Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

March 31, 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 *upt* 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 . . . 26
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 . . . 41
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 Decided 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 Decided Literals . . . 82
14.3.1 Definition . . . 82
14.3.2 Entailment of the Propagated by the Decided 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 . . . 94
16.2.1 The state . . . 94
16.2.2 Definition of the operation . . . 97
16.3 DPLL with backjumping . . . 99
16.3.1 Definition . . . 100
16.3.2 Basic properties . . . 100
16.3.3 Termination . . . 101
16.3.4 Normal Forms . . . 102
16.4 CDCL . . . 106
16.4.1 Learn and Forget . . . 106
16.4.2 Definition of CDCL . . . 108
16.4.3 CDCL with invariant . . . 110
16.4.4 Termination . . . 112
16.4.5 Restricting learn and forget . . . 112
16.5 CDCL with restarts . . . 117
16.5.1 Definition . . . 117
16.5.2 Increasing restarts . . . 118
16.6 Merging backjump and learning . . . 123
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 138
18.1 Rules . . . 138
18.2 Invariants . . . 138
18.3 Termination . . . 140
18.4 Final States . . . 141
18.5 Link with NOT's DPLL . . . 142
18.5.1 Level of literals and clauses . . . 142
18.5.2 Properties about the levels . . . 146

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 . . . 169
19.3.5 Learned Clause . . . 170
19.3.6 No alien atom in the state . . . 170
19.3.7 No duplicates all around . . . 172
19.3.8 Conflicts . . . 172
19.3.9 Putting all the invariants together . . . 174
19.3.10 No tautology is learned . . . 176
19.4 CDCL Strong Completeness . . . 176
19.5 Higher level strategy . . . 177
19.5.1 Definition . . . 177
19.5.2 Invariants . . . 179
19.5.3 Literal of highest level in conflicting clauses . . . 181
19.5.4 Literal of highest level in decided literals . . . 182
19.5.5 Strong completeness . . . 184
19.5.6 No conflict with only variables of level less than backtrack level . . . 186
19.5.7 Final States are Conclusive . . . 189
19.6 Termination . . . 191
19.7 No Relearning of a clause . . . 191
19.8 Decrease of a measure . . . 195

20 Simple Implementation of the DPLL and CDCL 197
20.1 Common Rules . . . 197
20.1.1 Propagation . . . 197
20.1.2 Unit propagation for all clauses . . . 198
20.1.3 Decide . . . 198
20.2 Simple Implementation of DPLL . . . 199
20.2.1 Combining the propagate and decide: a DPLL step . . . 199
20.2.2 Adding invariants . . . 200
20.2.3 Code export . . . 202
20.3 CDCL Implementation . . . 206
20.3.1 Types and Additional Lemmas . . . 206
20.3.2 The Transitions . . . 208
20.3.3 Code generation . . . 213

21 Merging backjump rules **219**
21.1 Inclusion of the states 219
21.2 More lemmas conflict–propagate and backjumping 220
21.2.1 Termination 220
21.2.2 More backjumping 221
21.3 CDCL FW 223
21.4 FW with strategy 224
21.4.1 The intermediate step 224
21.4.2 Full Transformation 227
21.4.3 Termination and full Equivalence 231
21.5 Adding Restarts 232

22 Link between Weidenbach's and NOT's CDCL **237**
22.1 Inclusion of the states 237
22.2 Additional Lemmas between NOT and W states 240
22.3 More lemmas conflict–propagate and backjumping 241
22.4 CDCL FW 241

23 Incremental SAT solving **242**

24 2-Watched-Literal **245**
24.1 Essence of 2-WL 246
24.1.1 Datastructure and Access Functions 246
24.1.2 Invariants 248
24.1.3 Abstract 2-WL 250
24.1.4 Instanciation of the previous locale 252
24.2 Two Watched-Literals with invariant 255
24.2.1 Interpretation for $\mathit{conflict\text{-}driven\text{-}clause\text{-}learning}_W.\mathit{cdcl}_W$ 256

25 Superposition **261**
25.1 We can now define the rules of the calculus 266

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 $?r \leq ?s \Longrightarrow ?r^{**} \leq ?s^{**}$ for *tranclp*

lemma *tranclp-mono-explicit*:
$r^{++}\ a\ b \Longrightarrow r \leq s \Longrightarrow s^{++}\ a\ b$
⟨*proof*⟩

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

shows $r^{++} \le s^{++}$
⟨*proof*⟩

lemma *tranclp-idemp-rel*:
$R^{++++}\ a\ b \longleftrightarrow R^{++}\ a\ b$
⟨*proof*⟩

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

lemma *trancl-idemp*: $(r^{+})^{+} = r^{+}$
⟨*proof*⟩

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

This theorem already exists as $?r^{**}\ ?a\ ?b \equiv ?a = ?b \lor ?r^{++}\ ?a\ ?b$ (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*: $\textit{rtranclp}\ r\ a\ b \longleftrightarrow (a = b \lor \textit{tranclp}\ r\ a\ b)$
⟨*proof*⟩

lemma *tranclp-unfold-end*: $\textit{tranclp}\ r\ a\ b \longleftrightarrow (\exists\, a'.\ \textit{rtranclp}\ r\ a\ a' \land r\ a'\ b)$
⟨*proof*⟩

Near duplicate of $?R^{++}\ ?x\ ?y \Longrightarrow \exists\, z.\ ?R\ ?x\ z \land ?R^{**}\ z\ ?y$:

lemma *tranclp-unfold-begin*: $\textit{tranclp}\ r\ a\ b \longleftrightarrow (\exists\, a'.\ r\ a\ a' \land \textit{rtranclp}\ r\ a'\ b)$
⟨*proof*⟩

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

lemma *tranclp-rtranclp-rtranclp-rel*: $R^{++**}\ a\ b \longleftrightarrow R^{**}\ a\ b$
⟨*proof*⟩

lemma *tranclp-rtranclp-rtranclp*[*simp*]: $R^{++**} = R^{**}$
⟨*proof*⟩

lemma *rtranclp-exists-last-with-prop*:
assumes $R\ x\ z$
and $R^{**}\ z\ z'$ **and** $P\ x\ z$
shows $\exists\, y\ y'.\ R^{**}\ x\ y \land R\ y\ y' \land P\ y\ y' \land (\lambda a\ b.\ R\ a\ b \land \neg P\ a\ b)^{**}\ y'\ z'$
⟨*proof*⟩

lemma *rtranclp-and-rtranclp-left*: $(\lambda\ a\ b.\ P\ a\ b \land Q\ a\ b)^{**}\ S\ T \Longrightarrow P^{**}\ S\ T$
⟨*proof*⟩

1.2 Full Transitions

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

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

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

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

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

We define output notations only for printing:

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

lemma *rtranclp-full1I*:
$R^{**}\ a\ b \Longrightarrow full1\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
⟨*proof*⟩

lemma *tranclp-full1I*:
$R^{++}\ a\ b \Longrightarrow full1\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
⟨*proof*⟩

lemma *rtranclp-fullI*:
$R^{**}\ a\ b \Longrightarrow full\ R\ b\ c \Longrightarrow full\ R\ a\ c$
⟨*proof*⟩

lemma *tranclp-full-full1I*:
$R^{++}\ a\ b \Longrightarrow full\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
⟨*proof*⟩

lemma *full-fullI*:
$R\ a\ b \Longrightarrow full\ R\ b\ c \Longrightarrow full1\ R\ a\ c$
⟨*proof*⟩

lemma *full-unfold*:
$full\ r\ S\ S' \longleftrightarrow ((S = S' \wedge no\text{-}step\ r\ S') \vee full1\ r\ S\ S')$
⟨*proof*⟩

lemma *full1-is-full*[*intro*]: $full1\ R\ S\ T \Longrightarrow full\ R\ S\ T$
⟨*proof*⟩

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

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

lemma *full1-tranclp-relation-full*:
$full1\ R^{++}\ a\ b \longleftrightarrow full1\ R\ a\ b$
⟨*proof*⟩

lemma *full-tranclp-relation-full*:
$full\ R^{++}\ a\ b \longleftrightarrow full\ R\ a\ b$
⟨*proof*⟩

lemma *rtranclp-full1-eq-or-full1*:
$(full1\ R)^{**}\ a\ b \longleftrightarrow (a = b \vee full1\ R\ a\ b)$
⟨*proof*⟩

lemma *tranclp-full1-full1*:
$(full1\ R)^{++}\ a\ b \longleftrightarrow full1\ R\ a\ b$
⟨*proof*⟩

1.3 Well-Foundedness and Full Transitions

lemma *wf-exists-normal-form*:
  assumes *wf*:$wf\ \{(x,\ y).\ R\ y\ x\}$
  shows $\exists\, b.\ R^{**}\ a\ b \wedge no\text{-}step\ R\ b$
$\langle proof\rangle$

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: $wf\ ?r = (\nexists f.\ \forall\, i.\ (f\ (Suc\ i),\ f\ i) \in\ ?r)$, $[\![wf\ ?r;\ \bigwedge k.\ (?f\ (Suc\ k),\ ?f\ k) \notin\ ?r \Longrightarrow\ ?thesis]\!] \Longrightarrow\ ?thesis$

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

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

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: $(\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow (f\ y,\ f\ x) \in r)$
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 $(\bigwedge x\ y.\ P\ x\ y \Longrightarrow g\ x\ y\ \Longrightarrow f\ x < f\ y) \Longrightarrow wf\ \{(x,y).\ P\ x\ y \wedge g\ x\ y\}$
  $\langle proof\rangle$

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

lemma *wf-fst-wf-pair*:

assumes *wf* $\{(M', M).\ R\ M'\ M\}$
shows *wf* $\{((M', N'), (M, N)).\ R\ M'\ M\}$
⟨*proof*⟩

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

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

lemma *wf-wf-if-measure′-notation2*:
assumes *wf r* **and** *H*: $(\bigwedge x\ y.\ P\ x \Longrightarrow g\ x\ y \Longrightarrow (f\ y, f\ (h\ x)) \in r)$
shows *wf* $\{(y,h\ x)|\ y\ x.\ P\ x \wedge g\ x\ y\}$
⟨*proof*⟩

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

Sledgehammer parameters

sledgehammer-params[*debug*]

2 Various Lemmas

Close to $(\bigwedge n.\ \forall m{<}n.\ ?P\ m \Longrightarrow ?P\ n) \Longrightarrow ?P\ ?n$, 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**
$\bigwedge n.\ (\forall m < Suc\ n.\ P\ m) \Longrightarrow P\ (Suc\ n)$
shows $P\ n$
⟨*proof*⟩

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*]:
$0 < (if\ P\ then\ a\ else\ (0::nat)) \longleftrightarrow P \wedge 0 < a$
⟨*proof*⟩

Bounded function have not yet been defined in Isabelle.

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

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* $\longleftrightarrow$ *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 $[?i..<Suc\ ?j] = (if\ ?i \le\ ?j\ then\ [?i..<?j]\ @\ [?j]\ else\ [])$ leads to a case distinction, that we do not want if the condition is not in the context.

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

A more restrictive version holds:

lemma $B \neq [] \Longrightarrow A\ @\ B = [m..< n] \longleftrightarrow A = [m\ ..<m+\mathit{length}\ A] \wedge B = [m + \mathit{length}\ A..<n]$
(**is** $?P \Longrightarrow ?A = ?B$)
⟨*proof*⟩

lemma *append-cons-eq-upt-length-i*:
assumes $A\ @\ i\ \#\ B = [m..<n]$
shows $A = [m\ ..<i]$
⟨*proof*⟩

lemma *append-cons-eq-upt-length*:
assumes $A\ @\ i\ \#\ B = [m..<n]$
shows $\mathit{length}\ A = i - m$
⟨*proof*⟩

lemma *append-cons-eq-upt-length-i-end*:
assumes $A\ @\ i\ \#\ B = [m..<n]$
shows $B = [\mathit{Suc}\ i\ ..<n]$
⟨*proof*⟩

lemma *Max-n-upt*: $\mathit{Max}\ (\mathit{insert}\ 0\ \{\mathit{Suc}\ 0..<n\}) = n - \mathit{Suc}\ 0$
⟨*proof*⟩

lemma *upt-decomp-lt*:
assumes H: $xs\ @\ i\ \#\ ys\ @\ j\ \#\ zs = [m\ ..< n]$
shows $i < j$
⟨*proof*⟩

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

lemma *length-list-Suc-0*:
$\mathit{length}\ W = \mathit{Suc}\ 0 \longleftrightarrow (\exists L.\ W = [L])$
⟨*proof*⟩

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

lemma *finite-bounded-list*:
fixes $b :: \mathit{nat}$
shows $\mathit{finite}\ \{xs.\ \mathit{length}\ xs < s \wedge (\forall i< \mathit{length}\ xs.\ xs\ !\ i < b)\}$ (**is** $\mathit{finite}\ (?S\ s)$)
⟨*proof*⟩

3.2 Lexicographic Ordering

lemma *lexn-Suc*:
$(x\ \#\ xs, y\ \#\ ys) \in \mathit{lexn}\ r\ (\mathit{Suc}\ n) \longleftrightarrow$
$(\mathit{length}\ xs = n \wedge \mathit{length}\ ys = n) \wedge ((x, y) \in r \vee (x = y \wedge (xs, ys) \in \mathit{lexn}\ r\ n))$

$\langle proof \rangle$

lemma *lexn-n*:
$n > 0 \Longrightarrow (x \# xs,\ y \# ys) \in lexn\ r\ n \longleftrightarrow$
$(length\ xs = n{-}1 \wedge length\ ys = n{-}1) \wedge ((x,\ y) \in r \vee (x = y \wedge (xs,\ ys) \in lexn\ r\ (n - 1)))$
$\langle proof \rangle$

There is some subtle point in the proof here. *1* is converted to *Suc 0*, but *2* is not: meaning that *1* is automatically simplified by default using the default simplification rule $lexn\ ?r\ 0 = \{\}$
$lexn\ ?r\ (Suc\ ?n) = map\text{-}prod\ (\lambda(x,\ xs).\ x \# xs)\ (\lambda(x,\ xs).\ x \# xs)\ \text{`}\ (?r <*lex*> lexn\ ?r\ ?n) \cap \{(xs,\ ys).\ length\ xs = Suc\ ?n \wedge length\ ys = Suc\ ?n\}$. However, the latter needs additional simplification rule (see the proof of the theorem above).

lemma *lexn2-conv*:
$([a,\ b],\ [c,\ d]) \in lexn\ r\ 2 \longleftrightarrow (a,\ c) \in r \vee (a = c \wedge (b,\ d) \in r)$
$\langle proof \rangle$

lemma *lexn3-conv*:
$([a,\ b,\ c],\ [a',\ b',\ c']) \in lexn\ r\ 3 \longleftrightarrow$
$(a,\ a') \in r \vee (a = a' \wedge (b,\ b') \in r) \vee (a = a' \wedge b = b' \wedge (c,\ c') \in r)$
$\langle proof \rangle$

3.3 Remove

3.3.1 More lemmas about remove

lemma *remove1-nil*:
$remove1\ (-\ L)\ W = [] \longleftrightarrow (W = [] \vee W = [-L])$
$\langle proof \rangle$

lemma *remove1-mset-single-add*:
$a \neq b \Longrightarrow remove1\text{-}mset\ a\ (\{\#b\#\} + C) = \{\#b\#\} + remove1\text{-}mset\ a\ C$
$remove1\text{-}mset\ a\ (\{\#a\#\} + C) = C$
$\langle proof \rangle$

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

lemma $remove1\ x\ xs = remove1\text{-}cond\ ((op =)\ x)\ xs$
$\langle proof \rangle$

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

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

fun *removeAll-cond* **where**
$removeAll\text{-}cond\ f\ [] = []\ |$
$removeAll\text{-}cond\ f\ (C' \# L) =$

(*if* $f\ C'$ *then removeAll-cond* $f\ L$ *else* C' # *removeAll-cond* $f\ L$)

lemma *removeAll x xs* = *removeAll-cond* ((*op* =) *x*) *xs*
⟨*proof*⟩

lemma *removeAll-cond P xs* = *filter* ($\lambda x.\ \neg P\ x$) *xs*
⟨*proof*⟩

lemma *mset-map-mset-removeAll-cond*:
mset (*map mset* (*removeAll-cond* ($\lambda b.$ *mset b* = *mset a*) *C*))
= *removeAll-mset* (*mset a*) (*mset* (*map mset C*))
⟨*proof*⟩

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

abbreviation *union-mset-list* **where**
union-mset-list xs ys ≡ *case-prod append* (*fold* (λx (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))

lemma *union-mset-list*:
mset xs #∪ *mset ys* = *mset* (*union-mset-list xs 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 | *FF* | *FVar 'v* | *FNot 'v propo* | *FAnd 'v propo 'v propo* | *FOr 'v propo 'v propo*
| *FImp 'v propo 'v propo* | *FEq 'v propo 'v propo*

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

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

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

abbreviation *nullary-connective* ≡ {*CF*} ∪ {*CT*} ∪ {*CVar x* | *x. True*}
definition *binary-connectives* ≡ {*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*: $(\bigwedge\varphi\ x.\ \varphi = FF \vee \varphi = FT \vee \varphi = FVar\ x \Longrightarrow P\ \varphi)$
and *unary*: $(\bigwedge\psi.\ P\ \psi \Longrightarrow P\ (FNot\ \psi))$
and *binary*: $(\bigwedge\varphi\ \psi 1\ \psi 2.\ P\ \psi 1 \Longrightarrow P\ \psi 2 \Longrightarrow \varphi = FAnd\ \psi 1\ \psi 2 \vee \varphi = FOr\ \psi 1\ \psi 2 \vee \varphi = FImp\ \psi 1\ \psi 2$
$\vee\ \varphi = FEq\ \psi 1\ \psi 2 \Longrightarrow P\ \varphi)$
shows $P\ \psi$
⟨*proof*⟩

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

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

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

lemma *connective-cases-arity*[*case-names nullary binary unary*]:
assumes *nullary*: $\bigwedge x.\ c = CT \vee c = CF \vee c = CVar\ x \Longrightarrow P$
and *binary*: $c \in binary\text{-}connectives \Longrightarrow P$
and *unary*: $c = CNot \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *connective-cases-arity-2*[*case-names nullary unary binary*]:
assumes *nullary*: $c \in nullary\text{-}connective \Longrightarrow P$
and *unary*: $c = CNot \Longrightarrow P$
and *binary*: $c \in binary\text{-}connectives \Longrightarrow P$
shows P
⟨*proof*⟩

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

inductive $wf\text{-}conn :: 'v\ connective \Rightarrow 'v\ propo\ list \Rightarrow bool$ **for** $c :: 'v\ connective$ **where**
wf-conn-nullary[*simp*]: $(c = CT \vee c = CF \vee c = CVar\ v) \Longrightarrow wf\text{-}conn\ c\ []\ |$
wf-conn-unary[*simp*]: $c = CNot \Longrightarrow wf\text{-}conn\ c\ [\psi]\ |$
wf-conn-binary[*simp*]: $c \in binary\text{-}connectives \Longrightarrow wf\text{-}conn\ c\ (\psi\ \#\ \psi'\ \#\ [])$
thm *wf-conn.induct*
lemma *wf-conn-induct*[*consumes 1*, *case-names CT CF CVar CNot COr CAnd CImp CEq*]:
assumes $wf\text{-}conn\ c\ x$ **and**
$(\bigwedge v.\ c = CT \Longrightarrow P\ [])$ **and**
$(\bigwedge v.\ c = CF \Longrightarrow P\ [])$ **and**
$(\bigwedge v.\ c = CVar\ v \Longrightarrow P\ [])$ **and**
$(\bigwedge \psi.\ c = CNot \Longrightarrow P\ [\psi])$ **and**
$(\bigwedge \psi\ \psi'.\ c = COr \Longrightarrow P\ [\psi, \psi'])$ **and**
$(\bigwedge \psi\ \psi'.\ c = CAnd \Longrightarrow P\ [\psi, \psi'])$ **and**
$(\bigwedge \psi\ \psi'.\ c = CImp \Longrightarrow P\ [\psi, \psi'])$ **and**

$(\bigwedge \psi\ \psi'.\ c = CEq \Longrightarrow P\ [\psi,\ \psi'])$
shows $P\ x$
$\langle proof \rangle$

4.2 properties of the abstraction

First we can define simplification rules.

lemma *wf-conn-conn*[*simp*]:
wf-conn $CT\ l \Longrightarrow$ *conn* $CT\ l = FT$
wf-conn $CF\ l \Longrightarrow$ *conn* $CF\ l = FF$
wf-conn $(CVar\ x)\ l \Longrightarrow$ *conn* $(CVar\ x)\ l = FVar\ x$
$\langle proof \rangle$

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 $c\ l \Longrightarrow$ *conn* $c\ l = FT \longleftrightarrow (c = CT \wedge l = [])$
wf-conn $c\ l \Longrightarrow$ *conn* $c\ l = FF \longleftrightarrow (c = CF \wedge l = [])$
wf-conn $c\ l \Longrightarrow$ *conn* $c\ l = FVar\ x \longleftrightarrow (c = CVar\ x \wedge l = [])$
wf-conn $c\ l \Longrightarrow$ *conn* $c\ l = FAnd\ a\ b \longleftrightarrow (c = CAnd \wedge l = a\ \#\ b\ \#\ [])$
wf-conn $c\ l \Longrightarrow$ *conn* $c\ l = FOr\ a\ b \longleftrightarrow (c = COr \wedge l = a\ \#\ b\ \#\ [])$
wf-conn $c\ l \Longrightarrow$ *conn* $c\ l = FEq\ a\ b \longleftrightarrow (c = CEq \wedge l = a\ \#\ b\ \#\ [])$
wf-conn $c\ l \Longrightarrow$ *conn* $c\ l = FImp\ a\ b \longleftrightarrow (c = CImp \wedge l = a\ \#\ b\ \#\ [])$
wf-conn $c\ l \Longrightarrow$ *conn* $c\ l = FNot\ a \longleftrightarrow (c = CNot \wedge l = a\ \#\ [])$
$\langle proof \rangle$

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

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

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 $\exists\ a.\ l = [a]$
⟨*proof*⟩

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

lemma *wf-conn-no-arity-change*:
$length\ l = length\ l' \Longrightarrow wf\text{-}conn\ c\ l \longleftrightarrow wf\text{-}conn\ c\ l'$
⟨*proof*⟩

lemma *wf-conn-no-arity-change-helper*:
$length\ (\xi\ @\ \varphi\ \#\ \xi') = length\ (\xi\ @\ \varphi'\ \#\ \xi')$
⟨*proof*⟩

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

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

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

4.3 Subformulas and properties

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

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

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

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:
shows *b*: $FNot\ \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi$
⟨*proof*⟩

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

lemma *subformula-trans*:
$\psi \preceq \psi' \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi'$

⟨*proof*⟩

lemma *subformula-leaf*:
fixes $\varphi\ \psi :: {}'v\ propo$
assumes *incl*: $\varphi \preceq \psi$
and *simple*: $\psi = FT \vee \psi = FF \vee \psi = FVar\ x$
shows $\varphi = \psi$
⟨*proof*⟩

lemma *subfurmula-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 $\varphi = conn\ c\ l$
⟨*proof*⟩

lemma *wf-subformula-conn-cases*:
$wf\text{-}conn\ c\ l \implies \varphi \preceq conn\ c\ l \longleftrightarrow (\varphi = conn\ c\ l \vee (\exists \psi.\ \psi \in set\ l \wedge \varphi \preceq \psi))$
⟨*proof*⟩

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

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 $[\varphi, \psi]$
⟨*proof*⟩

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

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** $?A \longleftrightarrow ?B$)
⟨*proof*⟩

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

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

primrec *vars-of-prop*:: $'v\ propo \Rightarrow {}'v\ set$ **where**
vars-of-prop $FT = \{\}$ |

vars-of-prop FF = {} |
vars-of-prop (*FVar x*) = {*x*} |
vars-of-prop (*FNot* φ) = *vars-of-prop* φ |
vars-of-prop (*FAnd* φ ψ) = *vars-of-prop* φ ∪ *vars-of-prop* ψ |
vars-of-prop (*FOr* φ ψ) = *vars-of-prop* φ ∪ *vars-of-prop* ψ |
vars-of-prop (*FImp* φ ψ) = *vars-of-prop* φ ∪ *vars-of-prop* ψ |
vars-of-prop (*FEq* φ ψ) = *vars-of-prop* φ ∪ *vars-of-prop* ψ

lemma *vars-of-prop-incl-conn*:
  fixes ξ ξ' :: *'v propo list* **and** ψ :: *'v propo* **and** *c* :: *'v connective*
  assumes *corr*: *wf-conn c l* **and** *incl*: ψ ∈ *set l*
  shows *vars-of-prop* ψ ⊆ *vars-of-prop* (*conn c l*)
⟨*proof*⟩

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:
  $\varphi \preceq \psi \Longrightarrow$ *vars-of-prop* φ ⊆ *vars-of-prop* ψ
  ⟨*proof*⟩

4.4 Positions

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

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

We use *nil* instead of ε.

fun *pos* :: *'v propo* ⇒ *sign list set* **where**
pos FF = {[]} |
pos FT = {[]} |
pos (*FVar x*) = {[]} |
pos (*FAnd* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FOr* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FEq* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FImp* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FNot* φ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ}

lemma *finite-pos*: *finite* (*pos* φ)
  ⟨*proof*⟩

lemma *finite-inj-comp-set*:
  fixes *s* :: *'v set*
  assumes *finite*: *finite s*
  and *inj*: *inj f*
  shows *card* ({*f p* |*p*. *p* ∈ *s*}) = *card s*
  ⟨*proof*⟩

lemma *cons-inject*:
  inj (*op* # *s*)
  ⟨*proof*⟩

lemma *finite-insert-nil-cons*:
  finite s ⟹ *card* (*insert* [] {*L* # *p* |*p*. *p* ∈ *s*}) = *1* + *card* {*L* # *p* |*p*. *p* ∈ *s*}
  ⟨*proof*⟩

lemma *cord-not*[*simp*]:

$card\ (pos\ (FNot\ \varphi)) = 1 + card\ (pos\ \varphi)$
$\langle proof \rangle$

lemma *card-seperate*:
assumes *finite s1* **and** *finite s2*
shows $card\ (\{L \# p\ |p.\ p \in s1\} \cup \{R \# p\ |p.\ p \in s2\}) = card\ (\{L \# p\ |p.\ p \in s1\}) + card(\{R \# p\ |p.\ p \in s2\})$ (**is** $card\ (?L \cup ?R) = card\ ?L + card\ ?R$)
$\langle proof \rangle$

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

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

$\langle proof \rangle$

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

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

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

lemma *path-to-subformula*:
$path\text{-}to\ p\ \varphi\ \varphi' \Longrightarrow \varphi' \preceq \varphi$
$\langle proof \rangle$

lemma *subformula-path-exists*:
fixes $\varphi\ \varphi' :: 'v\ propo$
shows $\varphi' \preceq \varphi \Longrightarrow \exists p.\ path\text{-}to\ p\ \varphi\ \varphi'$
$\langle proof \rangle$

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

5 Semantics over the syntax

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

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

$\mathcal{A} \models FT = True$ |
$\mathcal{A} \models FF = False$ |
$\mathcal{A} \models FVar\ v = (\mathcal{A}\ v)$ |
$\mathcal{A} \models FNot\ \varphi = (\neg(\mathcal{A}\models \varphi))$ |
$\mathcal{A} \models FAnd\ \varphi_1\ \varphi_2 = (\mathcal{A}\models\varphi_1 \wedge \mathcal{A}\models\varphi_2)$ |
$\mathcal{A} \models FOr\ \varphi_1\ \varphi_2 = (\mathcal{A}\models\varphi_1 \vee \mathcal{A}\models\varphi_2)$ |
$\mathcal{A} \models FImp\ \varphi_1\ \varphi_2 = (\mathcal{A}\models\varphi_1 \longrightarrow \mathcal{A}\models\varphi_2)$ |
$\mathcal{A} \models FEq\ \varphi_1\ \varphi_2 = (\mathcal{A}\models\varphi_1 \longleftrightarrow \mathcal{A} \models\varphi_2)$

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

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

theorem *deduction-theorem*:
$(\varphi \models f\ \psi) \longleftrightarrow (\forall A.\ (A \models FImp\ \varphi\ \psi))$
⟨*proof*⟩

A shorter proof:

lemma $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A\models FImp\ \varphi\ \psi)$
⟨*proof*⟩

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

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

lemma *same-over-set-eval*:
assumes *same-over-set* $A\ B$ (*vars-of-prop* φ)
shows $A \models \varphi \longleftrightarrow B \models \varphi$
⟨*proof*⟩

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 full1 formula: the relation r works on terms, while *propo-rew-step* works on formulas.

