*: $twl.init-clss \ (init-state \ N) = clauses-of-l \ N$
 $\langle proof \rangle$

definition *restart'* **where**

$restart' \ S = TWL-State \ [] \ (raw-init-clss \ S) \ (restart-learned \ S) \ 0 \ None$

end

24.1.4 Instantiation 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
 $l = [] \implies P$ **and**
 $\bigwedge x. \ l = [x] \implies P$ **and**
 $\bigwedge x \ y \ xs. \ l = x \ # \ y \ # \ xs \implies P$
shows P
 $\langle proof \rangle$

lemma *filter-in-list-prop-verifiedD*:

assumes $[L \leftarrow P \ . \ Q \ L] = l$
shows $\forall x \in set \ l. \ x \in set \ P \wedge Q \ x$
 $\langle proof \rangle$

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
 $\langle proof \rangle$

lemma *watch-nat-lists-disjointD*:

assumes
 $l: [L \leftarrow remdups \ C. \ - \ L \notin lits-of-l \ (raw-trail \ S)] = l$ **and**
 $l': [L \leftarrow map \ (\lambda L. \ - \ lit-of \ L) \ (raw-trail \ S) \ . \ L \in set \ C] = l'$
shows $\forall x \in set \ l. \ \forall y \in set \ l'. \ x \neq y$

$\langle \text{proof} \rangle$

lemma *watch-nat-list-cases-witness*[consumes 2, case-names nil-nil nil-single nil-other single-nil single-other other]:

fixes

$C :: 'v \text{ literal list}$ **and**

$S :: 'v \text{ twl-state}$

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*) **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*

$\langle \text{proof} \rangle$

lemma *watch-nat-list-cases* [consumes 1, case-names nil-nil nil-single nil-other single-nil single-other other]:

fixes

$C :: 'v \text{ literal list}$ **and**

$S :: 'v \text{ twl-state}$

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*) **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*

$\langle \text{proof} \rangle$

lemma *watch-nat-lists-set-union-witness*:

fixes

$C :: 'v \text{ literal list}$ **and**

$S :: 'v \text{ twl-state}$

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 $\text{set } C = \text{set } xs \cup \text{set } ys$

$\langle \text{proof} \rangle$

lemma *mset-intersection-inclusion*: $A + (B - A) = B \longleftrightarrow A \subseteq \# B$

$\langle \text{proof} \rangle$

lemma *clause-watch-nat*:

assumes *no-dup* (*raw-trail* *S*)

shows *mset* (*raw-clause* (*watch-nat* *S* *C*)) = *mset* *C*

<proof>

lemma *index-uminus-index-map-uminus*:

$-a \in \text{set } L \implies \text{index } L \ (-a) = \text{index } (\text{map } \text{uminus } L) \ (a::'a \text{ literal})$

<proof>

lemma *index-filter*:

$a \in \text{set } L \implies b \in \text{set } L \implies P \ a \implies P \ b \implies$

$\text{index } L \ a \leq \text{index } L \ b \longleftrightarrow \text{index } (\text{filter } P \ L) \ a \leq \text{index } (\text{filter } P \ L) \ b$

<proof>

lemma *foldr-remove1-W-Nil[simp]*: *foldr* *remove1* *W* [] = []

<proof>

lemma *image-lit-of-mmset-of-mlit'[simp]*:

lit-of ' *mmset-of-mlit'* ' *A* = *lit-of* ' *A*

<proof>

lemma *distinct-filter-eq*:

assumes *distinct* *xs*

shows $[L \leftarrow xs. L = a] = (\text{if } a \in \text{set } xs \text{ then } [a] \text{ else } [])$

<proof>

lemma *no-dup-distinct-map-uminus-lit-of*:

no-dup *xs* $\implies \text{distinct } (\text{map } (\lambda L. - \text{lit-of } L) \ xs)$

<proof>

lemma *wf-watch-witness*:

fixes *C* :: '*v* literal list **and**

S :: '*v* twl-state

defines

ass: *neg-not-assigned* $\equiv \text{filter } (\lambda L. -L \notin \text{lits-of-l } (\text{raw-trail } S)) \ (\text{remdups } C)$ **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-twlc* (*raw-trail* *S*) (*TWL-Clause* *W* (*foldr* *remove1* *W* *C*))

<proof>

lemma *wf-watch-nat*: *no-dup* (*raw-trail* *S*) $\implies \text{wf-twlc}$ (*raw-trail* *S*) (*watch-nat* *S* *C*)

<proof>

definition

rewatch-nat ::

'v literal \Rightarrow *'v* twl-state \Rightarrow *'v* twl-clause \Rightarrow *'v* twl-clause

where

rewatch-nat *L* *S* *C* =

(*if* $-L \in \text{set } (\text{watched } C)$ *then*

case *filter* $(\lambda L'. L' \notin \text{set } (\text{watched } C) \wedge -L' \notin \text{insert } L \ (\text{lits-of-l } (\text{trail } S)))$