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

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

This lemma is only a health condition:

lemma *propo-rew-step-subformula-imp*:
shows *propo-rew-step* $r\ \varphi\ \varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r\ \psi\ \psi'$
⟨*proof*⟩

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

lemma *propo-rew-step-subformula-rec*:
fixes $\psi\ \psi'\ \varphi$ :: *'v propo*
shows $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists\varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step* $r\ \varphi\ \varphi')$
⟨*proof*⟩

lemma *propo-rew-step-subformula*:
$(\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi') \longleftrightarrow (\exists\varphi'.$ *propo-rew-step* $r\ \varphi\ \varphi')$
⟨*proof*⟩

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

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 propo* **and** r :: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool*
assumes *propo-rew-step* $r\ \varphi\ \varphi'$
shows $\exists\psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* $p\ \varphi\ \psi \wedge$ *replace-at* $p\ \varphi\ \psi' = \varphi'$
⟨*proof*⟩

6.2 Consistency preservation

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

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

lemma *propo-rew-step-preservers-val-explicit*:
propo-rew-step $r\ \varphi\ \psi \Longrightarrow$ *preserves-un-sat* $r \Longrightarrow$ *propo-rew-step* $r\ \varphi\ \psi \Longrightarrow (\forall A.\ A \models\varphi \longleftrightarrow A\models\psi)$
⟨*proof*⟩

lemma *propo-rew-step-preservers-val'*:
assumes *preserves-un-sat r*
shows *preserves-un-sat* (*propo-rew-step r*)
⟨*proof*⟩

lemma *preserves-un-sat-OO*[*intro*]:
preserves-un-sat f $\Longrightarrow$ *preserves-un-sat g* $\Longrightarrow$ *preserves-un-sat* (*f OO g*)

$\langle proof \rangle$

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

lemma *star-consistency-preservation*:
$preserves\text{-}un\text{-}sat\ r \Longrightarrow\ preserves\text{-}un\text{-}sat\ (propo\text{-}rew\text{-}step\ r)\hat{}{**}$
$\langle proof \rangle$

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*]:
$preserves\text{-}un\text{-}sat\ r \Longrightarrow preserves\text{-}un\text{-}sat\ (full\ (propo\text{-}rew\text{-}step\ r))$
$\langle proof \rangle$

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

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

lemma *test-symb-imp-all-subformula-st*[*simp*]:
$test\text{-}symb\ FT \Longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ FT$
$test\text{-}symb\ FF \Longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ FF$
$test\text{-}symb\ (FVar\ \ x) \Longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ (FVar\ x)$
$\langle proof \rangle$

lemma *all-subformula-st-test-symb-true-phi*:
$all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi \Longrightarrow test\text{-}symb\ \varphi$
$\langle proof \rangle$

lemma *all-subformula-st-decomp-imp*:
$wf\text{-}conn\ c\ l \Longrightarrow (test\text{-}symb\ (conn\ c\ l) \land (\forall \varphi \in\ set\ l.\ all\text{-}subformula\text{-}st\ test\text{-}symb\ \varphi))$
$\Longrightarrow all\text{-}subformula\text{-}st\ test\text{-}symb\ (conn\ c\ l)$
$\langle proof \rangle$

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

lemma *all-subformula-st-decomp-rec*:
$\mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \ (\mathit{conn}\ c\ l) \Longrightarrow \mathit{wf\text{-}conn}\ c\ l$
$\Longrightarrow (\mathit{test\text{-}symb}\ (\mathit{conn}\ c\ l) \wedge (\forall \varphi \in \mathit{set}\ l.\ \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi))$
⟨*proof*⟩

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

lemma *helper-fact*: $c \in \mathit{binary\text{-}connectives} \longleftrightarrow (c = \mathit{COr} \vee c = \mathit{CAnd} \vee c = \mathit{CEq} \vee c = \mathit{CImp})$
⟨*proof*⟩
lemma *all-subformula-st-decomp-explicit*[*simp*]:
fixes $\varphi\ \psi\ ::\ 'v\ \mathit{propo}$
shows $\mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ (\mathit{FAnd}\ \varphi\ \psi)$
$\longleftrightarrow (\mathit{test\text{-}symb}\ (\mathit{FAnd}\ \varphi\ \psi) \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \psi)$
and $\mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ (\mathit{FOr}\ \varphi\ \psi)$
$\longleftrightarrow (\mathit{test\text{-}symb}\ (\mathit{FOr}\ \varphi\ \psi) \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \psi)$
and $\mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ (\mathit{FNot}\ \varphi)$
$\longleftrightarrow (\mathit{test\text{-}symb}\ (\mathit{FNot}\ \varphi) \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi)$
and $\mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ (\mathit{FEq}\ \varphi\ \psi)$
$\longleftrightarrow (\mathit{test\text{-}symb}\ (\mathit{FEq}\ \varphi\ \psi) \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \psi)$
and $\mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ (\mathit{FImp}\ \varphi\ \psi)$
$\longleftrightarrow (\mathit{test\text{-}symb}\ (\mathit{FImp}\ \varphi\ \psi) \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi \wedge \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \psi)$
⟨*proof*⟩

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

lemma *subformula-all-subformula-st*:
$\psi \preceq \varphi \Longrightarrow \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi \Longrightarrow \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \psi$
⟨*proof*⟩

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 $\neg$ *all-subformula-st test-symb* φ, then something can be rewritten in φ.

lemma *no-test-symb-step-exists*:
fixes $r::\ 'v\ \mathit{propo} \Rightarrow\ 'v\ \mathit{propo} \Rightarrow \mathit{bool}$ **and** $\mathit{test\text{-}symb}::\ 'v\ \mathit{propo} \Rightarrow \mathit{bool}$ **and** $x\ ::\ 'v$
and $\varphi\ ::\ 'v\ \mathit{propo}$
assumes *test-symb-false-nullary*: $\forall x.\ \mathit{test\text{-}symb}\ \mathit{FF} \wedge \mathit{test\text{-}symb}\ \mathit{FT} \wedge \mathit{test\text{-}symb}\ (\mathit{FVar}\ x)$
and $\forall \varphi'.\ \varphi' \preceq \varphi \longrightarrow (\neg \mathit{test\text{-}symb}\ \varphi') \longrightarrow\ (\exists\ \psi.\ r\ \varphi'\ \psi)$ **and**
$\neg\ \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi$
shows $(\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi')$
⟨*proof*⟩

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 $\forall \varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \varphi' \longrightarrow \mathit{all\text{-}subformula\text{-}st}\ \mathit{test\text{-}symb}\ \psi$ 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: $\forall\, c\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi' \longrightarrow$ *wf-conn* $c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* $(conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* $(conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$

7.2.1 Invariant while lifting of the rewriting relation

The condition $\varphi \preceq \Phi$ (that will by 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:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and $\varphi\ \psi\ \Phi$:: $'v$ *propo*
assumes H: $\forall\, \varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* φ'
$\longrightarrow$ *all-subformula-st test-symb* ψ
and H': $\forall\, (c$:: $'v$ *connective*$)\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi'$
$\longrightarrow$ *wf-conn* $c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* $(conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* $(conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$ **and**
propo-rew-step $r\ \varphi\ \psi$ **and**
$\varphi \preceq \Phi$ **and**
all-subformula-st test-symb φ
shows *all-subformula-st test-symb* ψ
$\langle proof \rangle$

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

The lemmas can be lifted to *propo-rew-step* $r^{\downarrow}$ instead of *propo-rew-step*

7.2.2 Invariant after all rewriting

lemma *full-propo-rew-step-inv-stay-with-inc*:
fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and $\varphi\ \psi$ :: $'v$ *propo*
assumes
H: $\forall\ \varphi\ \psi.$ *propo-rew-step* $r\ \varphi\ \psi \longrightarrow$ *all-subformula-st test-symb* φ
$\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall\, (c$:: $'v$ *connective*$)\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi'$
$\longrightarrow$ *wf-conn* $c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* $(conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* $(conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$ **and**
$\varphi \preceq \Phi$ **and**
full: *full* $($*propo-rew-step* $r)\ \varphi\ \psi$ **and**

init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

lemma *full-propo-rew-step-inv-stay'*:
fixes r:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** x :: *'v*
and φ ψ :: *'v propo*
assumes
H: ∀ φ ψ. *propo-rew-step* r φ ψ ⟶ *all-subformula-st test-symb* φ
⟶ *all-subformula-st test-symb* ψ **and**
H': ∀ (*c*:: *'v connective*) ξ φ ξ' φ'. *propo-rew-step* r φ φ' ⟶ *wf-conn* c (ξ @ φ # ξ')
⟶ *test-symb* (*conn* c (ξ @ φ # ξ')) ⟶ *test-symb* φ' ⟶ *test-symb* (*conn* c (ξ @ φ' # ξ')) **and**
full: *full* (*propo-rew-step* r) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

lemma *full-propo-rew-step-inv-stay*:
fixes r:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** x :: *'v*
and φ ψ :: *'v propo*
assumes
H: ∀ φ ψ. r φ ψ ⟶ *all-subformula-st test-symb* φ ⟶ *all-subformula-st test-symb* ψ **and**
H': ∀ (*c*:: *'v connective*) ξ φ ξ' φ'. *wf-conn* c (ξ @ φ # ξ') ⟶ *test-symb* (*conn* c (ξ @ φ # ξ'))
⟶ *test-symb* φ' ⟶ *test-symb* (*conn* c (ξ @ φ' # ξ')) **and**
full: *full* (*propo-rew-step* r) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

lemma *full-propo-rew-step-inv-stay-conn*:
fixes r:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** x :: *'v*
and φ ψ :: *'v propo*
assumes
H: ∀ φ ψ. r φ ψ ⟶ *all-subformula-st test-symb* φ ⟶ *all-subformula-st test-symb* ψ **and**
H': ∀ (*c*:: *'v connective*) l l'. *wf-conn* c l ⟶ *wf-conn* c l'
⟶ (*test-symb* (*conn* c l) ⟷ *test-symb* (*conn* c l')) **and**
full: *full* (*propo-rew-step* r) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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 equivalencies, then implication, after that the unused true/false and finally the reorganizing the or/and.

We will prove each transformation seperately.

8.1 Elimination of the equivalences

The first transformation consists in removing every equivalence symbol.

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

lemma *elim-equiv-transformation-consistent*:
$A \models$ *FEq* φ ψ $\longleftrightarrow$ $A \models$ *FAnd* (*FImp* φ ψ) (*FImp* ψ φ)
⟨*proof*⟩

lemma *elim-equiv-explicit*: *elim-equiv* φ ψ $\Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
⟨*proof*⟩

lemma *elim-equiv-consistent*: *preserves-un-sat elim-equiv*
⟨*proof*⟩

lemma *elimEquv-lifted-consistant*:
preserves-un-sat (*full* (*propo-rew-step elim-equiv*))
⟨*proof*⟩

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

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

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

lemma *no-equiv-symb-conn-characterization*[*simp*]:
fixes *c* :: *'v connective* **and** *l* :: *'v propo list*
assumes *wf*: *wf-conn c l*
shows *no-equiv-symb* (*conn c l*) $\longleftrightarrow$ $c \neq$ *CEq*
⟨*proof*⟩

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

lemma *no-equiv-eq*[*simp*]:
fixes φ ψ :: *'v propo*
shows
¬*no-equiv* (*FEq* φ ψ)
no-equiv FT
no-equiv FF
⟨*proof*⟩

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 φ ψ :: *'v propo*
shows
no-equiv (*FNot* φ) $\longleftrightarrow$ *no-equiv* φ
no-equiv (*FAnd* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)
no-equiv (*FOr* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)

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

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*: $\neg$ *no-equiv* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge elim\text{-}equiv\ \psi\ \psi'$
⟨*proof*⟩

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

lemma *no-equiv-full-propo-rew-step-elim-equiv*:
full (*propo-rew-step elim-equiv*) $\varphi\ \psi \Longrightarrow$ *no-equiv* ψ
⟨*proof*⟩

8.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* :: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **where**
[*simp*]: *elim-imp* $(FImp\ \varphi\ \psi)\ (FOr\ (FNot\ \varphi)\ \psi)$

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

lemma *elim-imp-explicit*: *elim-imp* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
⟨*proof*⟩

lemma *elim-imp-consistent*: *preserves-un-sat elim-imp*
⟨*proof*⟩

lemma *elim-imp-lifted-consistant*:
preserves-un-sat (*full* (*propo-rew-step elim-imp*))
⟨*proof*⟩

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

lemma *no-imp-symb-conn-characterization*:
wf-conn c l $\Longrightarrow$ *no-imp-symb* (*conn c l*) $\longleftrightarrow c \neq CImp$
⟨*proof*⟩

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

lemma *no-imp-Imp*[*simp*]:
$\neg$*no-imp* (*FImp* $\varphi\ \psi$)
no-imp FT
no-imp FF
⟨*proof*⟩

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

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:
$elim\text{-}imp\ \varphi\ \psi \Longrightarrow no\text{-}equiv\ \varphi \Longrightarrow\ no\text{-}equiv\ \psi$
⟨*proof*⟩

lemma *elim-imp-inv*:
fixes $\varphi\ \psi :: 'v\ propo$
assumes $full\ (propo\text{-}rew\text{-}step\ elim\text{-}imp)\ \varphi\ \psi$ **and** $no\text{-}equiv\ \varphi$
shows $no\text{-}equiv\ \psi$
⟨*proof*⟩

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

lemma *no-imp-full-propo-rew-step-elim-imp*: $full\ (propo\text{-}rew\text{-}step\ elim\text{-}imp)\ \varphi\ \psi \Longrightarrow no\text{-}imp\ \psi$
⟨*proof*⟩

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: $elimTB\ (FAnd\ \varphi\ FT)\ \varphi$ |
ElimTB1': $elimTB\ (FAnd\ FT\ \varphi)\ \varphi$ |

ElimTB2: $elimTB\ (FAnd\ \varphi\ FF)\ FF$ |
ElimTB2': $elimTB\ (FAnd\ FF\ \varphi)\ FF$ |

ElimTB3: $elimTB\ (FOr\ \varphi\ FT)\ FT$ |
ElimTB3': $elimTB\ (FOr\ FT\ \varphi)\ FT$ |

ElimTB4: $elimTB\ (FOr\ \varphi\ FF)\ \varphi$ |
ElimTB4': $elimTB\ (FOr\ FF\ \varphi)\ \varphi$ |

ElimTB5: $elimTB\ (FNot\ FT)\ FF$ |
ElimTB6: $elimTB\ (FNot\ FF)\ FT$

lemma *elimTB-consistent*: *preserves-un-sat elimTB*
⟨*proof*⟩

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

lemma *wf-conn-no-T-F-symb-iff*[*simp*]:
  wf-conn c $\psi s \Longrightarrow$
    no-T-F-symb (*conn c* ψs) $\longleftrightarrow$ ($c \neq CF \wedge c \neq CT \wedge (\forall \psi \in set\ \psi s.\ \psi \neq FF \wedge \psi \neq FT)$)
  ⟨*proof*⟩

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

lemma *no-T-F-symb-false*[*simp*]:
  fixes *c* :: *'v connective*
  shows
    ¬*no-T-F-symb* (*FT* :: *'v propo*)
    ¬*no-T-F-symb* (*FF* :: *'v propo*)
    ⟨*proof*⟩

lemma *no-T-F-symb-bool*[*simp*]:
  fixes *x* :: *'v*
  shows *no-T-F-symb* (*FVar x*)
  ⟨*proof*⟩

lemma *no-T-F-symb-fnot-imp*:
  ¬*no-T-F-symb* (*FNot* φ) $\Longrightarrow \varphi = FT \vee \varphi = FF$
⟨*proof*⟩

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

Actually it is not possible to remover 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* $\varphi \Longrightarrow$ *no-T-F-symb-except-toplevel* φ

lemma *no-T-F-symb-except-toplevel-bool*:
  fixes *x* :: *'v*
  shows *no-T-F-symb-except-toplevel* (*FVar x*)
  ⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-not-decom*:
  $\varphi \neq FT \Longrightarrow \varphi \neq FF \Longrightarrow$ *no-T-F-symb-except-toplevel* (*FNot* φ)
  ⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-bin-decom*:
  fixes φ ψ :: *'v propo*
  assumes $\varphi \neq FT$ **and** $\varphi \neq FF$ **and** $\psi \neq FT$ **and** $\psi \neq FF$

and c: $c \in$ *binary-connectives*
shows *no-T-F-symb-except-toplevel* (*conn c* [φ, ψ])
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-if-is-a-true-false*:
fixes l :: $'v$ *propo list* **and** c :: $'v$ *connective*
assumes *corr*: *wf-conn c l*
and $FT \in set\ l \lor FF \in set\ l$
shows $\neg$*no-T-F-symb-except-toplevel* (*conn c l*)
⟨*proof*⟩

lemma *no-T-F-symb-except-top-level-false-example*[*simp*]:
fixes φ ψ :: $'v$ *propo*
assumes $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FAnd* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FOr* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FImp* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FEq* φ ψ)
⟨*proof*⟩

lemma *no-T-F-symb-except-top-level-false-not*[*simp*]:
fixes φ ψ :: $'v$ *propo*
assumes $\varphi = FT \lor \varphi = FF$
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FNot* φ)
⟨*proof*⟩

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

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

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

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

lemma *no-T-F-except-top-level-false*:
fixes l :: $'v$ *propo list* **and** c :: $'v$ *connective*
assumes *wf-conn c l*
and $FT \in set\ l \lor FF \in set\ l$
shows $\neg$*no-T-F-except-top-level* (*conn c l*)
⟨*proof*⟩

lemma *no-T-F-except-top-level-false-example*[*simp*]:
fixes φ ψ :: $'v$ *propo*
assumes $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
$\neg$*no-T-F-except-top-level* (*FAnd* φ ψ)
$\neg$*no-T-F-except-top-level* (*FOr* φ ψ)
$\neg$*no-T-F-except-top-level* (*FEq* φ ψ)
$\neg$*no-T-F-except-top-level* (*FImp* φ ψ)
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-no-T-F-symb*:
  $no\text{-}T\text{-}F\text{-}symb\text{-}except\text{-}toplevel\ \varphi \Longrightarrow \varphi \neq FF \Longrightarrow \varphi \neq FT \Longrightarrow no\text{-}T\text{-}F\text{-}symb\ \varphi$
  ⟨*proof*⟩

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

lemma *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*:
  $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\ \varphi \Longrightarrow \varphi \neq FF \Longrightarrow \varphi \neq FT \Longrightarrow no\text{-}T\text{-}F\ \varphi$
  ⟨*proof*⟩

lemma *no-T-F-no-T-F-except-top-level*:
  $no\text{-}T\text{-}F\ \varphi \Longrightarrow no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\ \varphi$
  ⟨*proof*⟩

lemma *no-T-F-except-top-level-simp*[*simp*]: *no-T-F-except-top-level FF no-T-F-except-top-level FT*
  ⟨*proof*⟩

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

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

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

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

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

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

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

$\langle proof \rangle$

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

lemma *no-T-F-except-top-level-rew*:
fixes φ :: *'v propo*
assumes *noTB*: $\neg$ *no-T-F-except-top-level* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elimTB* ψ ψ'
$\langle proof \rangle$

lemma *elimTB-inv*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elimTB*) φ ψ
and *no-equiv* φ **and** *no-imp* φ
shows *no-equiv* ψ **and** *no-imp* ψ
$\langle proof \rangle$

lemma *elimTB-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes *no-equiv* φ **and** *no-imp* φ **and** *full* (*propo-rew-step elimTB*) φ ψ
shows *no-T-F-except-top-level* ψ
$\langle proof \rangle$

8.4 PushNeg

Push the negation inside the formula, until the litteral.

inductive *pushNeg* **where**
PushNeg1[*simp*]: *pushNeg* (*FNot* (*FAnd* φ ψ)) (*FOr* (*FNot* φ) (*FNot* ψ)) |
PushNeg2[*simp*]: *pushNeg* (*FNot* (*FOr* φ ψ)) (*FAnd* (*FNot* φ) (*FNot* ψ)) |
PushNeg3[*simp*]: *pushNeg* (*FNot* (*FNot* φ)) φ

lemma *pushNeg-transformation-consistent*:
$A \models$ *FNot* (*FAnd* φ ψ) $\longleftrightarrow A \models$ (*FOr* (*FNot* φ) (*FNot* ψ))
$A \models$ *FNot* (*FOr* φ ψ) $\longleftrightarrow A \models$ (*FAnd* (*FNot* φ) (*FNot* ψ))
$A \models$ *FNot* (*FNot* φ) $\longleftrightarrow A \models \varphi$
$\langle proof \rangle$

lemma *pushNeg-explicit*: *pushNeg* φ $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
$\langle proof \rangle$

lemma *pushNeg-consistent*: *preserves-un-sat pushNeg*
$\langle proof \rangle$

lemma *pushNeg-lifted-consistant*:
preserves-un-sat (*full* (*propo-rew-step pushNeg*))
$\langle proof \rangle$

fun *simple* **where**

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

lemma *simple-decomp*:
$simple\ \varphi \longleftrightarrow (\varphi = FT \lor \varphi = FF \lor (\exists x.\ \varphi = FVar\ x))$
⟨*proof*⟩

lemma *subformula-conn-decomp-simple*:
fixes $\varphi\ \psi$ :: $'v$ *propo*
assumes *s*: *simple* ψ
shows $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \lor \varphi = \psi)$
⟨*proof*⟩

lemma *subformula-conn-decomp-explicit*[*simp*]:
fixes φ :: $'v$ *propo* **and** x :: $'v$
shows
$\varphi \preceq FNot\ FT \longleftrightarrow (\varphi = FNot\ FT \lor \varphi = FT)$
$\varphi \preceq FNot\ FF \longleftrightarrow (\varphi = FNot\ FF \lor \varphi = FF)$
$\varphi \preceq FNot\ (FVar\ x) \longleftrightarrow (\varphi = FNot\ (FVar\ x) \lor \varphi = FVar\ x)$
⟨*proof*⟩

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

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

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

lemma *simple-not-step-exists*:
fixes $\varphi\ \psi$ :: $'v$ *propo*
assumes *no-equiv* φ **and** *no-imp* φ
shows $\psi \preceq \varphi \Longrightarrow \neg\ simple\text{-}not\text{-}symb\ \psi \Longrightarrow \exists \psi'.\ pushNeg\ \psi\ \psi'$
⟨*proof*⟩

lemma *simple-not-rew*:
fixes φ :: $'v$ *propo*
assumes *noTB*: $\neg$ *simple-not* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \land pushNeg\ \psi\ \psi'$
⟨*proof*⟩

lemma *no-T-F-except-top-level-pushNeg1*:
no-T-F-except-top-level (*FNot* (*FAnd* $\varphi\ \psi$)) $\Longrightarrow$ *no-T-F-except-top-level* (*FOr* (*FNot* φ) (*FNot* ψ))
⟨*proof*⟩

lemma *no-T-F-except-top-level-pushNeg2*:
no-T-F-except-top-level (*FNot* (*FOr* $\varphi\ \psi$)) $\Longrightarrow$ *no-T-F-except-top-level* (*FAnd* (*FNot* φ) (*FNot* ψ))

$\langle proof \rangle$

lemma *no-T-F-symb-pushNeg*:
no-T-F-symb (*FOr* (*FNot* φ') (*FNot* ψ'))
no-T-F-symb (*FAnd* (*FNot* φ') (*FNot* ψ'))
no-T-F-symb (*FNot* (*FNot* φ'))
$\langle proof \rangle$

lemma *propo-rew-step-pushNeg-no-T-F-symb*:
propo-rew-step pushNeg φ ψ $\Longrightarrow$ *no-T-F-except-top-level* φ $\Longrightarrow$ *no-T-F-symb* φ $\Longrightarrow$ *no-T-F-symb* ψ
$\langle proof \rangle$

lemma *propo-rew-step-pushNeg-no-T-F*:
propo-rew-step pushNeg φ ψ $\Longrightarrow$ *no-T-F* φ $\Longrightarrow$ *no-T-F* ψ
$\langle proof \rangle$

lemma *pushNeg-inv*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step pushNeg*) φ ψ
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
$\langle proof \rangle$

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

8.5 Push inside

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

lemma *push-conn-inside-explicit*: *push-conn-inside* c c' φ ψ $\Longrightarrow$ $\forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
$\langle proof \rangle$

lemma *push-conn-inside-consistent*: *preserves-un-sat* (*push-conn-inside* c c')
$\langle proof \rangle$

lemma *propo-rew-step-push-conn-inside*[*simp*]:
$\neg$*propo-rew-step* (*push-conn-inside* c c') *FT* ψ $\neg$*propo-rew-step* (*push-conn-inside* c c') *FF* ψ
$\langle proof \rangle$

inductive *not-c-in-c′-symb*:: *′v connective* ⇒ *′v connective* ⇒ *′v propo* ⇒ *bool* **for** *c c′* **where**
not-c-in-c′-symb-l[*simp*]: *wf-conn c* [*conn c′* [φ, φ'], ψ] $\Longrightarrow$ *wf-conn c′* [φ, φ']
$\Longrightarrow$ *not-c-in-c′-symb c c′* (*conn c* [*conn c′* [φ, φ'], ψ]) |
not-c-in-c′-symb-r[*simp*]: *wf-conn c* [ψ, *conn c′* [φ, φ']] $\Longrightarrow$ *wf-conn c′* [φ, φ']
$\Longrightarrow$ *not-c-in-c′-symb c c′* (*conn c* [ψ, *conn c′* [φ, φ']])

abbreviation *c-in-c′-symb c c′* $\varphi \equiv \neg$*not-c-in-c′-symb c c′* φ

lemma *c-in-c′-symb-simp*:
not-c-in-c′-symb c c′ $\xi \Longrightarrow \xi$ = *FF* ∨ ξ = *FT* ∨ ξ = *FVar x* ∨ ξ = *FNot FF* ∨ ξ = *FNot FT*
∨ ξ = *FNot* (*FVar x*)$\Longrightarrow$ *False*
⟨*proof*⟩

lemma *c-in-c′-symb-simp′*[*simp*]:
¬*not-c-in-c′-symb c c′ FF*
¬*not-c-in-c′-symb c c′ FT*
¬*not-c-in-c′-symb c c′* (*FVar x*)
¬*not-c-in-c′-symb c c′* (*FNot FF*)
¬*not-c-in-c′-symb c c′* (*FNot FT*)
¬*not-c-in-c′-symb c c′* (*FNot* (*FVar x*))
⟨*proof*⟩

definition *c-in-c′-only* **where**
c-in-c′-only c c′ ≡ *all-subformula-st* (*c-in-c′-symb c c′*)

lemma *c-in-c′-only-simp*[*simp*]:
c-in-c′-only c c′ FF
c-in-c′-only c c′ FT
c-in-c′-only c c′ (*FVar x*)
c-in-c′-only c c′ (*FNot FF*)
c-in-c′-only c c′ (*FNot FT*)
c-in-c′-only c c′ (*FNot* (*FVar x*))
⟨*proof*⟩

lemma *not-c-in-c′-symb-commute*:
not-c-in-c′-symb c c′ $\xi \Longrightarrow$ *wf-conn c* [φ, ψ] $\Longrightarrow \xi$ = *conn c* [φ, ψ]
$\Longrightarrow$ *not-c-in-c′-symb c c′* (*conn c* [ψ, φ])
⟨*proof*⟩

lemma *not-c-in-c′-symb-commute′*:
wf-conn c [φ, ψ] $\Longrightarrow$ *c-in-c′-symb c c′* (*conn c* [φ, ψ]) $\longleftrightarrow$ *c-in-c′-symb c c′* (*conn c* [ψ, φ])
⟨*proof*⟩

lemma *not-c-in-c′-comm*:
assumes *wf*: *wf-conn c* [φ, ψ]
shows *c-in-c′-only c c′* (*conn c* [φ, ψ]) $\longleftrightarrow$ *c-in-c′-only c c′* (*conn c* [ψ, φ]) (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

lemma *not-c-in-c′-simp*[*simp*]:
fixes $\varphi 1$ $\varphi 2$ ψ :: *′v propo* **and** *x* :: *′v*
shows

$c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ FT$
$c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ FF$
$c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ (FVar\ x)$
$wf\text{-}conn\ c\ [conn\ c'\ [\varphi 1,\ \varphi 2],\ \psi] \Longrightarrow wf\text{-}conn\ c'\ [\varphi 1,\ \varphi 2]$
$\Longrightarrow \neg\ c\text{-}in\text{-}c'\text{-}only\ c\ c'\ (conn\ c\ [conn\ c'\ [\varphi 1,\ \varphi 2],\ \psi])$
⟨*proof*⟩

lemma *c-in-c′-symb-not*[*simp*]:
fixes $c\ c' :: 'v\ connective$ **and** $\psi :: 'v\ propo$
shows $c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ (FNot\ \psi)$
⟨*proof*⟩

lemma *c-in-c′-symb-step-exists*:
fixes $\varphi :: 'v\ propo$
assumes c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
shows $\psi \preceq \varphi \Longrightarrow \neg\ c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ \psi \Longrightarrow \exists \psi'.\ push\text{-}conn\text{-}inside\ c\ c'\ \psi\ \psi'$
⟨*proof*⟩

lemma *c-in-c′-symb-rew*:
fixes $\varphi :: 'v\ propo$
assumes *noTB*: $\neg c\text{-}in\text{-}c'\text{-}only\ c\ c'\ \varphi$
and c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \land push\text{-}conn\text{-}inside\ c\ c'\ \psi\ \psi'$
⟨*proof*⟩

lemma *push-conn-insidec-in-c′-symb-no-T-F*:
fixes $\varphi\ \psi :: 'v\ propo$
shows $propo\text{-}rew\text{-}step\ (push\text{-}conn\text{-}inside\ c\ c')\ \varphi\ \psi \Longrightarrow no\text{-}T\text{-}F\ \varphi \Longrightarrow no\text{-}T\text{-}F\ \psi$
⟨*proof*⟩

lemma *simple-propo-rew-step-push-conn-inside-inv*:
$propo\text{-}rew\text{-}step\ (push\text{-}conn\text{-}inside\ c\ c')\ \varphi\ \psi \Longrightarrow simple\ \varphi \Longrightarrow simple\ \psi$
⟨*proof*⟩

lemma *simple-propo-rew-step-inv-push-conn-inside-simple-not*:
fixes $c\ c' :: 'v\ connective$ **and** $\varphi\ \psi :: 'v\ propo$
shows $propo\text{-}rew\text{-}step\ (push\text{-}conn\text{-}inside\ c\ c')\ \varphi\ \psi \Longrightarrow simple\text{-}not\ \varphi \Longrightarrow simple\text{-}not\ \psi$
⟨*proof*⟩

lemma *propo-rew-step-push-conn-inside-simple-not*:
fixes $\varphi\ \varphi' :: 'v\ propo$ **and** $\xi\ \xi' :: 'v\ propo\ list$ **and** $c :: 'v\ connective$
assumes
$propo\text{-}rew\text{-}step\ (push\text{-}conn\text{-}inside\ c\ c')\ \varphi\ \varphi'$ **and**
$wf\text{-}conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')$ **and**
$simple\text{-}not\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi'))$ **and**
$simple\text{-}not\text{-}symb\ \varphi'$
shows $simple\text{-}not\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$
⟨*proof*⟩

lemma *push-conn-inside-not-true-false*:
$push\text{-}conn\text{-}inside\ c\ c'\ \varphi\ \psi \Longrightarrow \psi \neq FT \land \psi \neq FF$
⟨*proof*⟩

lemma *push-conn-inside-inv*:
  fixes φ ψ :: $'v$ *propo*
  assumes *full* (*propo-rew-step* (*push-conn-inside* c c')) φ ψ
  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 φ ψ :: $'v$ *propo*
  assumes
    no-equiv φ **and**
    no-imp φ **and**
    full (*propo-rew-step* (*push-conn-inside* c c')) φ ψ **and**
    no-T-F-except-top-level φ **and**
    simple-not φ **and**
    $c = CAnd \lor c = COr$ **and**
    $c' = CAnd \lor c' = COr$
  shows *c-in-c'-only* c c' ψ
  ⟨*proof*⟩

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

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

lemma *only-c-inside-symb-simp*[*simp*]:
  only-c-inside-symb c *FF* *only-c-inside-symb* c *FT* *only-c-inside-symb* c (*FVar* x) ⟨*proof*⟩

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

lemma *only-c-inside-symb-decomp*:
  only-c-inside-symb c ψ $\longleftrightarrow$ (*simple* ψ
    $\lor$ ($\exists$ φ'. ψ = *FNot* φ' $\land$ *simple* φ')
    $\lor$ ($\exists$ l. ψ = *conn* c l $\land$ *wf-conn* c l))
  ⟨*proof*⟩

lemma *only-c-inside-symb-decomp-not*[*simp*]:
  fixes c :: $'v$ *connective*
  assumes c: $c \neq CNot$
  shows *only-c-inside-symb* c (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
  ⟨*proof*⟩

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

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

$(\forall\,\psi.\ \psi \preceq \varphi \longrightarrow (simple\ \psi \lor (\exists\ \varphi'.\ \psi = FNot\ \varphi' \land simple\ \varphi')$
$\lor (\exists\, l.\ \psi = conn\ c\ l \land wf\text{-}conn\ c\ l)))$
⟨*proof*⟩

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

lemma *only-c-inside-implies-c-in-c′-symb*:
assumes δ: $c \neq c'$ **and** c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
shows $only\text{-}c\text{-}inside\ c\ \varphi \Longrightarrow c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ \varphi$
⟨*proof*⟩

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

lemma *only-c-inside-implies-c-in-c′-only*:
assumes δ: $c \neq c'$ **and** c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
shows $only\text{-}c\text{-}inside\ c\ \varphi \Longrightarrow c\text{-}in\text{-}c'\text{-}only\ c\ c'\ \varphi$
⟨*proof*⟩

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

8.5.2 Push Conjunction

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

lemma *pushConj-consistent*: $preserves\text{-}un\text{-}sat\ pushConj$
⟨*proof*⟩

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

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

lemma *pushConj-inv*:
fixes $\varphi\ \psi :: 'v\ propo$
assumes $full\ (propo\text{-}rew\text{-}step\ pushConj)\ \varphi\ \psi$
and $no\text{-}equiv\ \varphi$ **and** $no\text{-}imp\ \varphi$ **and** $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\ \varphi$ **and** $simple\text{-}not\ \varphi$
shows $no\text{-}equiv\ \psi$ **and** $no\text{-}imp\ \psi$ **and** $no\text{-}T\text{-}F\text{-}except\text{-}top\text{-}level\ \psi$ **and** $simple\text{-}not\ \psi$
⟨*proof*⟩

lemma *pushConj-full-propo-rew-step*:
fixes $\varphi\ \psi :: 'v\ propo$

assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step pushConj*) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ
shows *and-in-or-only* ψ
⟨*proof*⟩

8.5.3 Push Disjunction

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

lemma *pushDisj-consistent*: *preserves-un-sat pushDisj*
⟨*proof*⟩

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*]:
~*or-in-and-only* (*FOr* (*FAnd* $\psi 1$ $\psi 2$) φ')
⟨*proof*⟩

lemma *pushDisj-inv*:
fixes φ ψ :: *′v propo*
assumes *full* (*propo-rew-step pushDisj*) φ ψ
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 *pushDisj-full-propo-rew-step*:
fixes φ ψ :: *′v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step pushDisj*) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ
shows *or-in-and-only* ψ
⟨*proof*⟩

9 The full transformations

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

9.1.1 Definition

The normal is a super group of groups

inductive *grouped-by* :: *′a connective* ⇒ *′a propo* ⇒ *bool* **for** *c* **where**
simple-is-grouped[*simp*]: *simple* φ ⟹ *grouped-by c* φ |
simple-not-is-grouped[*simp*]: *simple* φ ⟹ *grouped-by c* (*FNot* φ) |

connected-is-group[*simp*]: *grouped-by c* $\varphi \Longrightarrow$ *grouped-by c* $\psi \Longrightarrow$ *wf-conn c* [φ, ψ]
$\Longrightarrow$ *grouped-by c* (*conn c* [φ, ψ])

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

lemma *only-c-inside-symb-c-eq-c'*:
only-c-inside-symb c (*conn c'* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \vee c' = COr$ $\Longrightarrow$ *wf-conn c'* [$\varphi 1$, $\varphi 2$]
$\Longrightarrow c' = c$
⟨*proof*⟩

lemma *only-c-inside-c-eq-c'*:
only-c-inside c (*conn c'* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \vee c' = COr$ $\Longrightarrow$ *wf-conn c'* [$\varphi 1$, $\varphi 2$] $\Longrightarrow c = c'$
⟨*proof*⟩

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

lemma *grouped-by-false*:
grouped-by c (*conn c'* [φ, ψ]) $\Longrightarrow c \neq c' \Longrightarrow$ *wf-conn c'* [φ, ψ] $\Longrightarrow$ *False*
⟨*proof*⟩

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* $\varphi \Longrightarrow$ *super-grouped-by c c'* φ |
connected-is-super-group: *super-grouped-by c c'* $\varphi \Longrightarrow$ *super-grouped-by c c'* $\psi \Longrightarrow$ *wf-conn c* [φ, ψ]
$\Longrightarrow$ *super-grouped-by c c'* (*conn c'* [φ, ψ])

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*))
⟨*proof*⟩

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* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *c-in-c'-only c c'* φ
$\Longrightarrow$ *super-grouped-by c c'* φ
(**is** *?NE* $\varphi \Longrightarrow$ *?NI* $\varphi \Longrightarrow$ *?SN* $\varphi \Longrightarrow$ *?C* $\varphi \Longrightarrow$ *?S* φ)
⟨*proof*⟩

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* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *or-in-and-only* $\varphi \Longrightarrow$ *is-conj-with-TF* φ
⟨*proof*⟩

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

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

lemma *cnf-rew-consistent*: *preserves-un-sat cnf-rew*
⟨*proof*⟩

lemma *cnf-rew-is-cnf*: *cnf-rew* $\varphi\ \varphi' \Longrightarrow$ *is-cnf* φ'
⟨*proof*⟩

9.3 Disjunctive Normal Form

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

lemma *and-in-or-only-conjunction-in-disj*:
shows *no-equiv* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *and-in-or-only* $\varphi \Longrightarrow$ *is-disj-with-TF* φ
⟨*proof*⟩

definition *is-dnf* :: *'a propo* ⇒ *bool* **where**
is-dnf $\varphi \longleftrightarrow$ *is-disj-with-TF* $\varphi \wedge$ *no-T-F-except-top-level* φ

9.3.1 Full DNF transform

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

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

lemma *dnf-rew-consistent*: *preserves-un-sat dnf-rew*
⟨*proof*⟩

theorem *dnf-transformation-correction*:
dnf-rew $\varphi\ \varphi' \Longrightarrow$ *is-dnf* φ'
⟨*proof*⟩

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 $[\![$*no-equiv* $?\varphi$; *no-imp* $?\varphi$; $?\psi \preceq ?\varphi$; $\neg$ *no-T-F-symb-except-toplevel* $?\psi]\!] \Longrightarrow \exists \psi'$. *elimTB* $?\psi$ ψ', 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 φ :: *'v propo*
  shows $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'$. *elimTBFull* ψ ψ'
⟨*proof*⟩

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

lemma *no-T-F-except-top-level-rew'*:
  fixes φ :: *'v propo*
  assumes *noTB*: $\neg$ *no-T-F-except-top-level* φ
  shows $\exists \psi\ \psi'$. $\psi \preceq \varphi \land$ *elimTBFull* ψ ψ'
⟨*proof*⟩

lemma *elimTBFull-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elimTBFull*) φ ψ
shows *no-T-F-except-top-level* ψ
⟨*proof*⟩

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* φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
⟨*proof*⟩

lemma *elim-equiv-inv'*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elim-equiv*) φ ψ **and** *no-T-F-except-top-level* φ
shows *no-T-F-except-top-level* ψ
⟨*proof*⟩

lemma *propo-rew-step-ElimImp-no-T-F*: *propo-rew-step elim-imp* φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
⟨*proof*⟩

lemma *elim-imp-inv'*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elim-imp*) φ ψ **and** *no-T-F-except-top-level* φ
shows*no-T-F-except-top-level* ψ
⟨*proof*⟩

10.3 The new CNF and DNF transformation

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

definition *dnf-rew'* :: *'a propo* ⇒ *'a propo* ⇒ *bool* **where**
dnf-rew' =
(*full* (*propo-rew-step elimTBFull*)) *OO*
(*full* (*propo-rew-step elim-equiv*)) *OO*
(*full* (*propo-rew-step elim-imp*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushConj*))

lemma *dnf-rew'-consistent*: *preserves-un-sat dnf-rew'*
⟨*proof*⟩

theorem *cnf-transformation-correction*:
dnf-rew' φ $\varphi' \Longrightarrow$ *is-dnf* φ'
⟨*proof*⟩

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

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

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

lemma *cnf-rew′-consistent*: *preserves-un-sat cnf-rew′*
⟨*proof*⟩

theorem *cnf′-transformation-correction*:
cnf-rew′ φ φ' $\Longrightarrow$ *is-cnf* φ'
⟨*proof*⟩

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* ⇒ *′a literal* ⇒ *bool* (**infix** ⊨*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* ⇒ *bool* **where**
consistent-interp $I = (\forall L.\ \neg(L \in I \wedge -\ L \in I))$

lemma *consistent-interp-empty*[*simp*]:
consistent-interp {} ⟨*proof*⟩

lemma *consistent-interp-single*[*simp*]:
consistent-interp $\{L\}$ ⟨*proof*⟩

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

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

lemma *consistent-interp-change-insert*:
$a \notin A \Longrightarrow -a \notin A \Longrightarrow$ *consistent-interp* (*insert* $(-a)$ A) $\longleftrightarrow$ *consistent-interp* (*insert a A*)
⟨*proof*⟩

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

lemma *consistent-interp-insert-not-in*:
consistent-interp A $\Longrightarrow a \notin A \Longrightarrow -a \notin A \Longrightarrow$ *consistent-interp* (*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* ⇒ *'a set* **where**
atms-of-ms $\psi s = \bigcup$(*atms-of* ‘ ψs)

lemma *atms-of-mmltiset*[*simp*]:
atms-of (*mset a*) = *atm-of* ‘ *set a*
⟨*proof*⟩

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

definition *atms-of-s* :: *'a literal set* ⇒ *'a set* **where**
atms-of-s C = *atm-of* ‘ *C*

lemma *atms-of-ms-emtpy-set*[*simp*]:
atms-of-ms {} = {}
⟨*proof*⟩

lemma *atms-of-ms-memtpy*[*simp*]:
atms-of-ms {{#}} = {}
⟨*proof*⟩

lemma *atms-of-ms-mono*:
$A \subseteq B \Longrightarrow$ *atms-of-ms A* $\subseteq$ *atms-of-ms B*
⟨*proof*⟩

lemma *atms-of-ms-finite*[*simp*]:
finite $\psi s \Longrightarrow$ *finite* (*atms-of-ms* ψs)
⟨*proof*⟩

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

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

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

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

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

lemma *atms-of-ms-single-set-mset-atns-of*[*simp*]:
$atms\text{-}of\text{-}ms\ (single\ `\ set\text{-}mset\ B) = atms\text{-}of\ B$
⟨proof⟩

lemma *atms-of-ms-remove-incl*:
shows $atms\text{-}of\text{-}ms\ (Set.remove\ a\ \psi) \subseteq atms\text{-}of\text{-}ms\ \psi$
⟨proof⟩

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

lemma *in-atms-of-s-decomp*[*iff*]:
$P \in$ *atms-of-s* $I \longleftrightarrow$ (*Pos* $P \in I \lor$ *Neg* $P \in I$) (**is** *?P* $\longleftrightarrow$ *?Q*)
$\langle proof \rangle$

lemma *atm-of-in-atm-of-set-in-uminus*:
atm-of $L' \in$ *atm-of* ' $B \Longrightarrow L' \in B \lor - L' \in B$
$\langle proof \rangle$

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* $l \in I \lor$ *Neg* $l \in I$)

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

lemma *total-over-set-empty*[*simp*]:
total-over-set I $\{\}$
$\langle proof \rangle$

lemma *total-over-m-empty*[*simp*]:
total-over-m I $\{\}$
$\langle proof \rangle$

lemma *total-over-set-single*[*iff*]:
total-over-set I $\{L\} \longleftrightarrow$ (*Pos* $L \in I \lor$ *Neg* $L \in I$)
$\langle proof \rangle$

lemma *total-over-set-insert*[*iff*]:
total-over-set I (*insert* L *Ls*) $\longleftrightarrow$ ((*Pos* $L \in I \lor$ *Neg* $L \in I$) $\land$ *total-over-set* I *Ls*)
$\langle proof \rangle$

lemma *total-over-set-union*[*iff*]:
total-over-set I (*Ls* $\cup$ *Ls'*) $\longleftrightarrow$ (*total-over-set* I *Ls* $\land$ *total-over-set* I *Ls'*)
$\langle proof \rangle$

lemma *total-over-m-subset*:
$A \subseteq B \Longrightarrow$ *total-over-m* I $B \Longrightarrow$ *total-over-m* I A
$\langle proof \rangle$

lemma *total-over-m-sum*[*iff*]:
shows *total-over-m* I $\{C + D\} \longleftrightarrow$ (*total-over-m* I $\{C\} \land$ *total-over-m* I $\{D\}$)
$\langle proof \rangle$

lemma *total-over-m-union*[*iff*]:
total-over-m I ($A \cup B$) $\longleftrightarrow$ (*total-over-m* I $A \land$ *total-over-m* I B)
$\langle proof \rangle$

lemma *total-over-m-insert*[*iff*]:
total-over-m I (*insert* a A) $\longleftrightarrow$ (*total-over-set* I (*atms-of* a) $\land$ *total-over-m* I A)
$\langle proof \rangle$

lemma *total-over-m-extension*:
fixes I :: *'v literal set* **and** A :: *'v clauses*

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

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 $\exists I'$. *total-over-m* $(I \cup I')$ $(A \cup B)$
$\wedge$ $(\forall x \in I'$. *atm-of* $x \in$ *atms-of-ms* B $\wedge$ *atm-of* $x \notin$ *atms-of-ms* $A)$ $\wedge$ *consistent-interp* $(I \cup I')$
$\langle proof \rangle$

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

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

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 $\{\psi\}$
$\langle proof \rangle$

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

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

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*]: $\neg$ $I \models \{\#\}$
$\langle proof \rangle$

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

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

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

$\langle proof \rangle$

lemma *true-cls-mono-leD*[*dest*]: $A \subseteq\# B \Longrightarrow I \models A \Longrightarrow I \models B$
$\langle proof \rangle$

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*]: $I \models \mathit{replicate\text{-}mset}\ n\ L \longleftrightarrow n \neq 0 \wedge I \models l\ L$
$\langle proof \rangle$

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*]:
$M \models A \Longrightarrow \mathit{insert}\ L\ M \models A$
$\langle proof \rangle$

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*]: $I \models s\ \mathit{insert}\ C\ DD \longleftrightarrow I \models C \wedge I \models s\ DD$
$\langle proof \rangle$

lemma *true-clss-mono*: $DD \subseteq CC \Longrightarrow I \models s\ CC \Longrightarrow I \models s\ DD$
$\langle proof \rangle$

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

assumes $I \models s\ \psi$
shows $I \cup I' \models s\ \psi$
⟨*proof*⟩

lemma *true-clss-union-increase'*[*simp*]:
assumes $I' \models s\ \psi$
shows $I \cup I' \models s\ \psi$
⟨*proof*⟩

lemma *true-clss-commute-l*:
$(I \cup I' \models s\ \psi) \longleftrightarrow (I' \cup I \models s\ \psi)$
⟨*proof*⟩

lemma *model-remove*[*simp*]: $I \models s\ N \Longrightarrow I \models s\ Set.remove\ a\ N$
⟨*proof*⟩

lemma *model-remove-minus*[*simp*]: $I \models s\ N \Longrightarrow I \models s\ N - A$
⟨*proof*⟩

lemma *notin-vars-union-true-cls-true-cls*:
assumes $\forall x{\in}I'.\ atm\text{-}of\ x \notin atms\text{-}of\text{-}ms\ A$
and $atms\text{-}of\ L \subseteq atms\text{-}of\text{-}ms\ A$
and $I \cup I' \models L$
shows $I \models L$
⟨*proof*⟩

lemma *notin-vars-union-true-clss-true-clss*:
assumes $\forall x{\in}I'.\ atm\text{-}of\ x \notin atms\text{-}of\text{-}ms\ A$
and $atms\text{-}of\text{-}ms\ L \subseteq atms\text{-}of\text{-}ms\ A$
and $I \cup I' \models s\ L$
shows $I \models s\ L$
⟨*proof*⟩

11.2.5 Satisfiability

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

lemma *satisfiable-single*[*simp*]:
$satisfiable\ \{\{\#L\#\}\}$
⟨*proof*⟩

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

lemma *satisfiable-decreasing*:
assumes $satisfiable\ (\psi \cup \psi')$
shows $satisfiable\ \psi$
⟨*proof*⟩

lemma *satisfiable-def-min*:
$satisfiable\ CC$
$\longleftrightarrow (\exists I.\ I \models s\ CC \land consistent\text{-}interp\ I \land total\text{-}over\text{-}m\ I\ CC \land atm\text{-}of`I = atms\text{-}of\text{-}ms\ CC)$
(**is** $?sat \longleftrightarrow ?B$)
⟨*proof*⟩

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*]: $I \models m\ CC + DD \longleftrightarrow I \models m\ CC \land I \models m\ DD$
$\langle proof \rangle$

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

lemma *true-cls-mset-mono*: *set-mset DD* $\subseteq$ *set-mset CC* $\Longrightarrow I \models m\ CC \Longrightarrow I \models m\ DD$
$\langle proof \rangle$

lemma *true-clss-set-mset*[*iff*]: $I \models s$ *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 $\{v \in I.\ \textit{atm-of}\ v \in \textit{atms-of}\ \psi\} \models \psi$
$\langle proof \rangle$

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

A simple application of the previous theorem:

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

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

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

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* $\psi\}$ 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*) $\equiv \forall\, I.$ *total-over-set I* (*atms-of* ψ) $\longrightarrow I \models \psi$

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** $-L \in\# A$
 shows *tautology A*
 $\langle proof \rangle$

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

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

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

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

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 $I \models \varphi \lor tautology\ \varphi$
$\langle proof \rangle$

lemma *tautology-imp-tautology*:
fixes $\chi\ \chi' :: 'v\ clause$
assumes $\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi'$ **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* :: $'a\ clause \Rightarrow 'a\ clause \Rightarrow bool$ (**infix** $\models f$ *49*) **where**
$\psi \models f\ \chi \longleftrightarrow (\forall I.\ total\text{-}over\text{-}m\ I\ (\{\psi\} \cup \{\chi\}) \longrightarrow consistent\text{-}interp\ I \longrightarrow I \models \psi \longrightarrow I \models \chi)$

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

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

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

lemma *true-cls-cls-refl*[*simp*]:
$A \models f\ A$
$\langle proof \rangle$

lemma *true-cls-cls-insert-l*[*simp*]:
$a \models f\ C \Longrightarrow insert\ a\ A \models p\ C$
$\langle proof \rangle$