(*unwatched* *C*) *of*

```

    []  $\Rightarrow$  C
  | L' # -  $\Rightarrow$ 
    TWL-Clause (L' # remove1 (-L) (watched C)) (-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 mset (raw-clause (rewatch-nat L S C)) = mset (raw-clause C)
<proof>

```

lemma *filter-sorted-list-of-multiset-Nil*:

```

[x  $\leftarrow$  sorted-list-of-multiset M. p x] = []  $\longleftrightarrow$  ( $\forall x \in \# M. \neg p x$ )
<proof>

```

lemma *filter-sorted-list-of-multiset-ConsD*:

```

[x  $\leftarrow$  sorted-list-of-multiset M. p x] = x # xs  $\implies$  p x
<proof>

```

lemma *mset-minus-single-eq-mempty*:

```

a - {#b#} = {#}  $\longleftrightarrow$  a = {#b#}  $\vee$  a = {#}
<proof>

```

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>

```

lemma *filter-sorted-list-of-multiset-eqD*:

```

assumes [x  $\leftarrow$  sorted-list-of-multiset A. p x] = x # xs (is ?comp = -)
shows x  $\in \# A$ 
<proof>

```

lemma *clause-rewatch-witness'*:

```

assumes
  wf: wf-tw-cls (raw-trail S) C and
  undef: undefined-lit (raw-trail S) (lit-of L)
shows wf-tw-cls (L # raw-trail S) (rewatch-nat (lit-of L) S C)
<proof>

```

interpretation *twl*: abstract-twl watch-nat rewatch-nat raw-learned-clss

<proof>

interpretation *twl2*: abstract-twl watch-nat rewatch-nat $\lambda-. []$

<proof>

end

24.2 Two Watched-Literals with invariant

theory *CDCL-Two-Watched-Literals-Invariant*

imports *CDCL-Two-Watched-Literals DPLL-CDCL-W-Implementation*

begin

24.2.1 Interpretation for *conflict-driven-clause-learning_W.cdcl_W*

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-tw*

begin

Direct Interpretation lemma *mset-map-removeAll-cond:*

mset (map (λx. mset (raw-clause x))
(removeAll-cond (λD. mset (raw-clause D) = mset (raw-clause C)) N))
= mset (removeAll (mset (raw-clause C)) (map (λx. mset (raw-clause x)) N))
⟨proof⟩

lemma *mset-raw-init-clss-init-state:*

mset (map (λx. mset (raw-clause x)) (raw-init-clss (init-state (map raw-clause N))))
= mset (map (λx. mset (raw-clause x)) N)
⟨proof⟩

interpretation *rough-cdcl: state_W*

λC. mset (raw-clause C)

λL C. TWL-Clause (watched C) (L # unwatched C)
λL C. TWL-Clause [] (remove1 L (raw-clause C))
λC. clauses-of-l (map raw-clause C) op @
λL C. L ∈ set C op # λC. remove1-cond (λD. mset (raw-clause D) = mset (raw-clause C))

mset λxs ys. case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))
op # remove1

raw-clause λC. TWL-Clause [] C
trail λS. hd (raw-trail S)
raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
cons-trail tl-trail λC. add-init-cls (raw-clause C) λC. add-learned-cls (raw-clause C)
λC. remove-cls (raw-clause C)
update-backtrack-lvl
update-conflicting λN. init-state (map raw-clause N) restart'
⟨proof⟩

interpretation *rough-cdcl: conflict-driven-clause-learning_W*

λC. mset (raw-clause C)

λL C. TWL-Clause (watched C) (L # unwatched C)
λL C. TWL-Clause [] (remove1 L (raw-clause C))
λC. clauses-of-l (map raw-clause C) op @
λL C. L ∈ set C op # λC. remove1-cond (λD. mset (raw-clause D) = mset (raw-clause C))

mset λxs ys. case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs (ys, []))
op # remove1

λC. raw-clause C λC. TWL-Clause [] C
trail λS. hd (raw-trail S)
raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting

$cons_trail\ tl_trail\ \lambda C. add_init_cls\ (raw_clause\ C)\ \lambda C. add_learned_cls\ (raw_clause\ C)$
 $\lambda C. remove_cls\ (raw_clause\ C)$
 $update_backtrack_lvl$
 $update_conflicting\ \lambda N. init_state\ (map\ raw_clause\ N)\ restart'$
 $\langle proof \rangle$

declare $local.rough_cdcl.mset_ccls_ccls_of_cls[simp\ del]$

Opaque Type with Invariant **declare** $rough_cdcl.state_simp[simp\ del]$

definition $cons_trail_twl :: ('v, nat, 'v\ twl_clause)\ marked_lit \Rightarrow 'v\ wf_twl \Rightarrow 'v\ wf_twl$
where
 $cons_trail_twl\ L\ S \equiv twl_of_rough_state\ (cons_trail\ L\ (rough_state_of_twl\ S))$

lemma $wf_twl_state_cons_trail$:

assumes

$undef: undefined_lit\ (raw_trail\ S)\ (lit_of\ L)$ **and**

$wf: wf_twl_state\ S$

shows $wf_twl_state\ (cons_trail\ L\ S)$

$\langle proof \rangle$

lemma $rough_state_of_twl_cons_trail$:

$undefined_lit\ (raw_trail_twl\ S)\ (lit_of\ L) \implies$

$rough_state_of_twl\ (cons_trail_twl\ L\ S) = cons_trail\ L\ (rough_state_of_twl\ S)$

$\langle proof \rangle$