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*]:
$N \models ps \ \{\}$
⟨*proof*⟩

lemma *true-clss-cls-subset*:
$A \subseteq B \Longrightarrow A \models p\ CC \Longrightarrow B \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cs-mono-l*[*simp*]:
$A \models p\ CC \Longrightarrow A \cup B\ \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cs-mono-l2*[*simp*]:
$B \models p\ CC \Longrightarrow A \cup B\ \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cls-mono-r*[*simp*]:
$A \models p\ CC \Longrightarrow A \models p\ CC + CC'$
⟨*proof*⟩

lemma *true-clss-cls-mono-r'*[*simp*]:
$A \models p\ CC' \Longrightarrow A \models p\ CC + CC'$
⟨*proof*⟩

lemma *true-clss-clss-union-l*[*simp*]:
$A \models ps\ CC \Longrightarrow A \cup B\ \models ps\ CC$
⟨*proof*⟩

lemma *true-clss-clss-union-l-r*[*simp*]:
$B \models ps\ CC \Longrightarrow A \cup B \models ps\ CC$
⟨*proof*⟩

lemma *true-clss-cls-in*[*simp*]:
$CC \in A \Longrightarrow A \models p\ CC$
⟨*proof*⟩

lemma *true-clss-cls-insert-l*[*simp*]:
$A \models p\ C \Longrightarrow insert\ a\ A \models p\ C$
⟨*proof*⟩

lemma *true-clss-clss-insert-l*[*simp*]:
$A \models ps\ C \Longrightarrow insert\ a\ A \models ps\ C$
⟨*proof*⟩

lemma *true-clss-clss-union-and*[*iff*]:
$A \models ps\ C \cup D \longleftrightarrow (A \models ps\ C \land A \models ps\ D)$
⟨*proof*⟩

lemma *true-clss-clss-insert*[*iff*]:
$A \models ps\ insert\ L\ Ls \longleftrightarrow (A \models p\ L \land A \models ps\ Ls)$
⟨*proof*⟩

lemma *true-clss-clss-subset*:
$A \subseteq B \Longrightarrow A \models ps\ CC \Longrightarrow B \models ps\ CC$
⟨*proof*⟩

lemma *union-trus-clss-clss*[*simp*]: $A \cup B \models ps\ B$
⟨*proof*⟩

lemma *true-clss-clss-remove*[*simp*]:
$A \models ps\ B \Longrightarrow A \models ps\ B - C$
⟨*proof*⟩

lemma *true-clss-clss-subsetE*:
$N \models ps\ B \Longrightarrow A \subseteq B \Longrightarrow N \models ps\ A$
⟨*proof*⟩

lemma *true-clss-clss-in-imp-true-clss-cls*:
assumes $N \models ps\ U$
and $A \in U$
shows $N \models p\ A$
⟨*proof*⟩

lemma *all-in-true-clss-clss*: $\forall x \in B.\ x \in A \Longrightarrow A \models ps\ B$
⟨*proof*⟩

lemma *true-clss-clss-left-right*:
assumes $A \models ps\ B$
and $A \cup B \models ps\ M$
shows $A \models ps\ M \cup B$
⟨*proof*⟩

lemma *true-clss-clss-generalise-true-clss-clss*:
$A \cup C \models ps\ D \Longrightarrow B \models ps\ C \Longrightarrow A \cup B \models ps\ D$
⟨*proof*⟩

lemma *true-clss-cls-or-true-clss-cls-or-not-true-clss-cls-or*:
assumes D: $N \models p\ D + \{\#- L\#\}$
and C: $N \models p\ C + \{\#L\#\}$
shows $N \models p\ D + C$
⟨*proof*⟩

lemma *true-cls-union-mset*[*iff*]: $I \models C\ \#\cup\ D \longleftrightarrow I \models C \lor I \models D$
⟨*proof*⟩

lemma *true-clss-cls-union-mset-true-clss-cls-or-not-true-clss-cls-or*:
assumes
D: $N \models p\ D + \{\#- L\#\}$ **and**
C: $N \models p\ C + \{\#L\#\}$
shows $N \models p\ D\ \#\cup\ C$
⟨*proof*⟩

lemma *satisfiable-carac*[*iff*]:
$(\exists I.\ \textit{consistent-interp}\ I \land I \models s\ \varphi) \longleftrightarrow \textit{satisfiable}\ \varphi$ (**is** $(\exists I.\ ?Q\ I) \longleftrightarrow ?S$)
⟨*proof*⟩

lemma *satisfiable-carac'*[*simp*]: $\textit{consistent-interp}\ I \Longrightarrow I \models s\ \varphi \Longrightarrow \textit{satisfiable}\ \varphi$
⟨*proof*⟩

11.3 Subsumptions

lemma *subsumption-total-over-m*:
assumes $A \subseteq\# B$
shows $\textit{total-over-m } I\ \{B\} \Longrightarrow \textit{total-over-m } I\ \{A\}$
$\langle \textit{proof} \rangle$

lemma *atms-of-replicate-mset-replicate-mset-uminus*[*simp*]:
$\textit{atms-of } (D - \textit{replicate-mset } (\textit{count } D\ L)\ L\ - \textit{replicate-mset } (\textit{count } D\ (-L))\ (-L))$
$= \textit{atms-of } D - \{\textit{atm-of } L\}$
$\langle \textit{proof} \rangle$

lemma *subsumption-chained*:
assumes
$\forall I.\ \textit{total-over-m } I\ \{D\} \longrightarrow I \models D \longrightarrow I \models \varphi$ **and**
$C \subseteq\# D$
shows $(\forall I.\ \textit{total-over-m } I\ \{C\} \longrightarrow I \models C \longrightarrow I \models \varphi) \lor \textit{tautology } \varphi$
$\langle \textit{proof} \rangle$

11.4 Removing Duplicates

lemma *tautology-remdups-mset*[*iff*]:
$\textit{tautology } (\textit{remdups-mset } C) \longleftrightarrow \textit{tautology } C$
$\langle \textit{proof} \rangle$

lemma *atms-of-remdups-mset*[*simp*]: $\textit{atms-of } (\textit{remdups-mset } C) = \textit{atms-of } C$
$\langle \textit{proof} \rangle$

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

lemma *true-clss-cls-remdups-mset*[*iff*]: $A \models p\ \textit{remdups-mset } C \longleftrightarrow A \models p\ C$
$\langle \textit{proof} \rangle$

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* :: $'v\ \textit{set} \Rightarrow\ 'v\ \textit{clause set}$ **where**
$\textit{simple-clss atms} = \{C.\ \textit{atms-of } C \subseteq \textit{atms} \land \neg\textit{tautology } C \land \textit{distinct-mset } C\}$

lemma *simple-clss-empty*[*simp*]:
$\textit{simple-clss } \{\} = \{\{\#\}\}$
$\langle \textit{proof} \rangle$

lemma *simple-clss-insert*:
assumes $l \notin \textit{atms}$

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
    $x \in$ *simple-clss atms*
  shows *atms-of* $x \subseteq$ *atms* $\wedge$ ¬*tautology* $x \wedge$ *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 $\chi \in$ *simple-clss* (*atms-of* χ)
  ⟨*proof*⟩

lemma *simplified-in-simple-clss*:
  assumes *distinct-mset-set* ψ **and** $\forall \chi \in \psi$. ¬*tautology* χ
  shows $\psi \subseteq$ *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*]: $I \neq \{\} \Longrightarrow$ *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* {}

$\langle proof \rangle$

lemma *entails-single*[*iff*]:
$I \models es\ \{a\} \longleftrightarrow I \models e\ a$
$\langle proof \rangle$

lemma *entails-insert-l*[*simp*]:
$M \models es\ A \Longrightarrow insert\ L\ M \models es\ A$
$\langle proof \rangle$

lemma *entails-union*[*iff*]: $I \models es\ CC \cup DD \longleftrightarrow I \models es\ CC \wedge I \models es\ DD$
$\langle proof \rangle$

lemma *entails-insert*[*iff*]: $I \models es\ insert\ C\ DD \longleftrightarrow I \models e\ C \wedge I \models es\ DD$
$\langle proof \rangle$

lemma *entails-insert-mono*: $DD \subseteq CC \Longrightarrow I \models es\ CC \Longrightarrow I \models es\ DD$
$\langle proof \rangle$

lemma *entails-union-increase*[*simp*]:
assumes $I \models es\ \psi$
shows $I \cup I' \models es\ \psi$
$\langle proof \rangle$

lemma *true-clss-commute-l*:
$(I \cup I' \models es\ \psi) \longleftrightarrow (I' \cup I \models es\ \psi)$
$\langle proof \rangle$

lemma *entails-remove*[*simp*]: $I \models es\ N \Longrightarrow I \models es\ Set.remove\ a\ N$
$\langle proof \rangle$

lemma *entails-remove-minus*[*simp*]: $I \models es\ N \Longrightarrow I \models es\ N - A$
$\langle proof \rangle$

end

interpretation *true-cls*: *entail true-cls*
$\langle proof \rangle$

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
$I \models sext\ N \longleftrightarrow (\forall J.\ I \subseteq J \longrightarrow consistent\text{-}interp\ J \longrightarrow total\text{-}over\text{-}m\ J\ N \longrightarrow J \models s\ N)$

lemma *true-clss-imp-true-cls-ext*:
$I \models s\ N \Longrightarrow I \models sext\ N$
$\langle proof \rangle$

lemma *true-clss-ext-decrease-right-remove-r*:
assumes $I \models sext\ N$
shows $I \models sext\ N - \{C\}$
⟨*proof*⟩

lemma *consistent-true-clss-ext-satisfiable*:
assumes *consistent-interp* I **and** $I \models sext\ A$
shows *satisfiable* A
⟨*proof*⟩

lemma *not-consistent-true-clss-ext*:
assumes $\neg$*consistent-interp* I
shows $I \models sext\ A$
⟨*proof*⟩
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**
$mset\text{-}of\text{-}conj\ (FOr\ \varphi\ \psi) = mset\text{-}of\text{-}conj\ \varphi + mset\text{-}of\text{-}conj\ \psi$ |
$mset\text{-}of\text{-}conj\ (FVar\ v) = \{\#\ Pos\ v\ \#\}$ |
$mset\text{-}of\text{-}conj\ (FNot\ (FVar\ v)) = \{\#\ Neg\ v\ \#\}$ |
$mset\text{-}of\text{-}conj\ FF = \{\#\}$

fun *mset-of-formula* :: $'a\ propo \Rightarrow 'a\ literal\ multiset\ set$ **where**
$mset\text{-}of\text{-}formula\ (FAnd\ \varphi\ \psi) = mset\text{-}of\text{-}formula\ \varphi \cup mset\text{-}of\text{-}formula\ \psi$ |
$mset\text{-}of\text{-}formula\ (FOr\ \varphi\ \psi) = \{mset\text{-}of\text{-}conj\ (FOr\ \varphi\ \psi)\}$ |
$mset\text{-}of\text{-}formula\ (FVar\ \psi) = \{mset\text{-}of\text{-}conj\ (FVar\ \psi)\}$ |
$mset\text{-}of\text{-}formula\ (FNot\ \psi) = \{mset\text{-}of\text{-}conj\ (FNot\ \psi)\}$ |
$mset\text{-}of\text{-}formula\ FF = \{\{\#\}\}$ |
$mset\text{-}of\text{-}formula\ FT = \{\}$

12.2 Equisatisfiability of the two Version

lemma *is-conj-with-TF-FNot*:
$is\text{-}conj\text{-}with\text{-}TF\ (FNot\ \varphi) \longleftrightarrow (\exists v.\ \varphi = FVar\ v \lor \varphi = FF \lor \varphi = FT)$
⟨*proof*⟩

lemma *grouped-by-COr-FNot*:
$grouped\text{-}by\ COr\ (FNot\ \varphi) \longleftrightarrow (\exists v.\ \varphi = FVar\ v \lor \varphi = FF \lor \varphi = FT)$
⟨*proof*⟩

lemma
shows *no-T-F-FF*[*simp*]: $\neg$*no-T-F FF* **and**
no-T-F-FT[*simp*]: $\neg$*no-T-F FT*
⟨*proof*⟩

lemma *grouped-by-CAnd-FAnd*:
$grouped\text{-}by\ CAnd\ (FAnd\ \varphi 1\ \varphi 2) \longleftrightarrow grouped\text{-}by\ CAnd\ \varphi 1 \land grouped\text{-}by\ CAnd\ \varphi 2$

⟨*proof*⟩

lemma *grouped-by-COr-FOr*:
grouped-by COr (*FOr* $\varphi 1$ $\varphi 2$) $\longleftrightarrow$ *grouped-by COr* $\varphi 1$ $\wedge$ *grouped-by COr* $\varphi 2$
⟨*proof*⟩

lemma *grouped-by-COr-FAnd*[*simp*]: $\neg$ *grouped-by COr* (*FAnd* $\varphi 1$ $\varphi 2$)
⟨*proof*⟩

lemma *grouped-by-COr-FEq*[*simp*]: $\neg$ *grouped-by COr* (*FEq* $\varphi 1$ $\varphi 2$)
⟨*proof*⟩

lemma [*simp*]: $\neg$*grouped-by COr* (*FImp* φ ψ)
⟨*proof*⟩

lemma [*simp*]: $\neg$ *is-conj-with-TF* (*FImp* φ ψ)
⟨*proof*⟩

lemma [*simp*]: $\neg$*grouped-by COr* (*FEq* φ ψ)
⟨*proof*⟩

lemma [*simp*]: $\neg$ *is-conj-with-TF* (*FEq* φ ψ)
⟨*proof*⟩

lemma *is-conj-with-TF-Fand*:
is-conj-with-TF (*FAnd* $\varphi 1$ $\varphi 2$) $\Longrightarrow$ *is-conj-with-TF* $\varphi 1$ $\wedge$ *is-conj-with-TF* $\varphi 2$
⟨*proof*⟩

lemma *is-conj-with-TF-FOr*:
is-conj-with-TF (*FOr* $\varphi 1$ $\varphi 2$) $\Longrightarrow$ *grouped-by COr* $\varphi 1$ $\wedge$ *grouped-by COr* $\varphi 2$
⟨*proof*⟩

lemma *grouped-by-COr-mset-of-formula*:
grouped-by COr φ $\Longrightarrow$ *mset-of-formula* φ = (*if* φ = *FT then* {} *else* {*mset-of-conj* φ})
⟨*proof*⟩

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 φ :: *'v propo*
assumes *is-cnf* φ
shows *eval A* φ $\longleftrightarrow$ *Partial-Clausal-Logic.true-clss* ({*Pos v*|*v*. *A v*} $\cup$ {*Neg v*|*v*. $\neg$*A v*})
(*mset-of-formula* φ)
⟨*proof*⟩

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:
  $(A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\}) \in N \Longrightarrow simplify\ N\ (N - \{A + \{\#Pos\ P\#\} + \{\#Neg\ P\#\}\})$ |
condensation:
  $(A + \{\#L\#\} + \{\#L\#\}) \in N \Longrightarrow simplify\ N\ (N - \{A + \{\#L\#\} + \{\#L\#\}\} \cup \{A + \{\#L\#\}\})$ |
subsumption:
  $A \in N \Longrightarrow A \subset\# B \Longrightarrow B \in N \Longrightarrow simplify\ N\ (N - \{B\})$

lemma *simplify-preserves-un-sat'*:
 fixes $N\ N'$ :: $'v$ *clauses*
 assumes *simplify* $N\ N'$
 and *total-over-m* $I\ N$
 shows $I \models s\ N' \longrightarrow I \models s\ N$
 ⟨*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 *rtranclp-simplify-preserves-finite*:
 assumes *rtranclp 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 *rtranclp-simplify-atms-of-ms*:

assumes *rtranclp simplify* ψ ψ'
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
⟨*proof*⟩

lemma *factoring-imp-simplify*:
assumes $\{\#L\#\} + \{\#L\#\} + C \in N$
shows $\exists N'.$ *simplify* N N'
⟨*proof*⟩

13.2 Unconstrained Resolution

type-synonym *'v uncon-state* = *'v clauses*
inductive *uncon-res* :: *'v uncon-state* ⇒ *'v uncon-state* ⇒ *bool* **where**
resolution:
$\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$ *already-used*
$\Longrightarrow$ *uncon-res* (N) $(N \cup \{C + D\})$ |
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow$ *uncon-res* N $(N \cup \{C + \{\#L\#\}\})$

lemma *uncon-res-increasing*:
assumes *uncon-res* S S' **and** $\psi \in S$
shows $\psi \in S'$
⟨*proof*⟩

lemma *rtranclp-uncon-inference-increasing*:
assumes *rtranclp uncon-res* S S' **and** $\psi \in S$
shows $\psi \in S'$
⟨*proof*⟩

13.2.1 Subsumption

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

lemma *subsumes-refl*[*simp*]:
subsumes χ χ
⟨*proof*⟩

lemma *subsumes-subsumption*:
assumes *subsumes* D χ
and $C \subset\# D$ **and** $\neg$*tautology* χ
shows *subsumes* C χ ⟨*proof*⟩

lemma *subsumes-tautology*:
assumes *subsumes* $(C + \{\#Pos\ P\#\} + \{\#Neg\ P\#\})$ χ
shows *tautology* χ
⟨*proof*⟩

13.3 Inference Rule

type-synonym *'v state* = *'v clauses* × (*'v clause* × *'v clause*) *set*
inductive *inference-clause* :: *'v state* ⇒ *'v clause* × (*'v clause* × *'v clause*) *set* ⇒ *bool*
(**infix** $\Rightarrow_{\mathrm{Res}}$ *100*) **where**

resolution:
$\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$ *already-used*
$\Longrightarrow$ *inference-clause* $(N, \textit{already-used})$ $(C + D, \textit{already-used} \cup \{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\})$ |
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow$ *inference-clause* $(N, \textit{already-used})$ $(C + \{\#L\#\}, \textit{already-used})$

inductive *inference* :: $'v\ state \Rightarrow\ 'v\ state \Rightarrow bool$ **where**
inference-step: *inference-clause* S $(\textit{clause}, \textit{already-used})$
$\Longrightarrow$ *inference* S $(\textit{fst}\ S \cup \{\textit{clause}\}, \textit{already-used})$

abbreviation *already-used-inv*
:: $'a\ literal\ multiset\ set \times ('a\ literal\ multiset \times\ 'a\ literal\ multiset)\ set \Rightarrow bool$ **where**
already-used-inv state $\equiv$
$(\forall (A, B) \in \textit{snd state}.\ \exists p.\ Pos\ p \in\#\ A \wedge Neg\ p \in\#\ B \wedge$
$((\exists \chi \in \textit{fst state}.\ \textit{subsumes}\ \chi\ ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\})))$
$\vee\ \textit{tautology}\ ((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))))$

lemma *inference-clause-preserves-already-used-inv*:
assumes *inference-clause* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* $(\textit{fst}\ S \cup \{\textit{fst}\ S'\}, \textit{snd}\ S')$
$\langle proof \rangle$

lemma *inference-preserves-already-used-inv*:
assumes *inference* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
$\langle proof \rangle$

lemma *rtranclp-inference-preserves-already-used-inv*:
assumes *rtranclp inference* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
$\langle proof \rangle$

lemma *subsumes-condensation*:
assumes *subsumes* $(C + \{\#L\#\} + \{\#L\#\})\ D$
shows *subsumes* $(C + \{\#L\#\})\ D$
$\langle proof \rangle$

lemma *simplify-preserves-already-used-inv*:
assumes *simplify* $N\ N'$
and *already-used-inv* $(N, \textit{already-used})$
shows *already-used-inv* $(N', \textit{already-used})$
$\langle proof \rangle$

lemma
factoring-satisfiable: $I \models \{\#L\#\} + \{\#L\#\} + C \longleftrightarrow I \models \{\#L\#\} + C$ **and**
resolution-satisfiable:
consistent-interp $I \Longrightarrow I \models \{\#Pos\ p\#\} + C \Longrightarrow I \models \{\#Neg\ p\#\} + D \Longrightarrow I \models C + D$ **and**
factoring-same-vars: *atms-of* $(\{\#L\#\} + \{\#L\#\} + C) =$ *atms-of* $(\{\#L\#\} + C)$
$\langle proof \rangle$

lemma *inference-increasing*:
assumes *inference* S S' **and** $\psi \in$ *fst* S
shows $\psi \in$ *fst* S'
⟨*proof*⟩

lemma *rtranclp-inference-increasing*:
assumes *rtranclp inference* S S' **and** $\psi \in$ *fst* S
shows $\psi \in$ *fst* S'
⟨*proof*⟩

lemma *inference-clause-already-used-increasing*:
assumes *inference-clause* S S'
shows *snd* $S \subseteq$ *snd* S'
⟨*proof*⟩

lemma *inference-already-used-increasing*:
assumes *inference* S S'
shows *snd* $S \subseteq$ *snd* S'
⟨*proof*⟩

lemma *inference-clause-preserves-un-sat*:
fixes N N' :: *'v clauses*
assumes *inference-clause* T T'
and *total-over-m* I (*fst* T)
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $T \longleftrightarrow I \models s$ *fst* $T \cup \{$*fst* $T'\}$
⟨*proof*⟩

lemma *inference-preserves-un-sat*:
fixes N N' :: *'v clauses*
assumes *inference* T T'
and *total-over-m* I (*fst* T)
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $T \longleftrightarrow I \models s$ *fst* T'
⟨*proof*⟩

lemma *inference-clause-preserves-atms-of-ms*:
assumes *inference-clause* S S'
shows *atms-of-ms* (*fst* (*fst* $S \cup \{$*fst* $S'\}$, *snd* S')) $\subseteq$ *atms-of-ms* (*fst* S)
⟨*proof*⟩

lemma *inference-preserves-atms-of-ms*:
fixes N N' :: *'v clauses*
assumes *inference* T T'
shows *atms-of-ms* (*fst* T') $\subseteq$ *atms-of-ms* (*fst* T)
⟨*proof*⟩

lemma *inference-preserves-total*:
fixes N N' :: *'v clauses*
assumes *inference* (N, *already-used*) (N', *already-used'*)
shows *total-over-m* I $N \Longrightarrow$ *total-over-m* I N'
⟨*proof*⟩

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference* T T'
shows *total-over-m* I (*fst* T) $\Longrightarrow$ *total-over-m* I (*fst* T')
⟨*proof*⟩

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'
⟨*proof*⟩

lemma *inference-preserves-finite*:
assumes *inference* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *inference-preserves-finite-snd*:
assumes *inference* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *rtranclp-inference-preserves-finite*:
assumes *rtranclp inference* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *consistent-interp-insert*:
assumes *consistent-interp* I
and *atm-of* $P \notin$ *atm-of* ‘ I
shows *consistent-interp* (*insert* P I)
⟨*proof*⟩

lemma *simplify-clause-preserves-sat*:
assumes *simp*: *simplify* ψ ψ'
and *satisfiable* ψ'
shows *satisfiable* ψ
⟨*proof*⟩

lemma *simplify-preserves-unsat*:
assumes *inference* ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *inference-preserves-unsat*:

assumes *inference*** S S'
shows *satisfiable* (*fst* S') $\longrightarrow$ *satisfiable* (*fst* S)
⟨*proof*⟩

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

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

lemma *sem-tree-size*[*case-names bigger*]:
($\bigwedge$*xs*:: *'v sem-tree*. ($\bigwedge$*ys*:: *'v sem-tree*. *sem-tree-size ys* < *sem-tree-size xs* $\Longrightarrow$ *P ys*) $\Longrightarrow$ *P xs*)
$\Longrightarrow$ *P xs*
⟨*proof*⟩

fun *partial-interps* :: *'v sem-tree* ⇒ *'v interp* ⇒ *'v clauses* ⇒ *bool* **where**
partial-interps Leaf I ψ = ($\exists\chi$. $\neg$ *I* $\models$ χ $\wedge$ $\chi \in \psi$ $\wedge$ *total-over-m 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' $\Longrightarrow$ *partial-interps Leaf I N* $\Longrightarrow$ *partial-interps Leaf I N'*
⟨*proof*⟩

lemma *simplify-preserve-partial-tree*:
assumes *simplify N N'*
and *partial-interps t I N*
shows *partial-interps t I N'*
⟨*proof*⟩

lemma *inference-preserve-partial-tree*:
assumes *inference S S'*
and *partial-interps t I* (*fst S*)
shows *partial-interps t I* (*fst S'*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

function *build-sem-tree* :: *'v* :: *linorder set* ⇒ *'v clauses* ⇒ *'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))
⟨*proof*⟩

termination
⟨*proof*⟩
declare *build-sem-tree.induct*[*case-names tree*]

lemma *unsatisfiable-empty*[*simp*]:
$\neg$*unsatisfiable* {}
⟨*proof*⟩

lemma *partial-interps-build-sem-tree-atms-general*:
fixes ψ :: $'v$ :: *linorder clauses* **and** p :: $'v$ *literal list*
assumes *unsat*: *unsatisfiable* ψ **and** *finite* ψ **and** *consistent-interp* I
and *finite atms*
and *atms-of-ms* ψ = *atms* $\cup$ *atms-of-s* I **and** *atms* $\cap$ *atms-of-s* I = {}
shows *partial-interps* (*build-sem-tree atms* ψ) I ψ
⟨*proof*⟩

lemma *partial-interps-build-sem-tree-atms*:
fixes ψ :: $'v$ :: *linorder clauses* **and** p :: $'v$ *literal list*
assumes *unsat*: *unsatisfiable* ψ **and** *finite*: *finite* ψ
shows *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) {} ψ
⟨*proof*⟩

lemma *can-decrease-count*:
fixes ψ'' :: $'v$ *clauses* $\times$ ($'v$ *clause* $\times$ $'v$ *clause* $\times$ $'v$) *set*
assumes *count* χ $L = n$
and $L \in\#\ \chi$ **and** $\chi \in$ *fst* ψ
shows $\exists \psi'\ \chi'$. *inference*** $\psi\ \psi' \wedge \chi' \in$ *fst* $\psi' \wedge (\forall L.\ L \in\#\ \chi \longleftrightarrow L \in\#\ \chi')$
$\wedge$ *count* χ' $L = 1$
$\wedge$ ($\forall \varphi$. $\varphi \in$ *fst* $\psi \longrightarrow \varphi \in$ *fst* ψ')
$\wedge$ ($I \models \chi \longleftrightarrow I \models \chi'$)
$\wedge$ ($\forall I'$. *total-over-m* I' {χ} $\longrightarrow$ *total-over-m* I' {χ'})
⟨*proof*⟩

lemma *can-decrease-tree-size*:
fixes ψ :: $'v$ *state* **and** *tree* :: $'v$ *sem-tree*
assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
and *partial-interps tree* I (*fst* ψ)
shows $\exists$(*tree*$'$:: $'v$ *sem-tree*) ψ'. *inference*** $\psi\ \psi' \wedge$ *partial-interps tree*$'$ I (*fst* ψ')
$\wedge$ (*sem-tree-size tree*$'$ < *sem-tree-size tree* $\vee$ *sem-tree-size tree* = *0*)
⟨*proof*⟩

lemma *inference-completeness-inv*:
fixes ψ :: $'v$ ::*linorder state*
assumes
unsat: $\neg$*satisfiable* (*fst* ψ) **and**
finite: *finite* (*fst* ψ) **and**
a-u-v: *already-used-inv* ψ
shows $\exists \psi'$. (*inference*** $\psi\ \psi' \wedge$ {#} $\in$ *fst* ψ')
⟨*proof*⟩

lemma *inference-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *unsat*: $\neg$*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)

and *snd* ψ = {}
shows $\exists\,\psi'$. (*rtranclp inference* ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')
⟨*proof*⟩

lemma *inference-soundness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *rtranclp inference* ψ ψ' **and** {#} $\in$ *fst* ψ'
shows *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *inference-soundness-and-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows ($\exists\,\psi'$. (*inference*** ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