abbreviation $add_init_cls_twl$ **where**

$add_init_cls_twl\ C\ S \equiv twl_of_rough_state\ (add_init_cls\ C\ (rough_state_of_twl\ S))$

lemma $wf_twl_add_init_cls: wf_twl_state\ S \implies wf_twl_state\ (add_init_cls\ L\ S)$

$\langle proof \rangle$

lemma $rough_state_of_twl_add_init_cls$:

$rough_state_of_twl\ (add_init_cls_twl\ L\ S) = add_init_cls\ L\ (rough_state_of_twl\ S)$

$\langle proof \rangle$

abbreviation $add_learned_cls_twl$ **where**

$add_learned_cls_twl\ C\ S \equiv twl_of_rough_state\ (add_learned_cls\ C\ (rough_state_of_twl\ S))$

lemma $wf_twl_add_learned_cls: wf_twl_state\ S \implies wf_twl_state\ (add_learned_cls\ L\ S)$

$\langle proof \rangle$

lemma $rough_state_of_twl_add_learned_cls$:

$rough_state_of_twl\ (add_learned_cls_twl\ L\ S) = add_learned_cls\ L\ (rough_state_of_twl\ S)$

$\langle proof \rangle$

abbreviation $remove_cls_twl$ **where**

$remove_cls_twl\ C\ S \equiv twl_of_rough_state\ (remove_cls\ C\ (rough_state_of_twl\ S))$

lemma $set_removeAll_condD: x \in set\ (removeAll_cond\ f\ xs) \implies x \in set\ xs$

$\langle proof \rangle$

lemma $wf_twl_remove_cls: wf_twl_state\ S \implies wf_twl_state\ (remove_cls\ L\ S)$

$\langle proof \rangle$

lemma *rough-state-of-twl-remove-cls*:
 $\text{rough-state-of-twl } (\text{remove-cls-twl } L \ S) = \text{remove-cls } L \ (\text{rough-state-of-twl } S)$
 $\langle \text{proof} \rangle$

abbreviation *init-state-twl* **where**
 $\text{init-state-twl } N \equiv \text{twl-of-rough-state } (\text{init-state } N)$

lemma *wf-twl-state-wf-twl-state-fold-add-init-cls*:
assumes *wf-twl-state* S
shows *wf-twl-state* ($\text{fold add-init-cls } N \ S$)
 $\langle \text{proof} \rangle$

lemma *wf-twl-state-epsilon-state[simp]*:
 $\text{wf-twl-state } (\text{TWL-State } [] [] 0 \text{ None})$
 $\langle \text{proof} \rangle$

lemma *wf-twl-init-state*: *wf-twl-state* ($\text{init-state } N$)
 $\langle \text{proof} \rangle$

lemma *rough-state-of-twl-init-state*:
 $\text{rough-state-of-twl } (\text{init-state-twl } N) = \text{init-state } N$
 $\langle \text{proof} \rangle$

abbreviation *tl-trail-twl* **where**
 $\text{tl-trail-twl } S \equiv \text{twl-of-rough-state } (\text{tl-trail } (\text{rough-state-of-twl } S))$

lemma *wf-twl-state-tl-trail*: *wf-twl-state* $S \implies \text{wf-twl-state } (\text{tl-trail } S)$
 $\langle \text{proof} \rangle$

lemma *rough-state-of-twl-tl-trail*:
 $\text{rough-state-of-twl } (\text{tl-trail-twl } S) = \text{tl-trail } (\text{rough-state-of-twl } S)$
 $\langle \text{proof} \rangle$

abbreviation *update-backtrack-lvl-twl* **where**
 $\text{update-backtrack-lvl-twl } k \ S \equiv \text{twl-of-rough-state } (\text{update-backtrack-lvl } k \ (\text{rough-state-of-twl } S))$

lemma *wf-twl-state-update-backtrack-lvl*:
 $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{update-backtrack-lvl } k \ S)$
 $\langle \text{proof} \rangle$

lemma *rough-state-of-twl-update-backtrack-lvl*:
 $\text{rough-state-of-twl } (\text{update-backtrack-lvl-twl } k \ S) = \text{update-backtrack-lvl } k \ (\text{rough-state-of-twl } S)$
 $\langle \text{proof} \rangle$

abbreviation *update-conflicting-twl* **where**
 $\text{update-conflicting-twl } k \ S \equiv \text{twl-of-rough-state } (\text{update-conflicting } k \ (\text{rough-state-of-twl } S))$

lemma *wf-twl-state-update-conflicting*:
 $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{update-conflicting } k \ S)$
 $\langle \text{proof} \rangle$

lemma *rough-state-of-twl-update-conflicting*:
 $\text{rough-state-of-twl } (\text{update-conflicting-twl } k \ S) = \text{update-conflicting } k \ (\text{rough-state-of-twl } S)$

$\langle \text{proof} \rangle$

abbreviation *raw-clauses-twl* **where**

raw-clauses-twl $S \equiv \text{twl.raw-clauses } (\text{rough-state-of-twl } S)$

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*:

$\text{mset } A \subseteq \# \text{ mset } B \implies \text{set } A \subseteq \text{set } B$

$\langle \text{proof} \rangle$

lemma *wf-wf-restart'*: $\text{wf-twl-state } S \implies \text{wf-twl-state } (\text{restart}' S)$

$\langle \text{proof} \rangle$

lemma *rough-state-of-twl-restart-twl*:

$\text{rough-state-of-twl } (\text{restart-twl } S) = \text{restart}' (\text{rough-state-of-twl } S)$

$\langle \text{proof} \rangle$

lemma *undefined-lit-trail-twl-raw-trail*[*iff*]:

$\text{undefined-lit } (\text{trail-twl } S) L \longleftrightarrow \text{undefined-lit } (\text{raw-trail-twl } S) L$

$\langle \text{proof} \rangle$

sublocale *wf-twl: conflict-driven-clause-learning_w*

$\lambda C. \text{mset } (\text{raw-clause } C)$

$\lambda L C. \text{TWL-Clause } (\text{watched } C) (L \# \text{unwatched } C)$

$\lambda L C. \text{TWL-Clause } [] (\text{remove1 } L (\text{raw-clause } C))$

$\lambda C. \text{clauses-of-l } (\text{map raw-clause } C) \text{ op } @$

$\lambda L C. L \in \text{set } C \text{ op } \# \lambda C. \text{remove1-cond } (\lambda D. \text{mset } (\text{raw-clause } D) = \text{mset } (\text{raw-clause } C))$

$\text{mset } \lambda xs \text{ ys. case-prod append } (\text{fold } (\lambda x (ys, zs). (\text{remove1 } x \text{ ys}, x \# zs)) xs (\text{ys}, []))$

$\text{op } \# \text{remove1}$

$\lambda C. \text{raw-clause } C \lambda C. \text{TWL-Clause } [] C$

trail-twl $\lambda S. \text{hd } (\text{raw-trail-twl } S)$

raw-init-clss-twl

raw-learned-clss-twl

backtrack-lvl-twl

raw-conflicting-twl

cons-trail-twl

tl-trail-twl

$\lambda C. \text{add-init-cls-twl } (\text{raw-clause } C)$

$\lambda C. \text{add-learned-cls-twl } (\text{raw-clause } C)$

$\lambda C. \text{remove-cls-twl } (\text{raw-clause } C)$

update-backtrack-lvl-twl

update-conflicting-twl

$\lambda N. \text{init-state-twl } (\text{map raw-clause } N)$

restart-twl

$\langle \text{proof} \rangle$

declare *local.rough-cdcl.mset-ccls-ccls-of-cls*[*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)

declare *wf-twl.state-simp*[*simp del*]

To avoid ambiguities:

no-notation *state-eq-tw*l (**infix** \sim 51)

Alternative Definition of CDCL using the candidates of 2-WL *inductive propagate-tw*l

$:: 'v \text{ wf-tw}l \Rightarrow 'v \text{ wf-tw}l \Rightarrow \text{bool}$ **where**
*propagate-tw*l-rule: $(L, C) \in \text{candidates-propagate-tw}l \ S \Rightarrow$
 $S' \sim \text{cons-trail-tw}l \ (\text{Propagated } L \ C) \ S \Rightarrow$
 $\text{raw-conflicting-tw}l \ S = \text{None} \Rightarrow$
 $\text{propagate-tw}l \ S \ S'$

inductive-cases *propagate-tw*lE: *propagate-tw*l $S \ T$

lemma *distinct-filter-eq-if*:

$\text{distinct } C \Rightarrow \text{length } (\text{filter } (op = L) \ C) = (\text{if } L \in \text{set } C \text{ then } 1 \text{ else } 0)$
 $\langle \text{proof} \rangle$

lemma *distinct-mset-remove1-All*:

$\text{distinct-mset } C \Rightarrow \text{remove1-mset } L \ C = \text{removeAll-mset } L \ C$
 $\langle \text{proof} \rangle$

lemma *propagate-tw*l-iff-*propagate*:

assumes *inv*: *wf-tw*l.cdcl_W-all-struct-*inv* S
shows *wf-tw*l.*propagate* $S \ T \longleftrightarrow \text{propagate-tw}l \ S \ T$ (**is** $?P \longleftrightarrow ?T$)
 $\langle \text{proof} \rangle$

no-notation *tw*l.state-eq-*tw*l (**infix** \sim TWL 51)

inductive *conflict-tw*l **where**

*conflict-tw*l-rule:
 $C \in \text{candidates-conflict-tw}l \ S \Rightarrow$
 $S' \sim \text{update-conflicting-tw}l \ (\text{Some } (\text{raw-clause } C)) \ S \Rightarrow$
 $\text{raw-conflicting-tw}l \ S = \text{None} \Rightarrow$
 $\text{conflict-tw}l \ S \ S'$

inductive-cases *conflict-tw*lE: *conflict-tw*l $S \ T$

lemma *conflict-tw*l-iff-*conflict*:

shows *wf-tw*l.*conflict* $S \ T \longleftrightarrow \text{conflict-tw}l \ S \ T$ (**is** $?C \longleftrightarrow ?T$)
 $\langle \text{proof} \rangle$

inductive *cdcl*_W-*tw*l $:: 'v \text{ wf-tw}l \Rightarrow 'v \text{ wf-tw}l \Rightarrow \text{bool}$ **for** $S :: 'v \text{ wf-tw}l$ **where**

propagate: *propagate-tw*l $S \ S' \Rightarrow \text{cdcl}_W\text{-tw}l \ S \ S' \mid$
conflict: *conflict-tw*l $S \ S' \Rightarrow \text{cdcl}_W\text{-tw}l \ S \ S' \mid$
other: *wf-tw*l.cdcl_W-o $S \ S' \Rightarrow \text{cdcl}_W\text{-tw}l \ S \ S' \mid$
rf: *wf-tw*l.cdcl_W-rf $S \ S' \Rightarrow \text{cdcl}_W\text{-tw}l \ S \ S'$

lemma *cdcl*_W-*tw*l-iff-*cdcl*_W:

assumes *wf-tw*l.cdcl_W-all-struct-*inv* S
shows *cdcl*_W-*tw*l $S \ T \longleftrightarrow \text{wf-tw}l.\text{cdcl}_W \ S \ T$
 $\langle \text{proof} \rangle$

lemma *rtranclp-cdcl*_W-*tw*l-all-struct-*inv*-*inv*:

assumes *cdcl*_W-*tw*l** $S \ T$ **and** *wf-tw*l.cdcl_W-all-struct-*inv* S
shows *wf-tw*l.cdcl_W-all-struct-*inv* T
 $\langle \text{proof} \rangle$