13.4 Lemma about the simplified state

abbreviation *simplified* ψ $\equiv$ (*no-step simplify* ψ)

lemma *simplified-count*:
assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
shows *count* χ $L \leq 1$
⟨*proof*⟩

lemma *simplified-no-both*:
assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
shows $\neg$ ($L \in\# \chi \wedge -L \in\# \chi$)
⟨*proof*⟩

lemma *simplified-not-tautology*:
assumes *simplified* {ψ}
shows ~*tautology* ψ
⟨*proof*⟩

lemma *simplified-remove*:
assumes *simplified* {ψ}
shows *simplified* {ψ − {#*l*#}}
⟨*proof*⟩

lemma *in-simplified-simplified*:
assumes *simp*: *simplified* ψ **and** *incl*: $\psi' \subseteq \psi$
shows *simplified* ψ'
⟨*proof*⟩

lemma *simplified-in*:
assumes *simplified* ψ
and $N \in \psi$
shows *simplified* {N}
⟨*proof*⟩

lemma *subsumes-imp-formula*:
assumes $\psi \leq\# \varphi$
shows {ψ} $\models p$ φ

⟨*proof*⟩

lemma *simplified-imp-distinct-mset-tauto*:
assumes *simp*: *simplified* ψ'
shows *distinct-mset-set* ψ' **and** $\forall \chi \in \psi'$. $\neg$*tautology* χ
⟨*proof*⟩

lemma *simplified-no-more-full1-simplified*:
assumes *simplified* ψ
shows $\neg$*full1 simplify* ψ ψ'
⟨*proof*⟩

13.5 Resolution and Invariants

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

13.5.1 Invariants

lemma *resolution-finite*:
assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *rtranclp-resolution-finite*:
assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
⟨*proof*⟩

lemma *resolution-finite-snd*:
assumes *resolution* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *rtranclp-resolution-finite-snd*:
assumes *resolution*** ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *resolution-always-simplified*:
assumes *resolution* ψ ψ'
shows *simplified* (*fst* ψ')
⟨*proof*⟩

lemma *tranclp-resolution-always-simplified*:
assumes *tranclp resolution* ψ ψ'
shows *simplified* (*fst* ψ')
⟨*proof*⟩

lemma *resolution-atms-of*:
assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-resolution-atms-of*:
  assumes $resolution^{**}\ \psi\ \psi'$ **and** *finite* $(fst\ \psi)$
  shows $atms\text{-}of\text{-}ms\ (fst\ \psi') \subseteq atms\text{-}of\text{-}ms\ (fst\ \psi)$
  ⟨*proof*⟩

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

lemma *rtranclp-resolution-include*:
  assumes *res*: $tranclp\ resolution\ \psi\ \psi'$ **and** *finite*: *finite* $(fst\ \psi)$
  shows $fst\ \psi' \subseteq simple\text{-}clss\ (atms\text{-}of\text{-}ms\ (fst\ \psi))$
  ⟨*proof*⟩

abbreviation *already-used-all-simple*
  $:: ('a\ literal\ multiset \times 'a\ literal\ multiset)\ set \Rightarrow 'a\ set \Rightarrow bool$ **where**
already-used-all-simple already-used vars $\equiv$
$(\forall\ (A,\ B) \in already\text{-}used.\ simplified\ \{A\} \wedge simplified\ \{B\} \wedge atms\text{-}of\ A \subseteq vars \wedge atms\text{-}of\ B \subseteq vars)$

lemma *already-used-all-simple-vars-incl*:
  assumes $vars \subseteq vars'$
  shows $already\text{-}used\text{-}all\text{-}simple\ a\ vars \Longrightarrow already\text{-}used\text{-}all\text{-}simple\ a\ vars'$
  ⟨*proof*⟩

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 $atms\text{-}of\text{-}ms\ (fst\ S) \subseteq vars$
  shows $already\text{-}used\text{-}all\text{-}simple\ (snd\ (fst\ S \cup \{fst\ S'\},\ snd\ S'))\ vars$
  ⟨*proof*⟩

lemma *inference-preserves-already-used-all-simple*:
  assumes *inference* $S\ S'$
  and *already-used-all-simple* $(snd\ S)$ *vars*
  and *simplified* $(fst\ S)$
  and $atms\text{-}of\text{-}ms\ (fst\ S) \subseteq vars$
  shows *already-used-all-simple* $(snd\ S')$ *vars*
  ⟨*proof*⟩

lemma *already-used-all-simple-inv*:
  assumes *resolution* $S\ S'$
  and *already-used-all-simple* $(snd\ S)$ *vars*
  and $atms\text{-}of\text{-}ms\ (fst\ S) \subseteq vars$
  shows *already-used-all-simple* $(snd\ S')$ *vars*
  ⟨*proof*⟩

lemma *rtranclp-already-used-all-simple-inv*:
  assumes $resolution^{**}\ S\ S'$
  and *already-used-all-simple* $(snd\ S)$ *vars*
  and $atms\text{-}of\text{-}ms\ (fst\ S) \subseteq vars$
  and *finite* $(fst\ S)$
  shows *already-used-all-simple* $(snd\ S')$ *vars*
  ⟨*proof*⟩

lemma *inference-clause-simplified-already-used-subset*:
  assumes *inference-clause* S S'
  and *simplified* (*fst* S)
  shows *snd* $S \subset$ *snd* S'
  ⟨*proof*⟩

lemma *inference-simplified-already-used-subset*:
  assumes *inference* S S'
  and *simplified* (*fst* S)
  shows *snd* $S \subset$ *snd* S'
  ⟨*proof*⟩

lemma *resolution-simplified-already-used-subset*:
  assumes *resolution* S S'
  and *simplified* (*fst* S)
  shows *snd* $S \subset$ *snd* S'
  ⟨*proof*⟩

lemma *tranclp-resolution-simplified-already-used-subset*:
  assumes *tranclp resolution* S S'
  and *simplified* (*fst* S)
  shows *snd* $S \subset$ *snd* S'
  ⟨*proof*⟩

abbreviation *already-used-top vars* ≡ *simple-clss vars* × *simple-clss vars*

lemma *already-used-all-simple-in-already-used-top*:
  assumes *already-used-all-simple s vars* **and** *finite vars*
  shows *s* ⊆ *already-used-top vars*
⟨*proof*⟩

lemma *already-used-top-finite*:
  assumes *finite vars*
  shows *finite* (*already-used-top vars*)
  ⟨*proof*⟩

lemma *already-used-top-increasing*:
  assumes *var* ⊆ *var'* **and** *finite var'*
  shows *already-used-top var* ⊆ *already-used-top var'*
  ⟨*proof*⟩

lemma *already-used-all-simple-finite*:
  fixes *s* :: (*'a literal multiset* × *'a literal multiset*) *set* **and** *vars* :: *'a set*
  assumes *already-used-all-simple s vars* **and** *finite vars*
  shows *finite s*
  ⟨*proof*⟩

abbreviation *card-simple vars* ψ ≡ *card* (*already-used-top vars* − ψ)

lemma *resolution-card-simple-decreasing*:
  assumes *res*: *resolution* ψ ψ'
  and *a-u-s*: *already-used-all-simple* (*snd* ψ) *vars*
  and *finite-v*: *finite vars*
  and *finite-fst*: *finite* (*fst* ψ)

and *finite-snd*: *finite* (*snd* ψ)
and *simp*: *simplified* (*fst* ψ)
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
⟨*proof*⟩

lemma *tranclp-resolution-card-simple-decreasing*:
assumes *tranclp resolution* ψ ψ' **and** *finite-fst*: *finite* (*fst* ψ)
and *already-used-all-simple* (*snd* ψ) *vars*
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
and *finite-v*: *finite vars*
and *finite-snd*: *finite* (*snd* ψ)
and *simplified* (*fst* ψ)
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
⟨*proof*⟩

lemma *tranclp-resolution-card-simple-decreasing-2*:
assumes *tranclp resolution* ψ ψ'
and *finite-fst*: *finite* (*fst* ψ)
and *empty-snd*: *snd* ψ = {}
and *simplified* (*fst* ψ)
shows *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ') $<$ *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ)
⟨*proof*⟩

13.5.2 well-foundness if the relation

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

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

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

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)
⟨*proof*⟩

lemma *full-simplify-already-used-inv*:
assumes *full simplify* S S'
and *already-used-inv* (S, N)
shows *already-used-inv* (S', N)
⟨*proof*⟩
lemma *resolution-already-used-inv*:
assumes *resolution* S S'
and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *rtranclp-resolution-already-used-inv*:
assumes *resolution*** S S'
and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *rtanclp-simplify-preserves-unsat*:
assumes *simplify*** ψ ψ'
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
⟨*proof*⟩

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* ψ ψ'
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
⟨*proof*⟩

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* ψ ψ'
shows *satisfiable* $\psi' \longrightarrow$ *satisfiable* ψ
⟨*proof*⟩

lemma *resolution-preserves-unsat*:
assumes *resolution* ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-resolution-preserves-unsat*:
assumes *resolution*** ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-simplify-preserve-partial-tree*:
assumes *simplify*** N N'
and *partial-interps* t I N
shows *partial-interps* t I N'
⟨*proof*⟩

lemma *full1-simplify-preserve-partial-tree*:
assumes *full1 simplify* N N'
and *partial-interps* t I N

shows *partial-interps t I N'*
⟨*proof*⟩

lemma *full-simplify-preserve-partial-tree*:
assumes *full simplify N N'*
and *partial-interps t I N*
shows *partial-interps t I N'*
⟨*proof*⟩

lemma *resolution-preserve-partial-tree*:
assumes *resolution S S'*
and *partial-interps t I (fst S)*
shows *partial-interps t I (fst S')*
⟨*proof*⟩

lemma *rtranclp-resolution-preserve-partial-tree*:
assumes $resolution^{**}$ *S S'*
and *partial-interps t I (fst S)*
shows *partial-interps t I (fst S')*
⟨*proof*⟩
thm *nat-less-induct nat.induct*

lemma *nat-ge-induct*[*case-names 0 Suc*]:
assumes *P 0*
and $(\bigwedge n. (\bigwedge m. m<Suc\ n \Longrightarrow P\ m) \Longrightarrow P\ (Suc\ n))$
shows *P n*
⟨*proof*⟩

lemma *wf-always-more-step-False*:
assumes *wf R*
shows $(\forall x. \exists z. (z, x)\in R) \Longrightarrow False$
⟨*proof*⟩

lemma *finite-finite-mset-element-of-mset*[*simp*]:
assumes *finite N*
shows *finite* $\{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \land L \in\#\ \varphi \land P\ \varphi\ L\}$
⟨*proof*⟩

value *card*
value *filter-mset*
value $\{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}$
value $(\lambda\varphi.\ msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\})$

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

definition *sum-count-ge-2* :: *'a multiset set* ⇒ *nat* (Ξ) **where**
sum-count-ge-2 ≡ *folding.F* $(\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}))\ 0$

interpretation *sum-count-ge-2*:
folding $(\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}))\ 0$
rewrites
folding.F $(\lambda\varphi.\ op\ +(msetsum\ \{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \leq count\ \varphi\ L\#\}))\ 0 = sum\text{-}count\text{-}ge\text{-}2$
⟨*proof*⟩

lemma *finite-incl-le-setsum*:
finite $(B::'a\ multiset\ set) \Longrightarrow A \subseteq B \Longrightarrow \Xi\ A \leq \Xi\ B$
⟨*proof*⟩

lemma *simplify-finite-measure-decrease*:
simplify $N\ N' \Longrightarrow finite\ N \Longrightarrow card\ N' + \Xi\ N' < card\ N + \Xi\ N$
⟨*proof*⟩

lemma *simplify-terminates*:
wf $\{(N', N).\ finite\ N \wedge simplify\ N\ N'\}$
⟨*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 $\exists N'.\ simplify^{**}\ N\ N' \wedge simplified\ N'$
⟨*proof*⟩

lemma *finite-simplified-full1-simp*:
assumes *finite N*
shows $simplified\ N \vee (\exists N'.\ full1\ simplify\ N\ N')$
⟨*proof*⟩

lemma *finite-simplified-full-simp*:
assumes *finite N*
shows $\exists N'.\ full\ simplify\ N\ N'$
⟨*proof*⟩

lemma *can-decrease-tree-size-resolution*:
fixes ψ :: *'v state* **and** *tree* :: *'v sem-tree*
assumes *finite* $(fst\ \psi)$ **and** *already-used-inv* ψ
and *partial-interps tree I* $(fst\ \psi)$
and *simplified* $(fst\ \psi)$
shows $\exists (tree' :: 'v\ sem\text{-}tree)\ \psi'.\ resolution^{**}\ \psi\ \psi' \wedge partial\text{-}interps\ tree'\ I\ (fst\ \psi')$
$\wedge (sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ tree \vee sem\text{-}tree\text{-}size\ tree = 0)$
⟨*proof*⟩

lemma *resolution-completeness-inv*:
fixes ψ :: *'v ::linorder state*
assumes
unsat: $\neg satisfiable\ (fst\ \psi)$ **and**
finite: *finite* $(fst\ \psi)$ **and**
a-u-v: *already-used-inv* ψ

shows $\exists \psi'. (\textit{resolution}^{**}\ \psi\ \psi' \wedge \{\#\} \in \textit{fst}\ \psi')$
$\langle \textit{proof} \rangle$

lemma *resolution-preserves-already-used-inv*:
assumes *resolution* $S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
$\langle \textit{proof} \rangle$

lemma *rtranclp-resolution-preserves-already-used-inv*:
assumes $\textit{resolution}^{**}\ S\ S'$
and *already-used-inv* S
shows *already-used-inv* S'
$\langle \textit{proof} \rangle$

lemma *resolution-completeness*:
fixes $\psi :: {'v} :: \textit{linorder state}$
assumes *unsat*: $\neg \textit{satisfiable}\ (\textit{fst}\ \psi)$
and *finite*: *finite* $(\textit{fst}\ \psi)$
and $\textit{snd}\ \psi = \{\}$
shows $\exists \psi'. (\textit{resolution}^{**}\ \psi\ \psi' \wedge \{\#\} \in \textit{fst}\ \psi')$
$\langle \textit{proof} \rangle$

lemma *rtranclp-preserves-sat*:
assumes $\textit{simplify}^{**}\ S\ S'$
and *satisfiable* S
shows *satisfiable* S'
$\langle \textit{proof} \rangle$

lemma *resolution-preserves-sat*:
assumes *resolution* $S\ S'$
and *satisfiable* $(\textit{fst}\ S)$
shows *satisfiable* $(\textit{fst}\ S')$
$\langle \textit{proof} \rangle$

lemma *rtranclp-resolution-preserves-sat*:
assumes $\textit{resolution}^{**}\ S\ S'$
and *satisfiable* $(\textit{fst}\ S)$
shows *satisfiable* $(\textit{fst}\ S')$
$\langle \textit{proof} \rangle$

lemma *resolution-soundness*:
fixes $\psi :: {'v} :: \textit{linorder state}$
assumes $\textit{resolution}^{**}\ \psi\ \psi'$ **and** $\{\#\} \in \textit{fst}\ \psi'$
shows *unsatisfiable* $(\textit{fst}\ \psi)$
$\langle \textit{proof} \rangle$

lemma *resolution-soundness-and-completeness*:
fixes $\psi :: {'v} :: \textit{linorder state}$
assumes *finite*: *finite* $(\textit{fst}\ \psi)$
and *snd*: $\textit{snd}\ \psi = \{\}$
shows $(\exists \psi'. (\textit{resolution}^{**}\ \psi\ \psi' \wedge \{\#\} \in \textit{fst}\ \psi')) \longleftrightarrow \textit{unsatisfiable}\ (\textit{fst}\ \psi)$
$\langle \textit{proof} \rangle$

lemma *simplified-falsity*:

assumes *simp*: *simplified* ψ
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 *rtranclp-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* ψ)
shows $(\exists \psi'. (resolution^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')) \longleftrightarrow (\exists a\text{-}u\text{-}v.\ resolution^{**}\ \psi\ (\{\{\#\}\},\ a\text{-}u\text{-}v))$
(**is** $?A \longleftrightarrow ?B$)
$\langle proof \rangle$

lemma *resolution-soundness-and-completeness'*:
fixes ψ :: $'v$ ::*linorder state*
assumes
finite: *finite* (*fst* ψ)**and**
snd: *snd* $\psi = \{\}$
shows $(\exists a\text{-}u\text{-}v.\ (resolution^{**}\ \psi\ (\{\{\#\}\},\ a\text{-}u\text{-}v))) \longleftrightarrow unsatisfiable\ (fst\ \psi)$
$\langle proof \rangle$

end

14 Partial Clausal Logic

We here define decided literals (that will be used in both DPLL and CDCL) and the entailment corresponding to it.

theory *Partial-Annotated-Clausal-Logic*
imports *Partial-Clausal-Logic*

begin

14.1 Decided Literals

14.1.1 Definition

datatype ($'v$, $'lvl$, $'mark$) *ann-lit* =
is-decided: *Decided* (*lit-of*: $'v$ *literal*) (*level-of*: $'lvl$) |
is-proped: *Propagated* (*lit-of*: $'v$ *literal*) (*mark-of*: $'mark$)

lemma *ann-lit-list-induct*[*case-names nil decided proped*]:
assumes *P* [] **and**
$\bigwedge L\ l\ xs.\ P\ xs \Longrightarrow P\ (Decided\ L\ l\ \#\ xs)$ **and**
$\bigwedge L\ m\ xs.\ P\ xs \Longrightarrow P\ (Propagated\ L\ m\ \#\ xs)$

shows $P\ xs$
⟨*proof*⟩

lemma *is-decided-ex-Decided*:
$is\text{-}decided\ L \Longrightarrow (\bigwedge K\ lvl.\ L = Decided\ K\ lvl \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

type-synonym $('v, 'l, 'm)\ ann\text{-}lits = ('v, 'l, 'm)\ ann\text{-}lit\ list$

definition *lits-of* :: $('a, 'b, 'c)\ ann\text{-}lit\ set \Rightarrow 'a\ literal\ set$ **where**
$lits\text{-}of\ Ls = lit\text{-}of\ `\ Ls$

abbreviation *lits-of-l* :: $('a, 'b, 'c)\ ann\text{-}lit\ list \Rightarrow 'a\ literal\ set$ **where**
$lits\text{-}of\text{-}l\ Ls \equiv lits\text{-}of\ (set\ Ls)$

lemma *lits-of-l-empty*[*simp*]:
$lits\text{-}of\ \{\} = \{\}$
⟨*proof*⟩

lemma *lits-of-insert*[*simp*]:
$lits\text{-}of\ (insert\ L\ Ls) = insert\ (lit\text{-}of\ L)\ (lits\text{-}of\ Ls)$
⟨*proof*⟩

lemma *lits-of-l-Un*[*simp*]:
$lits\text{-}of\ (l \cup l') = lits\text{-}of\ l \cup lits\text{-}of\ l'$
⟨*proof*⟩

lemma *finite-lits-of-def*[*simp*]:
$finite\ (lits\text{-}of\text{-}l\ L)$
⟨*proof*⟩

abbreviation *unmark* **where**
$unmark \equiv (\lambda a.\ \{\#lit\text{-}of\ a\#\})$

abbreviation *unmark-s* **where**
$unmark\text{-}s\ M \equiv unmark\ `\ M$

abbreviation *unmark-l* **where**
$unmark\text{-}l\ M \equiv unmark\text{-}s\ (set\ M)$

lemma *atms-of-ms-lambda-lit-of-is-atm-of-lit-of*[*simp*]:
$atms\text{-}of\text{-}ms\ (unmark\text{-}l\ M') = atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M'$
⟨*proof*⟩

lemma *lits-of-l-empty-is-empty*[*iff*]:
$lits\text{-}of\text{-}l\ M = \{\} \longleftrightarrow M = []$
⟨*proof*⟩

14.1.2 Entailment

definition *true-annot* :: $('a, 'l, 'm)\ ann\text{-}lits \Rightarrow 'a\ clause \Rightarrow bool$ (**infix** $\models a\ 49$) **where**
$I \models a\ C \longleftrightarrow (lits\text{-}of\text{-}l\ I) \models C$

definition *true-annots* :: $('a, 'l, 'm)\ ann\text{-}lits \Rightarrow 'a\ clauses \Rightarrow bool$ (**infix** $\models as\ 49$) **where**
$I \models as\ CC \longleftrightarrow (\forall C \in CC.\ I \models a\ C)$

lemma *true-annot-empty-model*[*simp*]:
$\neg[] \models a\ \psi$
⟨*proof*⟩

lemma *true-annot-empty*[*simp*]:
$\neg I \models a\ \{\#\}$
⟨*proof*⟩

lemma *empty-true-annots-def*[*iff*]:
$[] \models as\ \psi \longleftrightarrow \psi = \{\}$
⟨*proof*⟩

lemma *true-annots-empty*[*simp*]:
$I \models as\ \{\}$
⟨*proof*⟩

lemma *true-annots-single-true-annot*[*iff*]:
$I \models as\ \{C\} \longleftrightarrow I \models a\ C$
⟨*proof*⟩

lemma *true-annot-insert-l*[*simp*]:
$M \models a\ A \Longrightarrow L\ \#\ M \models a\ A$
⟨*proof*⟩

lemma *true-annots-insert-l* [*simp*]:
$M \models as\ A \Longrightarrow L\ \#\ M \models as\ A$
⟨*proof*⟩

lemma *true-annots-union*[*iff*]:
$M \models as\ A \cup B \longleftrightarrow (M \models as\ A \wedge M \models as\ B)$
⟨*proof*⟩

lemma *true-annots-insert*[*iff*]:
$M \models as\ insert\ a\ A \longleftrightarrow (M \models a\ a \wedge M \models as\ A)$
⟨*proof*⟩

Link between $\models as$ and $\models s$:

lemma *true-annots-true-cls*:
$I \models as\ CC \longleftrightarrow lits\text{-}of\text{-}l\ I \models s\ CC$
⟨*proof*⟩

lemma *in-lit-of-true-annot*:
$a \in lits\text{-}of\text{-}l\ M \longleftrightarrow M \models a\ \{\#a\#\}$
⟨*proof*⟩

lemma *true-annot-lit-of-notin-skip*:
$L\ \#\ M \models a\ A \Longrightarrow lit\text{-}of\ L \notin\#\ A \Longrightarrow M \models a\ A$
⟨*proof*⟩

lemma *true-clss-singleton-lit-of-implies-incl*:
$I \models s\ unmark\text{-}l\ MLs \Longrightarrow lits\text{-}of\text{-}l\ MLs \subseteq I$
⟨*proof*⟩

lemma *true-annot-true-clss-cls*:

$MLs \models a\ \psi \Longrightarrow set\ (map\ unmark\ MLs) \models p\ \psi$
⟨*proof*⟩

lemma *true-annots-true-clss-cls*:
$MLs \models as\ \psi \Longrightarrow set\ (map\ unmark\ MLs) \models ps\ \psi$
⟨*proof*⟩

lemma *true-annots-decided-true-cls*[*iff*]:
$map\ (\lambda M.\ Decided\ M\ a)\ M \models as\ N \longleftrightarrow set\ M \models s\ N$
⟨*proof*⟩

lemma *true-annot-singleton*[*iff*]: $M \models a\ \{\#L\#\} \longleftrightarrow L \in lits\text{-}of\text{-}l\ M$
⟨*proof*⟩

lemma *true-annots-true-clss-clss*:
$A \models as\ \Psi \Longrightarrow unmark\text{-}l\ A \models ps\ \Psi$
⟨*proof*⟩

lemma *true-annot-commute*:
$M\ @\ M' \models a\ D \longleftrightarrow M'\ @\ M \models a\ D$
⟨*proof*⟩

lemma *true-annots-commute*:
$M\ @\ M' \models as\ D \longleftrightarrow M'\ @\ M \models as\ D$
⟨*proof*⟩

lemma *true-annot-mono*[*dest*]:
$set\ I \subseteq set\ I' \Longrightarrow I \models a\ N \Longrightarrow I' \models a\ N$
⟨*proof*⟩

lemma *true-annots-mono*:
$set\ I \subseteq set\ I' \Longrightarrow I \models as\ N \Longrightarrow I' \models as\ N$
⟨*proof*⟩

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 decided literals (aka a trail in most cases).