```

lemma rtrancp-cdclW-twl-iff-rtrancp-cdclW:
  assumes wf-twl.cdclW-all-struct-inv S
  shows cdclW-twl** S T  $\longleftrightarrow$  wf-twl.cdclW** S T (is ?T  $\longleftrightarrow$  ?W)
  <proof>

end

end
theory Prop-Superposition
imports Partial-Clausal-Logic ../lib/Herbrand-Interpretation
begin

```

25 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:
   $S \models_h C \longleftrightarrow \{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\} \models C$ 
  <proof>

```

```

lemma herbrand-consistent-interp:
  consistent-interp ( $\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\}$ )
  <proof>

```

```

lemma herbrand-total-over-set:
  total-over-set ( $\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\}$ ) T
  <proof>

```

```

lemma herbrand-total-over-m:
  total-over-m ( $\{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\}$ ) T
  <proof>

```

```

lemma herbrand-interp-iff-partial-interp-clss:
   $S \models_{hs} C \longleftrightarrow \{Pos\ P|P. P \in S\} \cup \{Neg\ P|P. P \notin S\} \models_s C$ 
  <proof>

```

```

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:  $\bigwedge C\ L. L \in \# S\ C \Longrightarrow is\_neg\ L$ 

```

```

locale ground-resolution-with-selection =
  selection S for S :: ('a :: wellorder) clause  $\Rightarrow$  'a clause

```

begin

context

fixes $N :: 'a \text{ clause set}$

begin

We do not create an equivalent of δ , but we directly defined N_C by inlining the definition.

function

$\text{production} :: 'a \text{ clause} \Rightarrow 'a \text{ interp}$

where

$\text{production } C =$

$\{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max } (\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1$
 $\wedge \neg (\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D) \models_h C \wedge S C = \{\#\}\}$

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

declare $\text{production.simps}[simp \text{ del}]$

definition $\text{interp} :: 'a \text{ clause} \Rightarrow 'a \text{ interp}$ **where**

$\text{interp } C = (\bigcup D \in \{D. D \# \subset \# C\}. \text{production } D)$

lemma production-unfold :

$\text{production } C = \{A. C \in N \wedge C \neq \{\#\} \wedge \text{Max } (\text{set-mset } C) = \text{Pos } A \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge \neg$
 $\text{interp } C \models_h C \wedge S C = \{\#\}\}$

$\langle \text{proof} \rangle$

abbreviation $\text{productive } A \equiv (\text{production } A \neq \{\})$

abbreviation $\text{produces} :: 'a \text{ clause} \Rightarrow 'a \Rightarrow \text{bool}$ **where**

$\text{produces } C A \equiv \text{production } C = \{A\}$

lemma producesD :

$\text{produces } C A \Longrightarrow C \in N \wedge C \neq \{\#\} \wedge \text{Pos } A = \text{Max } (\text{set-mset } C) \wedge \text{count } C (\text{Pos } A) \leq 1 \wedge$
 $\neg \text{interp } C \models_h C \wedge S C = \{\#\}$

$\langle \text{proof} \rangle$

lemma $\text{produces } C A \Longrightarrow \text{Pos } A \in \# C$

$\langle \text{proof} \rangle$

lemma $\text{interp'-def-in-set}$:

$\text{interp } C = (\bigcup D \in \{D \in N. D \# \subset \# C\}. \text{production } D)$

$\langle \text{proof} \rangle$

lemma $\text{production-iff-produces}$:

$\text{produces } D A \longleftrightarrow A \in \text{production } D$

$\langle \text{proof} \rangle$

definition $\text{Interp} :: 'a \text{ clause} \Rightarrow 'a \text{ interp}$ **where**

$\text{Interp } C = \text{interp } C \cup \text{production } C$

lemma

assumes $\text{produces } C P$

shows $\text{Interp } C \models_h C$

$\langle \text{proof} \rangle$

definition $INTERP :: 'a \text{ interp where}$
 $INTERP = (\bigcup D \in N. \text{ production } D)$

lemma *interp-subseteq-Interp[simp]:* $\text{interp } C \subseteq \text{Interp } C$
 $\langle \text{proof} \rangle$

lemma *Interp-as-UNION:* $\text{Interp } C = (\bigcup D \in \{D. D \# \subseteq \# C\}. \text{ production } D)$
 $\langle \text{proof} \rangle$

lemma *productive-not-empty:* $\text{productive } C \implies C \neq \{\#\}$
 $\langle \text{proof} \rangle$

lemma *productive-imp-produces-Max-literal:* $\text{productive } C \implies \text{produces } C (\text{atm-of } (\text{Max } (\text{set-mset } C)))$
 $\langle \text{proof} \rangle$

lemma *productive-imp-produces-Max-atom:* $\text{productive } C \implies \text{produces } C (\text{Max } (\text{atms-of } C))$
 $\langle \text{proof} \rangle$

lemma *produces-imp-Max-literal:* $\text{produces } C A \implies A = \text{atm-of } (\text{Max } (\text{set-mset } C))$
 $\langle \text{proof} \rangle$

lemma *produces-imp-Max-atom:* $\text{produces } C A \implies A = \text{Max } (\text{atms-of } C)$
 $\langle \text{proof} \rangle$

lemma *produces-imp-Pos-in-lits:* $\text{produces } C A \implies \text{Pos } A \in \# C$
 $\langle \text{proof} \rangle$

lemma *productive-in-N:* $\text{productive } C \implies C \in N$
 $\langle \text{proof} \rangle$

lemma *produces-imp-atms-leq:* $\text{produces } C A \implies B \in \text{atms-of } C \implies B \leq A$
 $\langle \text{proof} \rangle$

lemma *produces-imp-neg-notin-lits:* $\text{produces } C A \implies \neg \text{Neg } A \in \# C$
 $\langle \text{proof} \rangle$

lemma *less-eq-imp-interp-subseteq-interp:* $C \# \subseteq \# D \implies \text{interp } C \subseteq \text{interp } D$
 $\langle \text{proof} \rangle$

lemma *less-eq-imp-interp-subseteq-Interp:* $C \# \subseteq \# D \implies \text{interp } C \subseteq \text{Interp } D$
 $\langle \text{proof} \rangle$