Remark that *undefined* already exists and is a completely different Isabelle function.

definition *defined-lit* :: $('a,\ 'l,\ 'm)\ ann\text{-}lits\ \Rightarrow\ 'a\ literal \Rightarrow bool$
where
$defined\text{-}lit\ I\ L \longleftrightarrow (\exists l.\ Decided\ L\ l \in set\ I) \lor (\exists P.\ Propagated\ L\ P \in set\ I)$
$\lor (\exists l.\ Decided\ (-L)\ l \in set\ I) \lor (\exists P.\ Propagated\ (-L)\ P \in set\ I)$

abbreviation *undefined-lit* :: $('a,\ 'l,\ 'm)\ ann\text{-}lit\ list\ \Rightarrow\ 'a\ literal \Rightarrow\ bool$
where $undefined\text{-}lit\ I\ L \equiv \neg defined\text{-}lit\ I\ L$

lemma *defined-lit-rev*[*simp*]:
$defined\text{-}lit\ (rev\ M)\ L \longleftrightarrow defined\text{-}lit\ M\ L$
⟨*proof*⟩

lemma *atm-imp-decided-or-proped*:
assumes $x \in set\ I$
shows
$(\exists l.\ Decided\ (-\ lit\text{-}of\ x)\ l \in set\ I)$

$\vee$ ($\exists\, l.$ *Decided* (*lit-of* x) $l \in$ *set* I)
$\vee$ ($\exists\, l.$ *Propagated* ($-$ *lit-of* x) $l \in$ *set* I)
$\vee$ ($\exists\, l.$ *Propagated* (*lit-of* x) $l \in$ *set* I)
⟨*proof*⟩

lemma *literal-is-lit-of-decided*:
assumes L = *lit-of* x
shows ($\exists\, l.$ x = *Decided* L l) $\vee$ ($\exists\, l'.$ x = *Propagated* L l')
⟨*proof*⟩

lemma *true-annot-iff-decided-or-true-lit*:
defined-lit I L $\longleftrightarrow$ (*lits-of-l* I $\models$*l* L $\vee$ *lits-of-l* I $\models$*l* $-L$)
⟨*proof*⟩

lemma *consistent-inter-true-annots-satisfiable*:
consistent-interp (*lits-of-l* I) $\Longrightarrow$ I $\models$*as* N $\Longrightarrow$ *satisfiable* N
⟨*proof*⟩

lemma *defined-lit-map*:
defined-lit *Ls* L $\longleftrightarrow$ *atm-of* $L \in$ ($\lambda l.$ *atm-of* (*lit-of* l)) ‘ *set* *Ls*
⟨*proof*⟩

lemma *defined-lit-uminus*[*iff*]:
defined-lit I ($-L$) $\longleftrightarrow$ *defined-lit* I L
⟨*proof*⟩

lemma *Decided-Propagated-in-iff-in-lits-of-l*:
defined-lit I L $\longleftrightarrow$ ($L \in$ *lits-of-l* I $\vee$ $-L \in$ *lits-of-l* I)
⟨*proof*⟩

lemma *consistent-add-undefined-lit-consistent*[*simp*]:
assumes
consistent-interp (*lits-of-l* *Ls*) **and**
undefined-lit *Ls* L
shows *consistent-interp* (*insert* L (*lits-of-l* *Ls*))
⟨*proof*⟩

lemma *decided-empty*[*simp*]:
$\neg$*defined-lit* [] L
⟨*proof*⟩

14.2 Backtracking

fun *backtrack-split* :: ($'v$, $'l$, $'m$) *ann-lits*
$\Rightarrow$ ($'v$, $'l$, $'m$) *ann-lits* $\times$ ($'v$, $'l$, $'m$) *ann-lits* **where**
backtrack-split [] = ([], []) |
backtrack-split (*Propagated* L P # *mlits*) = *apfst* ((*op* #) (*Propagated* L P)) (*backtrack-split* *mlits*) |
backtrack-split (*Decided* L l # *mlits*) = ([], *Decided* L l # *mlits*)

lemma *backtrack-split-fst-not-decided*: $a \in$ *set* (*fst* (*backtrack-split* l)) $\Longrightarrow$ $\neg$*is-decided* a
⟨*proof*⟩

lemma *backtrack-split-snd-hd-decided*:
snd (*backtrack-split* l) $\neq$ [] $\Longrightarrow$ *is-decided* (*hd* (*snd* (*backtrack-split* l)))
⟨*proof*⟩

lemma *backtrack-split-list-eq*[*simp*]:
$fst\ (backtrack\text{-}split\ l)\ @\ (snd\ (backtrack\text{-}split\ l)) = l$
⟨*proof*⟩

lemma *backtrack-snd-empty-not-decided*:
$backtrack\text{-}split\ M = (M'', []) \Longrightarrow \forall l{\in}set\ M.\ \neg\ is\text{-}decided\ l$
⟨*proof*⟩

lemma *backtrack-split-some-is-decided-then-snd-has-hd*:
$\exists l{\in}set\ M.\ is\text{-}decided\ l \Longrightarrow \exists M'\ L'\ M''.\ backtrack\text{-}split\ M = (M'', L'\ \#\ M')$
⟨*proof*⟩

Another characterisation of the result of *backtrack-split*. This view allows some simpler proofs, since *takeWhile* and *dropWhile* are highly automated:

lemma *backtrack-split-takeWhile-dropWhile*:
$backtrack\text{-}split\ M = (takeWhile\ (Not\ o\ is\text{-}decided)\ M,\ dropWhile\ (Not\ o\ is\text{-}decided)\ M)$
⟨*proof*⟩

14.3 Decomposition with respect to the First Decided Literals

In this section we define a function that returns a decomposition with the first decided literal. This function is useful to define the backtracking of DPLL.

14.3.1 Definition

The pattern *get-all-ann-decomposition* [] = [([], [])] is necessary otherwise, we can call the *hd* function in the other pattern.

fun *get-all-ann-decomposition* :: ('*a*, '*l*, '*m*) *ann-lits*
⇒ (('*a*, '*l*, '*m*) *ann-lits* × ('*a*, '*l*, '*m*) *ann-lits*) *list* **where**
get-all-ann-decomposition (*Decided L l* # *Ls*) =
(*Decided L l* # *Ls*, []) # *get-all-ann-decomposition Ls* |
get-all-ann-decomposition (*Propagated L P*# *Ls*) =
(*apsnd* ((*op* #) (*Propagated L P*)) (*hd* (*get-all-ann-decomposition Ls*)))
tl (*get-all-ann-decomposition Ls*) |
get-all-ann-decomposition [] = [([], [])]

value *get-all-ann-decomposition* [*Propagated A5 B5*, *Decided C4 D4*, *Propagated A3 B3*, *Propagated A2 B2*, *Decided C1 D1*, *Propagated A0 B0*]

Now we can prove several simple properties about the function.

lemma *get-all-ann-decomposition-never-empty*[*iff*]:
$get\text{-}all\text{-}ann\text{-}decomposition\ M = [] \longleftrightarrow False$
⟨*proof*⟩

lemma *get-all-ann-decomposition-never-empty-sym*[*iff*]:
$[] = get\text{-}all\text{-}ann\text{-}decomposition\ M \longleftrightarrow False$
⟨*proof*⟩

lemma *get-all-ann-decomposition-decomp*:
$hd\ (get\text{-}all\text{-}ann\text{-}decomposition\ S) = (a, c) \Longrightarrow S = c\ @\ a$
⟨*proof*⟩

lemma *get-all-ann-decomposition-backtrack-split*:

backtrack-split $S = (M, M') \longleftrightarrow$ *hd* (*get-all-ann-decomposition* S) $= (M', M)$
⟨*proof*⟩

lemma *get-all-ann-decomposition-nil-backtrack-split-snd-nil*:
get-all-ann-decomposition $S = [([], A)] \Longrightarrow$ *snd* (*backtrack-split* S) $= []$
⟨*proof*⟩

This functions says that the first element is either empty or starts with a decided element of the list.

lemma *get-all-ann-decomposition-length-1-fst-empty-or-length-1*:
assumes *get-all-ann-decomposition* $M = (a, b)$ # []
shows $a = [] \lor$ (*length* $a = 1 \land$ *is-decided* (*hd* a) $\land$ *hd* $a \in$ *set* M)
⟨*proof*⟩

lemma *get-all-ann-decomposition-fst-empty-or-hd-in-M*:
assumes *get-all-ann-decomposition* $M = (a, b)$ # l
shows $a = [] \lor$ (*is-decided* (*hd* a) $\land$ *hd* $a \in$ *set* M)
⟨*proof*⟩

lemma *get-all-ann-decomposition-snd-not-decided*:
assumes $(a, b) \in$ *set* (*get-all-ann-decomposition* M)
and $L \in$ *set* b
shows $\neg$*is-decided* L
⟨*proof*⟩

lemma *tl-get-all-ann-decomposition-skip-some*:
assumes $x \in$ *set* (*tl* (*get-all-ann-decomposition* $M1$))
shows $x \in$ *set* (*tl* (*get-all-ann-decomposition* ($M0$ @ $M1$)))
⟨*proof*⟩

lemma *hd-get-all-ann-decomposition-skip-some*:
assumes $(x, y) =$ *hd* (*get-all-ann-decomposition* $M1$)
shows $(x, y) \in$ *set* (*get-all-ann-decomposition* ($M0$ @ *Decided* K i # $M1$))
⟨*proof*⟩

lemma *in-get-all-ann-decomposition-in-get-all-ann-decomposition-prepend*:
$(a, b) \in$ *set* (*get-all-ann-decomposition* M') $\Longrightarrow$
$\exists b'$. $(a, b'$ @ $b) \in$ *set* (*get-all-ann-decomposition* (M @ M'))
⟨*proof*⟩

lemma *get-all-ann-decomposition-remove-undecided-length*:
assumes $\forall l \in$ *set* M'. $\neg$*is-decided* l
shows *length* (*get-all-ann-decomposition* (M' @ M''))
$=$ *length* (*get-all-ann-decomposition* M'')
⟨*proof*⟩

lemma *get-all-ann-decomposition-not-is-decided-length*:
assumes $\forall l \in$ *set* M'. $\neg$*is-decided* l
shows $1 +$ *length* (*get-all-ann-decomposition* (*Propagated* $(-L)$ P # M))
$=$ *length* (*get-all-ann-decomposition* (M' @ *Decided* L l # M))
⟨*proof*⟩

lemma *get-all-ann-decomposition-last-choice*:
assumes *tl* (*get-all-ann-decomposition* (M' @ *Decided* L l # M)) $\neq []$
and $\forall l \in$ *set* M'. $\neg$*is-decided* l

and $hd\ (tl\ (\textit{get-all-ann-decomposition}\ (M'\ @\ \textit{Decided}\ L\ l\ \#\ M))) = (M0', M0)$
shows $hd\ (\textit{get-all-ann-decomposition}\ (\textit{Propagated}\ (-L)\ P\ \#\ M)) = (M0', \textit{Propagated}\ (-L)\ P\ \#\ M0)$
⟨*proof*⟩

lemma *get-all-ann-decomposition-except-last-choice-equal*:
assumes $\forall\, l \in \textit{set}\ M'.\ \neg\textit{is-decided}\ l$
shows $tl\ (\textit{get-all-ann-decomposition}\ (\textit{Propagated}\ (-L)\ P\ \#\ M))$
$= tl\ (tl\ (\textit{get-all-ann-decomposition}\ (M'\ @\ \textit{Decided}\ L\ l\ \#\ M)))$
⟨*proof*⟩

lemma *get-all-ann-decomposition-hd-hd*:
assumes $\textit{get-all-ann-decomposition}\ Ls = (M, C)\ \#\ (M0, M0')\ \#\ l$
shows $tl\ M = M0'\ @\ M0 \wedge \textit{is-decided}\ (hd\ M)$
⟨*proof*⟩

lemma *get-all-ann-decomposition-exists-prepend*[*dest*]:
assumes $(a, b) \in \textit{set}\ (\textit{get-all-ann-decomposition}\ M)$
shows $\exists\, c.\ M = c\ @\ b\ @\ a$
⟨*proof*⟩

lemma *get-all-ann-decomposition-incl*:
assumes $(a, b) \in \textit{set}\ (\textit{get-all-ann-decomposition}\ M)$
shows $\textit{set}\ b \subseteq \textit{set}\ M$ **and** $\textit{set}\ a \subseteq \textit{set}\ M$
⟨*proof*⟩

lemma *get-all-ann-decomposition-exists-prepend'*:
assumes $(a, b) \in \textit{set}\ (\textit{get-all-ann-decomposition}\ M)$
obtains c **where** $M = c\ @\ b\ @\ a$
⟨*proof*⟩

lemma *union-in-get-all-ann-decomposition-is-subset*:
assumes $(a, b) \in \textit{set}\ (\textit{get-all-ann-decomposition}\ M)$
shows $\textit{set}\ a \cup \textit{set}\ b \subseteq \textit{set}\ M$
⟨*proof*⟩

lemma *Decided-cons-in-get-all-ann-decomposition-append-Decided-cons*:
$\exists\, M1\ M2.\ (\textit{Decided}\ K\ i\ \#\ M1, M2) \in \textit{set}\ (\textit{get-all-ann-decomposition}\ (c\ @\ \textit{Decided}\ K\ i\ \#\ c'))$
⟨*proof*⟩

14.3.2 Entailment of the Propagated by the Decided Literal

lemma *get-all-ann-decomposition-snd-union*:
$\textit{set}\ M = \bigcup(\textit{set}\ `\ \textit{snd}\ `\ \textit{set}\ (\textit{get-all-ann-decomposition}\ M)) \cup \{L\ |L.\ \textit{is-decided}\ L \wedge L \in \textit{set}\ M\}$
(**is** $?M\ M = ?U\ M \cup ?Ls\ M$)
⟨*proof*⟩

definition *all-decomposition-implies* :: $'a\ \textit{literal multiset set}$
$\Rightarrow (('a, 'l, 'm)\ \textit{ann-lit list} \times ('a, 'l, 'm)\ \textit{ann-lit list})\ \textit{list} \Rightarrow \textit{bool}$ **where**
$\textit{all-decomposition-implies}\ N\ S \longleftrightarrow (\forall\, (Ls, \textit{seen}) \in \textit{set}\ S.\ \textit{unmark-l}\ Ls \cup N \models\!ps\ \textit{unmark-l seen})$

lemma *all-decomposition-implies-empty*[*iff*]:
$\textit{all-decomposition-implies}\ N\ []$ ⟨*proof*⟩

lemma *all-decomposition-implies-single*[*iff*]:
$\textit{all-decomposition-implies}\ N\ [(Ls, \textit{seen})] \longleftrightarrow \textit{unmark-l}\ Ls \cup N \models\!ps\ \textit{unmark-l seen}$
⟨*proof*⟩

lemma *all-decomposition-implies-append*[*iff*]:
$\textit{all-decomposition-implies } N\ (S\ @\ S')$
$\longleftrightarrow (\textit{all-decomposition-implies } N\ S \land \textit{all-decomposition-implies } N\ S')$
⟨*proof*⟩

lemma *all-decomposition-implies-cons-pair*[*iff*]:
$\textit{all-decomposition-implies } N\ ((Ls,\ seen)\ \#\ S')$
$\longleftrightarrow (\textit{all-decomposition-implies } N\ [(Ls,\ seen)] \land \textit{all-decomposition-implies } N\ S')$
⟨*proof*⟩

lemma *all-decomposition-implies-cons-single*[*iff*]:
$\textit{all-decomposition-implies } N\ (l\ \#\ S') \longleftrightarrow$
$(\textit{unmark-l } (\textit{fst } l) \cup N \models ps\ \textit{unmark-l } (\textit{snd } l) \land$
$\textit{all-decomposition-implies } N\ S')$
⟨*proof*⟩

lemma *all-decomposition-implies-trail-is-implied*:
assumes $\textit{all-decomposition-implies } N\ (\textit{get-all-ann-decomposition } M)$
shows $N \cup \{\textit{unmark } L\ |L.\ \textit{is-decided } L \land L \in \textit{set } M\}$
$\models ps\ \textit{unmark}\ `\ \bigcup(\textit{set}\ `\ \textit{snd}\ `\ \textit{set } (\textit{get-all-ann-decomposition } M))$
⟨*proof*⟩

lemma *all-decomposition-implies-propagated-lits-are-implied*:
assumes $\textit{all-decomposition-implies } N\ (\textit{get-all-ann-decomposition } M)$
shows $N \cup \{\textit{unmark } L\ |L.\ \textit{is-decided } L \land L \in \textit{set } M\} \models ps\ \textit{unmark-l } M$
(**is** $?I \models ps\ ?A$)
⟨*proof*⟩

lemma *all-decomposition-implies-insert-single*:
$\textit{all-decomposition-implies } N\ M \Longrightarrow \textit{all-decomposition-implies } (\textit{insert } C\ N)\ M$
⟨*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 $\textit{CNot} :: 'v\ \textit{clause} \Rightarrow 'v\ \textit{clauses}$ **where**
$\textit{CNot } \psi = \{\ \{\#-L\#\}\ |\ L.\ \ L \in\#\ \psi\ \}$

lemma *in-CNot-uminus*[*iff*]:
shows $\{\#L\#\} \in \textit{CNot } \psi \longleftrightarrow -L \in\#\ \psi$
⟨*proof*⟩

lemma
shows
CNot-singleton[*simp*]: $\textit{CNot } \{\#L\#\} = \{\{\#-L\#\}\}$ **and**
CNot-empty[*simp*]: $\textit{CNot } \{\#\} = \{\}$ **and**
CNot-plus[*simp*]: $\textit{CNot } (A + B) = \textit{CNot } A \cup \textit{CNot } B$
⟨*proof*⟩

lemma *CNot-eq-empty*[*iff*]:
$\textit{CNot } D = \{\} \longleftrightarrow D = \{\#\}$
⟨*proof*⟩

lemma *in-CNot-implies-uminus*:
assumes $L \in\# D$ **and** $M \models as\ CNot\ D$
shows $M \models a\ \{\#-L\#\}$ **and** $-L \in lits\text{-}of\text{-}l\ M$
⟨*proof*⟩

lemma *CNot-remdups-mset*[*simp*]:
$CNot\ (remdups\text{-}mset\ A) = CNot\ A$
⟨*proof*⟩

lemma *Ball-CNot-Ball-mset*[*simp*]:
$(\forall x{\in}CNot\ D.\ P\ x) \longleftrightarrow (\forall L{\in}\#\ D.\ P\ \{\#-L\#\})$
⟨*proof*⟩

lemma *consistent-CNot-not*:
assumes *consistent-interp I*
shows $I \models s\ CNot\ \varphi \Longrightarrow \neg I \models \varphi$
⟨*proof*⟩

lemma *total-not-true-cls-true-clss-CNot*:
assumes $total\text{-}over\text{-}m\ I\ \{\varphi\}$ **and** $\neg I \models \varphi$
shows $I \models s\ CNot\ \varphi$
⟨*proof*⟩

lemma *total-not-CNot*:
assumes $total\text{-}over\text{-}m\ I\ \{\varphi\}$ **and** $\neg I \models s\ CNot\ \varphi$
shows $I \models \varphi$
⟨*proof*⟩

lemma *atms-of-ms-CNot-atms-of*[*simp*]:
$atms\text{-}of\text{-}ms\ (CNot\ C) = atms\text{-}of\ C$
⟨*proof*⟩

lemma *true-clss-clss-contradiction-true-clss-cls-false*:
$C \in D \Longrightarrow D \models ps\ CNot\ C \Longrightarrow D \models p\ \{\#\}$
⟨*proof*⟩

lemma *true-annots-CNot-all-atms-defined*:
assumes $M \models as\ CNot\ T$ **and** $a1\colon\ L \in\# T$
shows $atm\text{-}of\ L \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
⟨*proof*⟩

lemma *true-annots-CNot-all-uminus-atms-defined*:
assumes $M \models as\ CNot\ T$ **and** $a1\colon -L \in\# T$
shows $atm\text{-}of\ L \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
⟨*proof*⟩

lemma *true-clss-clss-false-left-right*:
assumes $\{\{\#L\#\}\} \cup B \models p\ \{\#\}$
shows $B \models ps\ CNot\ \{\#L\#\}$
⟨*proof*⟩

lemma *true-annots-true-cls-def-iff-negation-in-model*:
$M \models as\ CNot\ C \longleftrightarrow (\forall L \in\# C.\ -L \in lits\text{-}of\text{-}l\ M)$
⟨*proof*⟩

lemma *true-annot-CNot-diff*:
$I \models as\ CNot\ C \Longrightarrow I \models as\ CNot\ (C - C')$
⟨*proof*⟩

lemma *CNot-mset-replicate*[*simp*]:
$CNot\ (mset\ (replicate\ n\ L)) = (if\ n = 0\ then\ \{\}\ else\ \{\{\#-L\#\}\})$
⟨*proof*⟩

lemma *consistent-CNot-not-tautology*:
$consistent\text{-}interp\ M \Longrightarrow M \models s\ CNot\ D \Longrightarrow \neg tautology\ D$
⟨*proof*⟩

lemma *atms-of-ms-CNot-atms-of-ms*: $atms\text{-}of\text{-}ms\ (CNot\ CC) = atms\text{-}of\text{-}ms\ \{CC\}$
⟨*proof*⟩

lemma *total-over-m-CNot-toal-over-m*[*simp*]:
$total\text{-}over\text{-}m\ I\ (CNot\ C) = total\text{-}over\text{-}set\ I\ (atms\text{-}of\ C)$
⟨*proof*⟩

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*: $-(a::'a\ literal) = i \longleftrightarrow a = -i$
⟨*proof*⟩

lemma *true-clss-cls-plus-CNot*:
assumes
CC-L: $A \models p\ CC + \{\#L\#\}$ **and**
CNot-CC: $A \models ps\ CNot\ CC$
shows $A \models p\ \{\#L\#\}$
⟨*proof*⟩

lemma *true-annots-CNot-lit-of-notin-skip*:
assumes *LM*: $L\ \#\ M \models as\ CNot\ A$ **and** *LA*: $lit\text{-}of\ L \notin\#\ A\ -lit\text{-}of\ L \notin\#\ A$
shows $M \models as\ CNot\ A$
⟨*proof*⟩

lemma *true-clss-clss-union-false-true-clss-clss-cnot*:
$A \cup \{B\} \models ps\ \{\{\#\}\} \longleftrightarrow A \models ps\ CNot\ B$
⟨*proof*⟩

lemma *true-annot-remove-hd-if-notin-vars*:
assumes $a\ \#\ M' \models a\ D$ **and** $atm\text{-}of\ (lit\text{-}of\ a) \notin atms\text{-}of\ D$
shows $M' \models a\ D$
⟨*proof*⟩

lemma *true-annot-remove-if-notin-vars*:
assumes $M\ @\ M' \models a\ D$ **and** $\forall x \in atms\text{-}of\ D.\ x \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
shows $M' \models a\ D$
⟨*proof*⟩

lemma *true-annots-remove-if-notin-vars*:
assumes $M\ @\ M' \models as\ D$ **and** $\forall x \in atms\text{-}of\text{-}ms\ D.\ x \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
shows $M' \models as\ D$ ⟨*proof*⟩

lemma *all-variables-defined-not-imply-cnot*:
assumes
$\forall s \in$ *atms-of-ms* $\{B\}$. $s \in$ *atm-of* ‘ *lits-of-l* A **and**
$\neg$ $A \models a$ B
shows $A \models as$ *CNot* B
⟨*proof*⟩

lemma *CNot-union-mset*[*simp*]:
CNot $(A \#\cup B)$ = *CNot* $A \cup$ *CNot* B
⟨*proof*⟩

14.5 Other

abbreviation *no-dup* $L \equiv$ *distinct* (*map* (λl. *atm-of* (*lit-of* l)) L)

lemma *no-dup-rev*[*simp*]:
no-dup (*rev* M) $\longleftrightarrow$ *no-dup* M
⟨*proof*⟩

lemma *no-dup-length-eq-card-atm-of-lits-of-l*:
assumes *no-dup* M
shows *length* M = *card* (*atm-of* ‘ *lits-of-l* M)
⟨*proof*⟩

lemma *distinct-consistent-interp*:
no-dup $M \Longrightarrow$ *consistent-interp* (*lits-of-l* M)
⟨*proof*⟩

lemma *distinct-get-all-ann-decomposition-no-dup*:
assumes $(a, b) \in$ *set* (*get-all-ann-decomposition* M)
and *no-dup* M
shows *no-dup* $(a\ @\ b)$
⟨*proof*⟩

lemma *true-annots-lit-of-notin-skip*:
assumes $L \# M \models as$ *CNot* A
and $-$*lit-of* $L \notin\# A$
and *no-dup* $(L \# M)$
shows $M \models as$ *CNot* A
⟨*proof*⟩

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**
$I \models asm\ C \equiv I \models as$ (*set-mset* C)

abbreviation *true-clss-clss-m*:: $'v$ *clause multiset* $\Rightarrow$ $'v$ *clause multiset* $\Rightarrow$ *bool* (**infix** $\models psm$ *50*)
where
$I \models psm\ C \equiv$ *set-mset* $I \models ps$ (*set-mset* C)

Analog of $[\![?N \models ps\ ?B;\ ?A \subseteq ?B]\!] \Longrightarrow ?N \models ps\ ?A$

lemma *true-clss-clssm-subsetE*: $N \models psm\ B \Longrightarrow A \subseteq\# B \Longrightarrow N \models psm\ A$

⟨*proof*⟩

abbreviation *true-clss-cls-m*:: *'a clause multiset* ⇒ *'a clause* ⇒ *bool* (**infix** ⊨*pm 50*) **where**
I ⊨*pm C* ≡ *set-mset I* ⊨*p C*

abbreviation *distinct-mset-mset* :: *'a multiset multiset* ⇒ *bool* **where**
distinct-mset-mset Σ ≡ *distinct-mset-set* (*set-mset* Σ)

abbreviation *all-decomposition-implies-m* **where**
all-decomposition-implies-m A B ≡ *all-decomposition-implies* (*set-mset A*) *B*

abbreviation *atms-of-mm* :: *'a literal multiset multiset* ⇒ *'a set* **where**
atms-of-mm U ≡ *atms-of-ms* (*set-mset U*)

Other definition using *Union-mset*

lemma *atms-of-mm U* ≡ *set-mset* (⋃# *image-mset* (*image-mset atm-of*) *U*)
⟨*proof*⟩

abbreviation *true-clss-m*:: *'a interp* ⇒ *'a clause multiset* ⇒ *bool* (**infix** ⊨*sm 50*) **where**
I ⊨*sm C* ≡ *I* ⊨*s set-mset C*

abbreviation *true-clss-ext-m* (**infix** ⊨*sextm 49*) **where**
I ⊨*sextm C* ≡ *I* ⊨*sext 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
mset-cls :: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls*
assumes
insert-cls[*simp*]: *mset-cls* (*insert-cls L C*) = *mset-cls C* + {#*L*#} **and**
remove-lit[*simp*]: *mset-cls* (*remove-lit L C*) = *remove1-mset L* (*mset-cls C*)
begin
end

locale *raw-ccls-union* =
fixes

mset-cls :: *'cls* ⇒ *'v clause* **and**
union-cls :: *'cls* ⇒ *'cls* ⇒ *'cls* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls*
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* #∪ *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*
⟨*proof*⟩

interpretation *cls-cls*: *raw-cls id*
λL C. C + {#*L*#} *remove1-mset*
⟨*proof*⟩

interpretation *list-cls*: *raw-ccls-union mset*
union-mset-list
op # *remove1*
⟨*proof*⟩

interpretation *cls-cls*: *raw-ccls-union id*
op #∪ *λL C. C* + {#*L*#} *remove1-mset*
⟨*proof*⟩
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* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* +
fixes
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*
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 $\Longrightarrow$ *mset-cls* a ∈# *mset-clss* C **and**
in-mset-clss-exists-preimage: b ∈# *mset-clss* C $\Longrightarrow$ $\exists\, b'$. *in-clss* b' C $\wedge$ *mset-cls* b' = b **and**
remove-from-clss-mset-clss[*simp*]:
mset-clss (*remove-from-clss* a C) = *mset-clss* C − {#*mset-cls* a#} **and**
in-clss-union-clss[*simp*]:
in-clss a (*union-clss* C D) $\longleftrightarrow$ *in-clss* a C $\vee$ *in-clss* a D
begin

end

experiment
begin
fun *remove-first* **where**
remove-first - [] = [] |
remove-first C (C' # L) = (*if mset* C = *mset* C' *then* L *else* C' # *remove-first* C L)

lemma *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* ∈# $\lambda L\ C.\ C$ + {#L#} *remove1-mset*
⟨*proof*⟩

interpretation *list-clss*: *raw-clss mset*
op # *remove1* λL. *mset* (*map mset* L) *op* @ $\lambda L\ C.\ L \in$ *set* C *op* #
remove-first
⟨*proof*⟩
end

end
theory *CDCL-WNOT-Measure*
imports *Main List-More*
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 μ_C :: *nat* ⇒ *nat* ⇒ *nat list* ⇒ *nat* **where**
μ_C s b M ≡ ($\sum$ *i=0..<length* M. $M!i * b$^ (s +i − *length* M))

lemma μ_C-*nil*[*simp*]:
μ_C s b [] = 0
⟨*proof*⟩

lemma μ_C-*single*[*simp*]:
μ_C s b [L] = $L * b$ ^ (s − *Suc 0*)