lemma *less-imp-production-subseteq-interp:* $C \# \subset \# D \implies \text{production } C \subseteq \text{interp } D$
 $\langle \text{proof} \rangle$

lemma *less-eq-imp-production-subseteq-Interp:* $C \# \subseteq \# D \implies \text{production } C \subseteq \text{Interp } D$
 $\langle \text{proof} \rangle$

lemma *less-imp-Interp-subseteq-interp:* $C \# \subset \# D \implies \text{Interp } C \subseteq \text{interp } D$
 $\langle \text{proof} \rangle$

lemma *less-eq-imp-Interp-subseteq-Interp:* $C \# \subseteq \# D \implies \text{Interp } C \subseteq \text{Interp } D$
 $\langle \text{proof} \rangle$

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin \text{Interp } C \implies A \in \text{interp } D \implies C \# \subset \# D$
 $\langle \text{proof} \rangle$

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin \text{interp } C \implies A \in \text{interp } D \implies C \# \subset \# D$
 $\langle \text{proof} \rangle$

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin \text{Interp } C \implies A \in \text{Interp } D \implies C \# \subset \# D$
 $\langle \text{proof} \rangle$

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin \text{interp } C \implies A \in \text{Interp } D \implies C \# \subseteq \# D$
 $\langle \text{proof} \rangle$

lemma *interp-subseteq-INTERP*: $\text{interp } C \subseteq \text{INTERP}$
 $\langle \text{proof} \rangle$

lemma *production-subseteq-INTERP*: $\text{production } C \subseteq \text{INTERP}$
 $\langle \text{proof} \rangle$

lemma *Interp-subseteq-INTERP*: $\text{Interp } C \subseteq \text{INTERP}$
 $\langle \text{proof} \rangle$

This lemma corresponds to theorem 2.7.6 page 66 of CW.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: $\text{Neg } A \in \# C$ **and** *d*: *produces* $D A$
shows $A \in \text{interp } C$
 $\langle \text{proof} \rangle$

lemma *neg-notin-Interp-not-produce*: $\text{Neg } A \in \# C \implies A \notin \text{Interp } D \implies C \# \subseteq \# D \implies \neg \text{produces } D'' A$
 $\langle \text{proof} \rangle$

lemma *in-production-imp-produces*: $A \in \text{production } C \implies \text{produces } C A$
 $\langle \text{proof} \rangle$

lemma *not-produces-imp-notin-production*: $\neg \text{produces } C A \implies A \notin \text{production } C$
 $\langle \text{proof} \rangle$

lemma *not-produces-imp-notin-interp*: $(\bigwedge D. \neg \text{produces } D A) \implies A \notin \text{interp } C$
 $\langle \text{proof} \rangle$

The results below corresponds to Lemma 3.4.

Nitpicking: If $D = D'$ and D is productive, $I^D \subseteq I_{D'}$ does not hold.

lemma *true-Interp-imp-general*:
assumes
c-le-d: $C \# \subseteq \# D$ **and**
d-lt-d': $D \# \subset \# D'$ **and**
c-at-d: $\text{Interp } D \models_h C$ **and**
subs: $\text{interp } D' \subseteq (\bigcup C \in CC. \text{production } C)$
shows $(\bigcup C \in CC. \text{production } C) \models_h C$
 $\langle \text{proof} \rangle$

lemma *true-Interp-imp-interp*: $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{interp } D' \models_h C$
 $\langle \text{proof} \rangle$

lemma *true-Interp-imp-Interp*: $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{Interp } D \models_h C \implies \text{Interp } D' \models_h C$

$\langle \text{proof} \rangle$

lemma *true-Interp-imp-INTERP*: $C \# \subseteq \# D \implies \text{Interp } D \models_h C \implies \text{INTERP} \models_h C$
 $\langle \text{proof} \rangle$

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 $(\bigcup C \in CC. \text{production } C) \models_h C$

$\langle \text{proof} \rangle$

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

lemma *true-interp-imp-interp*: $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{interp } D' \models_h C$
 $\langle \text{proof} \rangle$

lemma *true-interp-imp-Interp*: $C \# \subseteq \# D \implies D \# \subset \# D' \implies \text{interp } D \models_h C \implies \text{Interp } D' \models_h C$
 $\langle \text{proof} \rangle$

lemma *true-interp-imp-INTERP*: $C \# \subseteq \# D \implies \text{interp } D \models_h C \implies \text{INTERP} \models_h C$
 $\langle \text{proof} \rangle$

lemma *productive-imp-false-interp*: $\text{productive } C \implies \neg \text{interp } C \models_h C$
 $\langle \text{proof} \rangle$