⟨proof⟩

lemma *set-sum-atLeastLessThan-add*:
$(\sum i{=}k..{<}k{+}(b{::}nat).\ f\ i) = (\sum i{=}0..{<}b.\ f\ (k{+}\ i))$
⟨proof⟩

lemma *set-sum-atLeastLessThan-Suc*:
$(\sum i{=}1..{<}Suc\ j.\ f\ i) = (\sum i{=}0..{<}j.\ f\ (Suc\ i))$
⟨proof⟩

lemma μ_C-*cons*:
$\mu_C\ s\ b\ (L\ \#\ M) = L * b\ \hat{}\ (s - 1 - length\ M) + \mu_C\ s\ b\ M$
⟨proof⟩

lemma μ_C-*append*:
assumes $s \geq length\ (M @ M')$
shows $\mu_C\ s\ b\ (M @ M') = \mu_C\ (s - length\ M')\ b\ M + \mu_C\ s\ b\ M'$
⟨proof⟩

lemma μ_C-*cons-non-empty-inf*:
assumes *M-ge-1*: $\forall i \in set\ M.\ i \geq 1$ **and** *M*: $M \neq []$
shows $\mu_C\ s\ b\ M \geq b\ \hat{}\ (s - length\ M)$
⟨proof⟩

Copy of `~~/src/HOL/ex/NatSum.thy` (but generalized to $0 \leq k$)

lemma *sum-of-powers*: $0 \leq k \Longrightarrow (k - 1) * (\sum i{=}0..{<}n.\ k\hat{}i) = k\hat{}n - (1{::}nat)$
⟨proof⟩

In the degenerated cases, we only have the large inequality holds. In the other cases, the following strict inequality holds:

lemma μ_C-*bounded-non-degenerated*:
fixes b ::*nat*
assumes
$b > 0$ **and**
$M \neq []$ **and**
M-le: $\forall i < length\ M.\ M!i < b$ **and**
$s \geq length\ M$
shows $\mu_C\ s\ b\ M < b\hat{}s$
⟨proof⟩

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-le: $\forall i < length\ M.\ M!i < b$ **and**
$s \geq length\ M$
$b > 0$
shows $\mu_C\ s\ b\ M < b\ \hat{}\ s$
⟨proof⟩

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 μ_C *s 0 M* $\leq$ *M!0*
⟨*proof*⟩

lemma *finite-bounded-pair-list*:
fixes *b* :: *nat*
shows *finite* $\{(ys, xs).$ *length xs* $< s \wedge$ *length ys* $< s \wedge$
$(\forall i<$ *length xs. xs* ! $i < b) \wedge (\forall i<$ *length ys. ys* ! $i < b)\}$
⟨*proof*⟩

definition νNOT :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ (*nat list* $\times$ *nat list*) *set* **where**
νNOT *s base* $= \{(ys, xs).$ *length xs* $< s \wedge$ *length ys* $< s \wedge$
$(\forall i<$ *length xs. xs* ! $i <$ *base*$) \wedge (\forall i<$ *length ys. ys* ! $i <$ *base*$) \wedge$
$(ys, xs) \in$ *lenlex less-than*$\}$

lemma *finite-νNOT*[*simp*]:
finite (νNOT *s base*)
⟨*proof*⟩

lemma *acyclic-νNOT*: *acyclic* (νNOT *s base*)
⟨*proof*⟩

lemma *wf-νNOT*: *wf* (νNOT *s base*)
⟨*proof*⟩

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 $\Longrightarrow -$ *lit-of xa* $=$ *lit-of x* $\Longrightarrow x \in$ *set M* $\Longrightarrow xa \notin$ *set M*
⟨*proof*⟩

lemma *atms-of-ms-single-atm-of*[*simp*]:
atms-of-ms $\{$*unmark L* $|L.$ *P L*$\}$ $=$ *atm-of* ‘ $\{$*lit-of L* $|L.$ *P L*$\}$
⟨*proof*⟩

lemma *atms-of-uminus-lit-atm-of-lit-of*:
atms-of $\{\#\ -$*lit-of x. x* $\in\#$ *A*$\#\}$ $=$ *atm-of* ‘ (*lit-of* ‘ (*set-mset A*))
⟨*proof*⟩

lemma *atms-of-ms-single-image-atm-of-lit-of*:
atms-of-ms (*unmark-s A*) $=$ *atm-of* ‘ (*lit-of* ‘ *A*)
⟨*proof*⟩

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-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
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* +
fixes
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st*
begin

notation *insert-cls* (**infix** !++ *50*)

notation *in-clss* (**infix** !∈! *50*)
notation *union-clss* (**infix** ⊕ *50*)
notation *insert-clss* (**infix** !++! *50*)

abbreviation $clauses_{NOT}$ **where**
$clauses_{NOT}$ *S* ≡ *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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ — related to the 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**
trail :: *'st* ⇒ (*'v*, *unit*, *unit*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**

$tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT} :: 'cls \Rightarrow 'st \Rightarrow 'st$ +
assumes
trail-prepend-trail[*simp*]:
$\bigwedge st\ L.\ undefined\text{-}lit\ (trail\ st)\ (lit\text{-}of\ L) \Longrightarrow trail\ (prepend\text{-}trail\ L\ st) = L\ \#\ trail\ st$
and
tl-trail[*simp*]: $trail\ (tl\text{-}trail\ S) = tl\ (trail\ S)$ **and**
trail-add-cls$_{NOT}$[*simp*]: $\bigwedge st\ C.\ no\text{-}dup\ (trail\ st) \Longrightarrow trail\ (add\text{-}cls_{NOT}\ C\ st) = trail\ st$ **and**
trail-remove-cls$_{NOT}$[*simp*]: $\bigwedge st\ C.\ trail\ (remove\text{-}cls_{NOT}\ C\ st) = trail\ st$ **and**

clauses-prepend-trail[*simp*]:
$\bigwedge st\ L.\ undefined\text{-}lit\ (trail\ st)\ (lit\text{-}of\ L) \Longrightarrow$
$clauses_{NOT}\ (prepend\text{-}trail\ L\ st) = clauses_{NOT}\ st$
and
clauses-tl-trail[*simp*]: $\bigwedge st.\ clauses_{NOT}\ (tl\text{-}trail\ st) = clauses_{NOT}\ st$ **and**
clauses-add-cls$_{NOT}$[*simp*]:
$\bigwedge st\ C.\ no\text{-}dup\ (trail\ st) \Longrightarrow clauses_{NOT}\ (add\text{-}cls_{NOT}\ C\ st) = \{\#mset\text{-}cls\ C\#\} + clauses_{NOT}\ st$
and
clauses-remove-cls$_{NOT}$[*simp*]:
$\bigwedge st\ C.\ clauses_{NOT}\ (remove\text{-}cls_{NOT}\ C\ st) = removeAll\text{-}mset\ (mset\text{-}cls\ C)\ (clauses_{NOT}\ st)$
begin

We define the following function doing the backtrack in the trail:

function $reduce\text{-}trail\text{-}to_{NOT} :: 'a\ list \Rightarrow 'st \Rightarrow 'st$ **where**
$reduce\text{-}trail\text{-}to_{NOT}\ F\ S =$
$(if\ length\ (trail\ S) = length\ F \lor trail\ S = []\ then\ S\ else\ reduce\text{-}trail\text{-}to_{NOT}\ F\ (tl\text{-}trail\ S))$
⟨*proof*⟩
termination ⟨*proof*⟩
declare *reduce-trail-to*$_{NOT}$*.simps*[*simp del*]

Then we need several lemmas about the *reduce-trail-to*$_{NOT}$.

lemma
shows
reduce-trail-to$_{NOT}$*-nil*[*simp*]: $trail\ S = [] \Longrightarrow reduce\text{-}trail\text{-}to_{NOT}\ F\ S = S$ **and**
reduce-trail-to$_{NOT}$*-eq-length*[*simp*]: $length\ (trail\ S) = length\ F \Longrightarrow reduce\text{-}trail\text{-}to_{NOT}\ F\ S = S$
⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$*-length-ne*[*simp*]:
$length\ (trail\ S) \neq length\ F \Longrightarrow trail\ S \neq [] \Longrightarrow$
$reduce\text{-}trail\text{-}to_{NOT}\ F\ S = reduce\text{-}trail\text{-}to_{NOT}\ F\ (tl\text{-}trail\ S)$
⟨*proof*⟩

lemma *trail-reduce-trail-to*$_{NOT}$*-length-le*:
assumes $length\ F > length\ (trail\ S)$
shows $trail\ (reduce\text{-}trail\text{-}to_{NOT}\ F\ S) = []$
⟨*proof*⟩

lemma *trail-reduce-trail-to*$_{NOT}$*-nil*[*simp*]:
$trail\ (reduce\text{-}trail\text{-}to_{NOT}\ []\ S) = []$
⟨*proof*⟩

lemma *clauses-reduce-trail-to*$_{NOT}$*-nil*:
$clauses_{NOT}\ (reduce\text{-}trail\text{-}to_{NOT}\ []\ S) = clauses_{NOT}\ S$
⟨*proof*⟩

lemma *trail-reduce-trail-to*$_{NOT}$*-drop*:
  trail (*reduce-trail-to*$_{NOT}$ *F S*) =
    (*if length* (*trail S*) $\geq$ *length F*
    then drop (*length* (*trail S*) $-$ *length F*) (*trail S*)
    else [])
  $\langle$*proof*$\rangle$

lemma *reduce-trail-to*$_{NOT}$*-skip-beginning*:
  assumes *trail S* = *F'* @ *F*
  shows *trail* (*reduce-trail-to*$_{NOT}$ *F S*) = *F*
  $\langle$*proof*$\rangle$

lemma *reduce-trail-to*$_{NOT}$*-clauses*[*simp*]:
  clauses$_{NOT}$ (*reduce-trail-to*$_{NOT}$ *F S*) = *clauses*$_{NOT}$ *S*
  $\langle$*proof*$\rangle$

lemma *trail-eq-reduce-trail-to*$_{NOT}$*-eq*:
  trail S = *trail T* $\Longrightarrow$ *trail* (*reduce-trail-to*$_{NOT}$ *F S*) = *trail* (*reduce-trail-to*$_{NOT}$ *F T*)
  $\langle$*proof*$\rangle$

lemma *trail-reduce-trail-to*$_{NOT}$*-add-cls*$_{NOT}$[*simp*]:
  no-dup (*trail S*) $\Longrightarrow$
    trail (*reduce-trail-to*$_{NOT}$ *F* (*add-cls*$_{NOT}$ *C S*)) = *trail* (*reduce-trail-to*$_{NOT}$ *F S*)
  $\langle$*proof*$\rangle$

lemma *reduce-trail-to*$_{NOT}$*-trail-tl-trail-decomp*[*simp*]:
  trail S = *F'* @ *Decided K* () # *F* $\Longrightarrow$
     trail (*reduce-trail-to*$_{NOT}$ *F* (*tl-trail S*)) = *F*
  $\langle$*proof*$\rangle$

lemma *reduce-trail-to*$_{NOT}$*-length*:
  length M = *length M'* $\Longrightarrow$ *reduce-trail-to*$_{NOT}$ *M S* = *reduce-trail-to*$_{NOT}$ *M' S*
  $\langle$*proof*$\rangle$

abbreviation *trail-weight* **where**
trail-weight S $\equiv$ *map* ((λ*l. 1* + *length l*) *o snd*) (*get-all-ann-decomposition* (*trail S*))

As we are defining abstract states, the Isabelle equality about them is too strong: we want the weaker equivalence stating that two states are equal if they cannot be distinguished, i.e. given the getter *trail* and *clauses*$_{NOT}$ do not distinguish them.

definition *state-eq*$_{NOT}$ :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* (**infix** $\sim$ *50*) **where**
S $\sim$ *T* $\longleftrightarrow$ *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* $\Longrightarrow$ *T* $\sim$ *U* $\Longrightarrow$ *S* $\sim$ *U*
  $\langle$*proof*$\rangle$

lemma
  shows
    state-eq$_{NOT}$*-trail*: $S \sim T \Longrightarrow$ *trail* $S =$ *trail* T **and**
    state-eq$_{NOT}$*-clauses*: $S \sim T \Longrightarrow$ *clauses*$_{NOT}$ $S =$ *clauses*$_{NOT}$ T
  ⟨*proof*⟩

lemmas *state-simp*$_{NOT}$[*simp*] = *state-eq*$_{NOT}$*-trail* *state-eq*$_{NOT}$*-clauses*

lemma *reduce-trail-to*$_{NOT}$*-state-eq*$_{NOT}$*-compatible*:
  assumes *ST*: $S \sim T$
  shows *reduce-trail-to*$_{NOT}$ F $S \sim$ *reduce-trail-to*$_{NOT}$ F T
⟨*proof*⟩

end

16.2.2 Definition of the operation

Each possible is in its own locale.

locale *propagate-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-cls$_{NOT}$ *remove-cls*$_{NOT}$
  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*) *ann-lits* **and**
    raw-clauses :: $'st \Rightarrow 'clss$ **and**
    prepend-trail :: $('v,$ *unit*, *unit*) *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
    tl-trail :: $'st \Rightarrow 'st$ **and**
    add-cls$_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
    remove-cls$_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ +
  fixes
    propagate-cond :: $('v,$ *unit*, *unit*) *ann-lit* $\Rightarrow 'st \Rightarrow$ *bool*
begin
inductive *propagate*$_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow$ *bool* **where**
propagate$_{NOT}$[*intro*]: $C + \{\#L\#\} \in\#$ *clauses*$_{NOT}$ $S \Longrightarrow$ *trail* $S \models as$ *CNot* C
    $\Longrightarrow$ *undefined-lit* (*trail* S) L
    $\Longrightarrow$ *propagate-cond* (*Propagated* L ()) S
    $\Longrightarrow$ $T \sim$ *prepend-trail* (*Propagated* L ()) S
    $\Longrightarrow$ *propagate*$_{NOT}$ S T
inductive-cases *propagate*$_{NOT}E$[*elim*]: *propagate*$_{NOT}$ 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-cls$_{NOT}$ *remove-cls*$_{NOT}$

for
  $mset\text{-}cls$ :: $'cls \Rightarrow 'v$ $clause$ **and**
  $insert\text{-}cls$ :: $'v$ $literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
  $remove\text{-}lit$ :: $'v$ $literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
  $mset\text{-}clss$:: $'clss \Rightarrow 'v$ $clauses$ **and**
  $union\text{-}clss$ :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
  $in\text{-}clss$ :: $'cls \Rightarrow 'clss \Rightarrow bool$ **and**
  $insert\text{-}clss$ :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
  $remove\text{-}from\text{-}clss$ :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
  $trail$ :: $'st \Rightarrow ('v, unit, unit)$ $ann\text{-}lits$ **and**
  $raw\text{-}clauses$ :: $'st \Rightarrow 'clss$ **and**
  $prepend\text{-}trail$ :: $('v, unit, unit)$ $ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
  $tl\text{-}trail$ :: $'st \Rightarrow 'st$ **and**
  $add\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
  $remove\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$
begin
inductive $decide_{NOT}$ :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
$decide_{NOT}[intro]$: $undefined\text{-}lit$ $(trail$ $S)$ $L \Longrightarrow atm\text{-}of$ $L \in atms\text{-}of\text{-}mm$ $(clauses_{NOT}$ $S)$
  $\Longrightarrow T \sim prepend\text{-}trail$ $(Decided$ L $())$ S
  $\Longrightarrow decide_{NOT}$ S T

inductive-cases $decide_{NOT}E[elim]$: $decide_{NOT}$ S S'
end

locale $backjumping\text{-}ops$ =
  $dpll\text{-}state$ $mset\text{-}cls$ $insert\text{-}cls$ $remove\text{-}lit$
    $mset\text{-}clss$ $union\text{-}clss$ $in\text{-}clss$ $insert\text{-}clss$ $remove\text{-}from\text{-}clss$
    $trail$ $raw\text{-}clauses$ $prepend\text{-}trail$ $tl\text{-}trail$ $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
  for
  $mset\text{-}cls$ :: $'cls \Rightarrow 'v$ $clause$ **and**
  $insert\text{-}cls$ :: $'v$ $literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
  $remove\text{-}lit$ :: $'v$ $literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
  $mset\text{-}clss$:: $'clss \Rightarrow 'v$ $clauses$ **and**
  $union\text{-}clss$ :: $'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
  $in\text{-}clss$ :: $'cls \Rightarrow 'clss \Rightarrow bool$ **and**
  $insert\text{-}clss$ :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
  $remove\text{-}from\text{-}clss$ :: $'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
  $trail$ :: $'st \Rightarrow ('v, unit, unit)$ $ann\text{-}lits$ **and**
  $raw\text{-}clauses$ :: $'st \Rightarrow 'clss$ **and**
  $prepend\text{-}trail$ :: $('v, unit, unit)$ $ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
  $tl\text{-}trail$ :: $'st \Rightarrow 'st$ **and**
  $add\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
  $remove\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ +
  fixes
  $backjump\text{-}conds$ :: $'v$ $clause \Rightarrow 'v$ $clause \Rightarrow 'v$ $literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$
begin

inductive $backjump$ **where**
$trail$ $S = F'$ @ $Decided$ K $()\#$ F
  $\Longrightarrow T \sim prepend\text{-}trail$ $(Propagated$ L $())$ $(reduce\text{-}trail\text{-}to_{NOT}$ F $S)$
  $\Longrightarrow C \in\# clauses_{NOT}$ S
  $\Longrightarrow trail$ $S \models as$ $CNot$ C
  $\Longrightarrow undefined\text{-}lit$ F L
  $\Longrightarrow atm\text{-}of$ $L \in atms\text{-}of\text{-}mm$ $(clauses_{NOT}$ $S) \cup atm\text{-}of$ ` $(lits\text{-}of\text{-}l$ $(trail$ $S))$
  $\Longrightarrow clauses_{NOT}$ $S \models pm$ $C' + \{\#L\#\}$

$\Longrightarrow F \models as\ CNot\ C'$
$\Longrightarrow backjump\text{-}conds\ C\ C'\ L\ S\ T$
$\Longrightarrow backjump\ S\ T$
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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *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 :: $'st \Rightarrow ('v,\ unit,\ unit)\ ann\text{-}lits$ **and**
raw-clauses :: $'st \Rightarrow 'clss$ **and**
prepend-trail :: $('v,\ unit,\ unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
inv :: $'st \Rightarrow bool$ **and**
backjump-conds :: $'v\ clause \Rightarrow 'v\ clause \Rightarrow 'v\ literal \Rightarrow 'st \Rightarrow 'st \Rightarrow bool$ **and**
propagate-conds :: $('v,\ unit,\ unit)\ ann\text{-}lit \Rightarrow 'st \Rightarrow bool$ +
assumes
bj-can-jump:
$\bigwedge S\ C\ F'\ K\ F\ L.$
$inv\ S \Longrightarrow$
$no\text{-}dup\ (trail\ S) \Longrightarrow$
$trail\ S = F'\ @\ Decided\ K\ ()\ \#\ F \Longrightarrow$
$C \in\#\ clauses_{NOT}\ S \Longrightarrow$
$trail\ S \models as\ CNot\ C \Longrightarrow$
$undefined\text{-}lit\ F\ L \Longrightarrow$
$atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses_{NOT}\ S) \cup atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (F'\ @\ Decided\ K\ ()\ \#\ F)) \Longrightarrow$
$clauses_{NOT}\ S \models pm\ C' + \{\#L\#\} \Longrightarrow$
$F \models as\ CNot\ C' \Longrightarrow$
$\neg no\text{-}step\ backjump\ S$
begin

We cannot add a like condition $atms\text{-}of\ C' \subseteq atms\text{-}of\text{-}ms\ N$ 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-of* $L \in$ *atm-of* ‘ *lits-of-l* (F' @ *Decided* 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* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
bj-decide$_{NOT}$: *decide*$_{NOT}$ S S' ⟹ *dpll-bj* S S' |
bj-propagate$_{NOT}$: *propagate*$_{NOT}$ S S' ⟹ *dpll-bj* S S' |
bj-backjump: *backjump* S S' ⟹ *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
    ⋀L T. *undefined-lit* (*trail* S) L ⟹ *atm-of* $L \in$ *atms-of-mm* (*clauses*$_{NOT}$ S)
      ⟹ T ∼ *prepend-trail* (*Decided* L ()) S
      ⟹ P S T **and**
    ⋀C L T. C + {#L#} ∈# *clauses*$_{NOT}$ S ⟹ *trail* S ⊨*as CNot* C ⟹ *undefined-lit* (*trail* S) L
      ⟹ T ∼ *prepend-trail* (*Propagated* L ()) S
      ⟹ P S T **and**
    ⋀C F' K F L C' T. C ∈# *clauses*$_{NOT}$ S ⟹ F' @ *Decided* K () # F ⊨*as CNot* C
      ⟹ *trail* S = F' @ *Decided* K () # F
      ⟹ *undefined-lit* F L
      ⟹ *atm-of* $L \in$ *atms-of-mm* (*clauses*$_{NOT}$ S) ∪ *atm-of* ‘ (*lits-of-l* (F' @ *Decided* K () # F))
      ⟹ *clauses*$_{NOT}$ S ⊨*pm* C' + {#L#}
      ⟹ F ⊨*as CNot* C'
      ⟹ T ∼ *prepend-trail* (*Propagated* L ()) (*reduce-trail-to*$_{NOT}$ F S)
      ⟹ P S T
  shows P S T
  ⟨*proof*⟩

16.3.2 Basic properties

First, some better suited induction principle **lemma** *dpll-bj-clauses*:
  assumes *dpll-bj* S T **and** *inv* S
  shows *clauses*$_{NOT}$ S = *clauses*$_{NOT}$ T
  ⟨*proof*⟩

No duplicates in the trail **lemma** *dpll-bj-no-dup*:
  assumes *dpll-bj* S T **and** *inv* S
  and *no-dup* (*trail* S)
  shows *no-dup* (*trail* T)
  ⟨*proof*⟩

Valuations **lemma** *dpll-bj-sat-iff*:
  assumes *dpll-bj* S T **and** *inv* S
  shows I ⊨*sm clauses*$_{NOT}$ S ⟷ I ⊨*sm clauses*$_{NOT}$ T
  ⟨*proof*⟩

Clauses **lemma** *dpll-bj-atms-of-ms-clauses-inv*:
assumes
dpll-bj S T **and**
inv S
shows *atms-of-mm* (*clauses*$_{NOT}$ S) = *atms-of-mm* (*clauses*$_{NOT}$ T)
⟨*proof*⟩

lemma *dpll-bj-atms-in-trail*:
assumes
dpll-bj S T **and**
inv S **and**
atm-of ‘ (*lits-of-l* (*trail* S)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ S)
shows *atm-of* ‘ (*lits-of-l* (*trail* T)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ S)
⟨*proof*⟩

lemma *dpll-bj-atms-in-trail-in-set*:
assumes *dpll-bj* S T**and**
inv S **and**
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 *dpll-bj-all-decomposition-implies-inv*:
assumes
dpll-bj S T **and**
inv: *inv* S **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ T) (*get-all-ann-decomposition* (*trail* T))
⟨*proof*⟩

16.3.3 Termination

Using a proper measure **lemma** *length-get-all-ann-decomposition-append-Decided*:
length (*get-all-ann-decomposition* (F' @ *Decided* K () # F)) =
length (*get-all-ann-decomposition* F')
+ *length* (*get-all-ann-decomposition* (*Decided* K () # F))
− *1*
⟨*proof*⟩

lemma *take-length-get-all-ann-decomposition-decided-sandwich*:
take (*length* (*get-all-ann-decomposition* F))
(*map* (*f o snd*) (*rev* (*get-all-ann-decomposition* (F' @ *Decided* K () # F))))
=
map (*f o snd*) (*rev* (*get-all-ann-decomposition* F))

⟨*proof*⟩

lemma *length-get-all-ann-decomposition-length*:
length (*get-all-ann-decomposition* M) ≤ *1* + *length* M
⟨*proof*⟩

lemma *length-in-get-all-ann-decomposition-bounded*:
assumes i:i ∈ *set* (*trail-weight* S)
shows i ≤ *Suc* (*length* (*trail* S))
⟨*proof*⟩

Well-foundedness The bounds are the following:

- *1 + card (atms-of-ms A)*: *card (atms-of-ms A)* is an upper bound on the length of the list. As *get-all-ann-decomposition* appends an possibly empty couple at the end, adding one is needed.
- *2 + card (atms-of-ms A)*: *card (atms-of-ms A)* is an upper bound on the number of elements, where adding one is necessary for the same reason as for the bound on the list, and one is needed to have a strict bound.

abbreviation *unassigned-lit* :: *'b literal multiset set* $\Rightarrow$ *'a list* $\Rightarrow$ *nat* **where**
  unassigned-lit N M $\equiv$ *card (atms-of-ms N)* $-$ *length M*
lemma *dpll-bj-trail-mes-increasing-prop*:
  fixes *M* :: *('v, unit, unit) ann-lits* **and** *N* :: *'v clauses*
  assumes
    dpll-bj S T **and**
    inv S **and**
    NA: *atms-of-mm* ($clauses_{NOT}$ *S*) $\subseteq$ *atms-of-ms A* **and**
    MA: *atm-of ‘ lits-of-l (trail S)* $\subseteq$ *atms-of-ms A* **and**
    n-d: *no-dup (trail S)* **and**
    finite: *finite A*
  shows μ_C *(1+card (atms-of-ms A)) (2+card (atms-of-ms A)) (trail-weight T)*
    $>$ μ_C *(1+card (atms-of-ms A)) (2+card (atms-of-ms A)) (trail-weight S)*
  ⟨*proof*⟩

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+card (atms-of-ms A)) ^ (1+card (atms-of-ms A))*
        $-$ μ_C *(1+card (atms-of-ms A)) (2+card (atms-of-ms A)) (trail-weight T)*
      $<$ *(2+card (atms-of-ms A)) ^ (1+card (atms-of-ms A))*
        $-$ μ_C *(1+card (atms-of-ms A)) (2+card (atms-of-ms A)) (trail-weight S)*
⟨*proof*⟩

lemma *wf-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*}
  (**is** *wf ?A*)
⟨*proof*⟩

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 tat *satisfiable N*, $\neg$ *M* $\models$*as N* and there is no remaining step is incompatible.