This lemma corresponds to theorem 2.7.6 page 66 of CW. Here the strict maximality is important

lemma *cls-gt-double-pos-no-production*:

assumes $D: \{\#Pos P, Pos P\} \# \subset \# C$

shows $\neg \text{produces } C P$

$\langle \text{proof} \rangle$

This lemma corresponds to theorem 2.7.6 page 66 of CW.

lemma

assumes $D: C + \{\#Neg P\} \# \subset \# D$

shows $\text{production } D \neq \{P\}$

$\langle \text{proof} \rangle$

lemma *in-interp-is-produced*:

assumes $P \in \text{INTERP}$

shows $\exists D. D + \{\#Pos P\} \in N \wedge \text{produces } (D + \{\#Pos P\}) P$

$\langle \text{proof} \rangle$

end

end

abbreviation $MMax M \equiv Max (\text{set-mset } M)$

25.1 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: $A \in N \Rightarrow B \in N \Rightarrow \text{superposition-rules } A \ B \ C$
 $\Rightarrow \text{superposition } N \ (N \cup \{C\})$

definition *abstract-red* :: 'a::wellorder clause \Rightarrow 'a clauses \Rightarrow bool **where**
abstract-red $C \ N = (\text{clss-lt } N \ C \models_p C)$

lemma *less-multiset[iff]*: $M < N \longleftrightarrow M \# \subset \# N$
 $\langle \text{proof} \rangle$

lemma *less-eq-multiset[iff]*: $M \leq N \longleftrightarrow M \# \subseteq \# N$
 $\langle \text{proof} \rangle$

lemma *herbrand-true-clss-true-clss-clss-herbrand-true-clss*:

assumes

$AB: A \models_{hs} B$ **and**

$BC: B \models_p C$

shows $A \models_h C$

$\langle \text{proof} \rangle$

lemma *abstract-red-subset-mset-abstract-red*:

assumes

$\text{abstr}: \text{abstract-red } C \ N$ **and**

$c\text{-lt-d}: C \subseteq \# D$

shows $\text{abstract-red } D \ N$

$\langle \text{proof} \rangle$

lemma *true-clss-clss-extended*:

assumes

$A \models_p B$ **and**

$\text{tot}: \text{total-over-m } I \ (A)$ **and**

$\text{cons}: \text{consistent-interp } I$ **and**

$I\text{-}A: I \models_s A$

shows $I \models B$

$\langle \text{proof} \rangle$

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 $\text{clss-lt } N \ (\{\#C\# \} + \{\#E\# \}) \models_p \{\#E\# \} + \{\#Pos \ P\# \}$

$\langle \text{proof} \rangle$

locale *ground-ordered-resolution-with-redundancy* =

ground-resolution-with-selection +

fixes *redundant* :: 'a::wellorder clause \Rightarrow 'a clauses \Rightarrow bool

assumes

redundant-iff-abstract: $\text{redundant } A \ N \longleftrightarrow \text{abstract-red } A \ N$

begin

definition *saturated* :: 'a clauses \Rightarrow bool **where**

saturated $N \longleftrightarrow (\forall A \ B \ C. A \in N \longrightarrow B \in N \longrightarrow \neg \text{redundant } A \ N \longrightarrow \neg \text{redundant } B \ N$
 $\longrightarrow \text{superposition-rules } A \ B \ C \longrightarrow \text{redundant } C \ N \vee C \in N)$

```

lemma
  assumes
    saturated: saturated N and
    finite: finite N and
    empty:  $\{\#\} \notin N$ 
  shows  $INTERP\ N \models_{hs} N$ 
   $\langle proof \rangle$ 

```

end

```

lemma tautology-is-redundant:
  assumes tautology C
  shows abstract-red C N
   $\langle proof \rangle$ 

```

```

lemma subsumed-is-redundant:
  assumes AB:  $A \subset\# B$ 
  and AN:  $A \in N$ 
  shows abstract-red B N
   $\langle proof \rangle$ 

```

```

inductive redundant :: 'a clause  $\Rightarrow$  'a clauses  $\Rightarrow$  bool where
  subsumption:  $A \in N \Longrightarrow A \subset\# B \Longrightarrow \text{redundant } B\ N$ 

```

```

lemma redundant-is-redundancy-criterion:
  fixes A :: 'a :: wellorder clause and N :: 'a :: wellorder clauses
  assumes redundant A N
  shows abstract-red A N
   $\langle proof \rangle$ 

```

```

lemma redundant-mono:
   $\text{redundant } A\ N \Longrightarrow A \subseteq\# B \Longrightarrow \text{redundant } B\ N$ 
   $\langle proof \rangle$ 

```

```

locale truc =
  selection S for S :: nat clause  $\Rightarrow$  nat clause
begin

```

end

end