1. The *decide* rule tells us that every variable in N has a value.

2. The assumption $\neg\ M \models as\ N$ implies that there is conflict.

3. There is at least one decision in the trail (otherwise, M would be a model of the set of clauses N).

4. Now if we build the clause with all the decision literals of the trail, we can apply the *backjump* rule.

   The assumption are saying that we have a finite upper bound A for the literals, that we cannot do any step *no-step dpll-bj S*

theorem *dpll-backjump-final-state*:
fixes A :: *'v 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-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* (*clauses$_{NOT}$ S*))
$\vee$ (*trail S* $\models$*asm clauses$_{NOT}$ S* $\wedge$ *satisfiable* (*set-mset* (*clauses$_{NOT}$ S*)))
⟨*proof*⟩

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* $\Rightarrow$ *'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*) *ann-lits* **and**
raw-clauses :: *'st* $\Rightarrow$ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *ann-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*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *bool*
+
assumes *dpll-bj-inv*: $\bigwedge S\ T.$ *dpll-bj S T* $\Longrightarrow$ *inv S* $\Longrightarrow$ *inv T*
begin

lemma *rtranclp-dpll-bj-inv*:
assumes *dpll-bj** S T* **and** *inv S*

shows *inv T*
⟨*proof*⟩

lemma *rtranclp-dpll-bj-no-dup*:
assumes *dpll-bj*** *S T* **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
⟨*proof*⟩

lemma *rtranclp-dpll-bj-atms-of-ms-clauses-inv*:
assumes
dpll-bj** *S T* **and** *inv S*
shows *atms-of-mm* (*clauses*$_{NOT}$ *S*) = *atms-of-mm* (*clauses*$_{NOT}$ *T*)
⟨*proof*⟩

lemma *rtranclp-dpll-bj-atms-in-trail*:
assumes
dpll-bj** *S T* **and**
inv S **and**
atm-of ‘ (*lits-of-l* (*trail S*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *T*)
⟨*proof*⟩

lemma *rtranclp-dpll-bj-sat-iff*:
assumes *dpll-bj*** *S T* **and** *inv S*
shows *I* ⊨*sm clauses*$_{NOT}$ *S* ⟷ *I* ⊨*sm clauses*$_{NOT}$ *T*
⟨*proof*⟩

lemma *rtranclp-dpll-bj-atms-in-trail-in-set*:
assumes
dpll-bj** *S T* **and**
inv S
atms-of-mm (*clauses*$_{NOT}$ *S*) ⊆ *A* **and**
atm-of ‘ (*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *A*
⟨*proof*⟩

lemma *rtranclp-dpll-bj-all-decomposition-implies-inv*:
assumes
dpll-bj** *S T* **and**
inv S
all-decomposition-implies-m (*clauses*$_{NOT}$ *S*) (*get-all-ann-decomposition* (*trail S*))
shows *all-decomposition-implies-m* (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
⟨*proof*⟩

lemma *rtranclp-dpll-bj-inv-incl-dpll-bj-inv-trancl*:
{(*T*, *S*). *dpll-bj*$^{++}$ *S T*
∧ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A*
∧ *no-dup* (*trail S*) ∧ *inv S*}
⊆ {(*T*, *S*). *dpll-bj S T* ∧ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A*
∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧ *no-dup* (*trail S*) ∧ *inv S*}$^{+}$
(**is** *?A* ⊆ *?B*$^{+}$)
⟨*proof*⟩

lemma *wf-tranclp-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 $\Longrightarrow$ *inv* S $\Longrightarrow$ I⊨*sextm* *clauses*$_{NOT}$ S $\longleftrightarrow$ I⊨*sextm* *clauses*$_{NOT}$ T
⟨*proof*⟩

lemma *rtranclp-dpll-bj-sat-ext-iff*:
dpll-bj** S T $\Longrightarrow$ *inv* S $\Longrightarrow$ I⊨*sextm* *clauses*$_{NOT}$ S $\longleftrightarrow$ 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) $\subseteq$ *atms-of-ms* A **and**
atms-trail: *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
finite A **and**
inv: *inv* S **and**
decomp: *all-decomposition-implies-m* (*clauses*$_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows *unsatisfiable* (*set-mset* (*clauses*$_{NOT}$ S))
$\vee$ (*trail* T ⊨*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 ⊨*asm* N $\wedge$ *satisfiable* (*set-mset* N))
⟨*proof*⟩

lemma *tranclp-dpll-bj-trail-mes-decreasing-prop*:
assumes *dpll*: *dpll-bj*$^{++}$ S T **and** *inv*: *inv* S **and**
N-A: *atms-of-mm* (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
M-A: *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
fin-A: *finite* A
shows (*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A))
$-$ μ_C (*1*+*card* (*atms-of-ms* A)) (*2*+*card* (*atms-of-ms* A)) (*trail-weight* T)
$<$ (*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A))
$-$ μ_C (*1*+*card* (*atms-of-ms* A)) (*2*+*card* (*atms-of-ms* A)) (*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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
  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 :: $'st \Rightarrow ('v, unit, unit)$ *ann-lits* **and**
    raw-clauses :: $'st \Rightarrow 'clss$ **and**
    prepend-trail :: $('v, unit, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow 'st$ **and**
    tl-trail :: $'st \Rightarrow 'st$ **and**
    $add\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ **and**
    $remove\text{-}cls_{NOT}$ :: $'cls \Rightarrow 'st \Rightarrow 'st$ +
  fixes
    learn-cond :: $'cls \Rightarrow 'st \Rightarrow bool$
begin
inductive *learn* :: $'st \Rightarrow 'st \Rightarrow bool$ **where**
$learn_{NOT}$*-rule*: $clauses_{NOT}$ $S \models pm$ *mset-cls* $C \Longrightarrow$
  atms-of (*mset-cls* C) $\subseteq$ *atms-of-mm* ($clauses_{NOT}$ S) $\cup$ *atm-of* ` (*lits-of-l* (*trail* S)) $\Longrightarrow$
  learn-cond C $S \Longrightarrow$
  $T \sim add\text{-}cls_{NOT}$ C $S \Longrightarrow$
  learn S T
inductive-cases $learn_{NOT}E$: *learn* S T

lemma *learn-*μ_C*-stable*:
  assumes *learn* S T **and** *no-dup* (*trail* S)
  shows μ_C A B (*trail-weight* S) = μ_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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
  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* ⇒ *'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*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* +
fixes
forget-cond :: *'cls* ⇒ *'st* ⇒ *bool*
begin
inductive *forget*$_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* **where**
forget$_{NOT}$:
removeAll-mset (*mset-cls* C)(*clauses*$_{NOT}$ S) ⊨*pm* *mset-cls* C ⟹
forget-cond C S ⟹
C !∈! *raw-clauses* S ⟹
T ∼ *remove-cls*$_{NOT}$ C S ⟹
forget$_{NOT}$ S T
inductive-cases *forget*$_{NOT}$*E*: *forget*$_{NOT}$ S T

lemma *forget-*μ_C*-stable*:
assumes *forget*$_{NOT}$ S T
shows μ_C A B (*trail-weight* S) = μ_C A B (*trail-weight* T)
⟨*proof*⟩
end

locale *learn-and-forget*$_{NOT}$ =
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-cls$_{NOT}$ *remove-cls*$_{NOT}$ *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-cls$_{NOT}$ *remove-cls*$_{NOT}$ *forget-cond*
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*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
add-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
remove-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
learn-cond forget-cond :: *'cls* ⇒ *'st* ⇒ *bool*
begin
inductive *learn-and-forget*$_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool*
where

lf-learn: *learn S T* $\Longrightarrow$ *learn-and-forget*$_{NOT}$ *S T* |
lf-forget: *forget*$_{NOT}$ *S T* $\Longrightarrow$ *learn-and-forget*$_{NOT}$ *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-cls$_{NOT}$ *remove-cls*$_{NOT}$
    inv backjump-conds propagate-conds +
  learn-and-forget$_{NOT}$ *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*$_{NOT}$ *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*) *ann-lits* **and**
    raw-clauses :: *'st* $\Rightarrow$ *'clss* **and**
    prepend-trail :: (*'v*, *unit*, *unit*) *ann-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*) *ann-lit* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **and**
    learn-cond forget-cond :: *'cls* $\Rightarrow$ *'st* $\Rightarrow$ *bool*
begin

inductive *cdcl*$_{NOT}$ :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** *S* :: *'st* **where**
c-dpll-bj: *dpll-bj S S'* $\Longrightarrow$ *cdcl*$_{NOT}$ *S S'* |
c-learn: *learn S S'* $\Longrightarrow$ *cdcl*$_{NOT}$ *S S'* |
c-forget$_{NOT}$: *forget*$_{NOT}$ *S S'* $\Longrightarrow$ *cdcl*$_{NOT}$ *S S'*

lemma *cdcl*$_{NOT}$*-all-induct*[*consumes 1*, *case-names dpll-bj learn forget*$_{NOT}$]:
  fixes *S T* :: *'st*
  assumes *cdcl*$_{NOT}$ *S T* **and**
    dpll: $\bigwedge T$. *dpll-bj S T* $\Longrightarrow$ *P S T* **and**
    learning:
      $\bigwedge C\ T$. *clauses*$_{NOT}$ *S* $\models$*pm mset-cls C* $\Longrightarrow$
      atms-of (*mset-cls C*) $\subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail S*)) $\Longrightarrow$
      T $\sim$ *add-cls*$_{NOT}$ *C S* $\Longrightarrow$
      P S T **and**
    forgetting: $\bigwedge C\ T$. *removeAll-mset* (*mset-cls C*) (*clauses*$_{NOT}$ *S*) $\models$*pm mset-cls C* $\Longrightarrow$
      C !$\in$! *raw-clauses S* $\Longrightarrow$
      T $\sim$ *remove-cls*$_{NOT}$ *C S* $\Longrightarrow$
      P S T
  shows *P S T*
  ⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-no-dup*:
assumes
cdcl$_{NOT}$ *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
cdcl$_{NOT}$ *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 *cdcl*$_{NOT}$*-atms-of-ms-clauses-decreasing*:
assumes *cdcl*$_{NOT}$ *S T***and** *inv S* **and** *no-dup* (*trail S*)
shows *atms-of-mm* (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *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*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *atms-of-mm* (*clauses*$_{NOT}$ *S*)
⟨*proof*⟩

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*) ⊆ *A* **and**
atm-of ‘ (*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *A*
⟨*proof*⟩

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-ann-decomposition* (*trail S*))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
⟨*proof*⟩

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*⊨*sextm clauses*$_{NOT}$ *S* ⟷ *I*⊨*sextm clauses*$_{NOT}$ *T*
⟨*proof*⟩

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*: $\bigwedge S\ T.\ cdcl_{NOT}\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
begin
sublocale *dpll-with-backjumping*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-inv*:
$cdcl_{NOT}{}^{**}\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-no-dup*:
assumes $cdcl_{NOT}{}^{**}\ S\ T$ **and** *inv S*
and *no-dup* (*trail S*)
shows *no-dup* (*trail T*)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-trail-clauses-bound*:
assumes
cdcl: $cdcl_{NOT}{}^{**}\ S\ T$ **and**
inv: *inv S* **and**
n-d: *no-dup* (*trail S*) **and**
atms-clauses-S: *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *A* **and**
atms-trail-S: *atm-of* ‘(*lits-of-l* (*trail S*)) ⊆ *A*
shows *atm-of* ‘ (*lits-of-l* (*trail T*)) ⊆ *A* ∧ *atms-of-mm* ($clauses_{NOT}$ *T*) ⊆ *A*
⟨*proof*⟩

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-ann-decomposition* (*trail S*))
shows
all-decomposition-implies-m ($clauses_{NOT}$ *T*) (*get-all-ann-decomposition* (*trail T*))
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-bj-sat-ext-iff*:
assumes $cdcl_{NOT}{}^{**}\ S\ T$**and** *inv S* **and** *no-dup* (*trail S*)
shows *I*⊨*sextm* $clauses_{NOT}$ *S* ⟷ *I*⊨*sextm* $clauses_{NOT}$ *T*
⟨*proof*⟩

definition $cdcl_{NOT}$*-NOT-all-inv* **where**
$cdcl_{NOT}$*-NOT-all-inv A S* ⟷ (*finite A* ∧ *inv S* ∧ *atms-of-mm* ($clauses_{NOT}$ *S*) ⊆ *atms-of-ms A*
∧ *atm-of* ‘ *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧ *no-dup* (*trail S*))

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*
⟨*proof*⟩

abbreviation *learn-or-forget* **where**
learn-or-forget S T ≡ *learn S T* ∨ $forget_{NOT}$ *S T*

lemma *rtranclp-learn-or-forget-*$cdcl_{NOT}$:
learn-or-forget** *S T* ⟹ $cdcl_{NOT}{}^{**}\ S\ T$

⟨*proof*⟩

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*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight U*)
< (*2*+*card* (*atms-of-ms A*)) ^ (*1*+*card* (*atms-of-ms A*))
− μ_C (*1*+*card* (*atms-of-ms A*)) (*2*+*card* (*atms-of-ms A*)) (*trail-weight S*)
(**is** *?μ U* < *?μ S*)
⟨*proof*⟩

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 ∃ *j*. ∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*))
⟨*proof*⟩

lemma *wf-*$cdcl_{NOT}$*-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: $\bigwedge$*f j*. ¬ (∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*)))
shows *wf* {(*T*, *S*). $cdcl_{NOT}$ *S T* ∧ $cdcl_{NOT}$*-NOT-all-inv A S*}
(**is** *wf* {(*T*, *S*). $cdcl_{NOT}$ *S T* ∧ *?inv S*})
⟨*proof*⟩

lemma *inv-and-tranclp-cdcl-*$_{NOT}$*-tranclp-*$cdcl_{NOT}$*-and-inv*:
$cdcl_{NOT}{}^{++}$ *S T* ∧ $cdcl_{NOT}$*-NOT-all-inv A S* ⟷ (λ*S T*. $cdcl_{NOT}$ *S T* ∧ $cdcl_{NOT}$*-NOT-all-inv A S*)$^{++}$ *S T*
(**is** *?A* ∧ *?I* ⟷ *?B*)
⟨*proof*⟩

lemma *wf-tranclp-*$cdcl_{NOT}$*-no-learn-and-forget-infinite-chain*:
assumes
no-infinite-lf: $\bigwedge$*f j*. ¬ (∀ *i*≥*j*. *learn-or-forget* (*f i*) (*f* (*Suc i*)))
shows *wf* {(*T*, *S*). $cdcl_{NOT}{}^{++}$ *S T* ∧ $cdcl_{NOT}$*-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-ann-decomposition* (*trail S*))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ *S*))
∨ (*trail S* ⊨*asm* $clauses_{NOT}$ *S* ∧ *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-ann-decomposition* (*trail S*))

shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ *T*))
∨ (*trail T* ⊨*asm* $clauses_{NOT}$ *T* ∧ *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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds
λ*C S*. *distinct-mset* (*mset-cls C*) ∧ ¬*tautology* (*mset-cls C*) ∧ *learn-restrictions C S* ∧
(∃ *F K d F′ C′ L*. *trail S* = *F′* @ *Decided K* () # *F* ∧ *mset-cls C* = *C′* + {#*L*#} ∧ *F* ⊨*as CNot C′*
∧ *C′* + {#*L*#} ∉# $clauses_{NOT}$ *S*)
λ*C S*. ¬(∃ *F′ F K d L*. *trail S* = *F′* @ *Decided K* () # *F* ∧ *F* ⊨*as CNot* (*remove1-mset L* (*mset-cls C*)))
∧ *forget-restrictions C S*
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*) *ann-lits* **and**
raw-clauses :: ′*st* ⇒ ′*clss* **and**
prepend-trail :: (′*v*, *unit*, *unit*) *ann-lit* ⇒ ′*st* ⇒ ′*st* **and**
tl-trail :: ′*st* ⇒′*st* **and**
$add\text{-}cls_{NOT}$ :: ′*cls* ⇒ ′*st* ⇒ ′*st* **and**
$remove\text{-}cls_{NOT}$ :: ′*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*) *ann-lit* ⇒ ′*st* ⇒ *bool* **and**
learn-restrictions forget-restrictions :: ′*cls* ⇒ ′*st* ⇒ *bool*
begin

lemma $cdcl_{NOT}$-*learn-all-induct*[*consumes 1*, *case-names dpll-bj learn* $forget_{NOT}$]:
fixes *S T* :: ′*st*
assumes $cdcl_{NOT}$ *S T* **and**

dpll: $\bigwedge T$. *dpll-bj* S $T \Longrightarrow P$ S T **and**
learning:
$\bigwedge C\ F\ K\ F'\ C'\ L\ T$. $clauses_{NOT}$ S $\models pm$ *mset-cls* $C \Longrightarrow$
atms-of (*mset-cls* C) $\subseteq$ *atms-of-mm* ($clauses_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail* S)) $\Longrightarrow$
distinct-mset (*mset-cls* C) $\Longrightarrow$
$\neg$ *tautology* (*mset-cls* C) $\Longrightarrow$
learn-restrictions C $S \Longrightarrow$
trail $S = F'$ @ *Decided* K () # $F \Longrightarrow$
mset-cls $C = C' + \{\#L\#\} \Longrightarrow$
$F \models as$ *CNot* $C' \Longrightarrow$
$C' + \{\#L\#\} \notin\#$ $clauses_{NOT}$ $S \Longrightarrow$
$T \sim add\text{-}cls_{NOT}$ C $S \Longrightarrow$
P S T **and**
forgetting: $\bigwedge C\ T$. *removeAll-mset* (*mset-cls* C) ($clauses_{NOT}$ S) $\models pm$ *mset-cls* $C \Longrightarrow$
C !∈! *raw-clauses* $S \Longrightarrow$
$\neg(\exists F'\ F\ K\ L$. *trail* $S = F'$ @ *Decided* K () # $F \wedge F \models as$ *CNot* (*mset-cls* $C - \{\#L\#\}$)) $\Longrightarrow$
$T \sim remove\text{-}cls_{NOT}$ C $S \Longrightarrow$
forget-restrictions C $S \Longrightarrow$
P S T
shows P S T
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-inv*:
$cdcl_{NOT}{}^{**}$ S $T \Longrightarrow inv$ $S \Longrightarrow inv$ 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-clss* (*atms-of-mm* ($clauses_{NOT}$ S) $\cup$ *atm-of* ‘ *lits-of-l* (*trail* S))
⟨*proof*⟩

definition *conflicting-bj-clss* $S \equiv$
$\{C+\{\#L\#\}$ |C L. $C+\{\#L\#\} \in\#$ $clauses_{NOT}$ S $\wedge$ *distinct-mset* ($C+\{\#L\#\}$)
$\wedge$ ¬*tautology* ($C+\{\#L\#\}$)
$\wedge$ ($\exists F'\ K\ F$. *trail* $S = F'$ @ *Decided* K () # $F \wedge F \models as$ *CNot* C)}

lemma *conflicting-bj-clss-remove-cls*$_{NOT}$[*simp*]:
conflicting-bj-clss ($remove\text{-}cls_{NOT}$ C S) = *conflicting-bj-clss* S $-$ {*mset-cls* C}
⟨*proof*⟩

lemma *conflicting-bj-clss-remove-cls*$_{NOT}$'[*simp*]:
$T \sim remove\text{-}cls_{NOT}$ C $S \Longrightarrow$ *conflicting-bj-clss* T = *conflicting-bj-clss* S $-$ {*mset-cls* C}
⟨*proof*⟩

lemma *conflicting-bj-clss-add-cls*$_{NOT}$*-state-eq*:
assumes
T: $T \sim add\text{-}cls_{NOT}$ C' S **and**
n-d: *no-dup* (*trail* S)
shows *conflicting-bj-clss* T
= *conflicting-bj-clss* S
$\cup$ (*if* $\exists C\ L$. *mset-cls* $C' = C + \{\#L\#\}$ $\wedge$ *distinct-mset* ($C+\{\#L\#\}$) $\wedge$ ¬*tautology* ($C+\{\#L\#\}$)
$\wedge$ ($\exists F'\ K\ d\ F$. *trail* $S = F'$ @ *Decided* K () # $F \wedge F \models as$ *CNot* C)

then {*mset-cls* C'} *else* {})
⟨*proof*⟩

lemma *conflicting-bj-clss-add-cls*$_{NOT}$:
no-dup (*trail* S) $\Longrightarrow$
conflicting-bj-clss (*add-cls*$_{NOT}$ C' S)
= *conflicting-bj-clss* S
$\cup$ (*if* $\exists$ C L. *mset-cls* C' = C +{#L#}$\wedge$ *distinct-mset* (C+{#L#}) $\wedge$ $\neg$*tautology* (C+{#L#})
$\wedge$ ($\exists$ F' K d F. *trail* S = F' @ *Decided* K () # F $\wedge$ F $\models$*as* *CNot* C)
then {*mset-cls* C'} *else* {})
⟨*proof*⟩

lemma *conflicting-bj-clss-incl-clauses*:
conflicting-bj-clss S $\subseteq$ *set-mset* (*clauses*$_{NOT}$ S)
⟨*proof*⟩

lemma *finite-conflicting-bj-clss*[*simp*]:
finite (*conflicting-bj-clss* S)
⟨*proof*⟩

lemma *learn-conflicting-increasing*:
no-dup (*trail* S) $\Longrightarrow$ *learn* S T $\Longrightarrow$ *conflicting-bj-clss* S $\subseteq$ *conflicting-bj-clss* T
⟨*proof*⟩

abbreviation *conflicting-bj-clss-yet* b S $\equiv$
3 ^ b $-$ *card* (*conflicting-bj-clss* S)

abbreviation μ_L :: *nat* $\Rightarrow$ *'st* $\Rightarrow$ *nat* $\times$ *nat* **where**
μ_L b S $\equiv$ (*conflicting-bj-clss-yet* b S, *card* (*set-mset* (*clauses*$_{NOT}$ S)))

lemma *do-not-forget-before-backtrack-rule-clause-learned-clause-untouched*:
assumes *forget*$_{NOT}$ S T
shows *conflicting-bj-clss* S = *conflicting-bj-clss* T
⟨*proof*⟩

lemma *forget-*μ_L*-decrease*:
assumes *forget*$_{NOT}$: *forget*$_{NOT}$ S T
shows (μ_L b T, μ_L b S) $\in$ *less-than* <**lex**> *less-than*
⟨*proof*⟩

lemma *set-condition-or-split*:
{a. (a = b $\vee$ Q a) $\wedge$ S a} = (*if* S b *then* {b} *else* {}) $\cup$ {a. Q a $\wedge$ S a}
⟨*proof*⟩

lemma *set-insert-neq*:
A $\neq$ *insert* a A $\longleftrightarrow$ a $\notin$ A
⟨*proof*⟩

lemma *learn-*μ_L*-decrease*:
assumes *learnST*: *learn* S T **and** *n-d*: *no-dup* (*trail* S) **and**
A: *atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ' *lits-of-l* (*trail* S) $\subseteq$ A **and**
fin-A: *finite* A
shows (μ_L (*card* A) T, μ_L (*card* A) S) $\in$ *less-than* <**lex**> *less-than*
⟨*proof*⟩

We have to assume the following:

- *inv S*: the invariant holds in the inital state.
- A is a (finite *finite A*) superset of the literals in the trail *atm-of ` lits-of-l (trail S)* $\subseteq$ *atms-of-ms A* and in the clauses *atms-of-mm* $(clauses_{NOT}\ S) \subseteq$ *atms-of-ms A*. This can the the set of all the literals in the starting set of clauses.
- *no-dup (trail S)*: no duplicate in the trail. This is invariant along the path.

definition μ_{CDCL} **where**
$\mu_{CDCL}\ A\ T \equiv ((2{+}card\ (atms\text{-}of\text{-}ms\ A))\ \widehat{}\ (1{+}card\ (atms\text{-}of\text{-}ms\ A))$
$\quad - \mu_C\ (1{+}card\ (atms\text{-}of\text{-}ms\ A))\ (2{+}card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T),$
$\quad conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ T,\ card\ (set\text{-}mset\ (clauses_{NOT}\ T)))$
lemma $cdcl_{NOT}$*-decreasing-measure*:
assumes
$cdcl_{NOT}\ S\ T$ **and**
inv: *inv S* **and**
atm-clss: *atms-of-mm* $(clauses_{NOT}\ S) \subseteq$ *atms-of-ms A* **and**
atm-lits: *atm-of ` lits-of-l (trail S)* $\subseteq$ *atms-of-ms A* **and**
n-d: *no-dup (trail S)* **and**
fin-A: *finite A*
shows $(\mu_{CDCL}\ A\ T,\ \mu_{CDCL}\ A\ S)$
$\in$ *less-than <*lex*> (less-than <*lex*> less-than)*
⟨*proof*⟩

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*)
$\wedge\ cdcl_{NOT}\ S\ T$ }
⟨*proof*⟩

definition $\mu_C{}'$ :: *'v literal multiset set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_C{}'\ A\ T \equiv \mu_C\ (1{+}card\ (atms\text{-}of\text{-}ms\ A))\ (2{+}card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T)$

definition $\mu_{CDCL}{}'$ :: *'v literal multiset set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_{CDCL}{}'\ A\ T \equiv$
$((2{+}card\ (atms\text{-}of\text{-}ms\ A))\ \widehat{}\ (1{+}card\ (atms\text{-}of\text{-}ms\ A)) - \mu_C{}'\ A\ T) * (1{+}\ 3\widehat{}card\ (atms\text{-}of\text{-}ms\ A)) * 2$
$+\ conflicting\text{-}bj\text{-}clss\text{-}yet\ (card\ (atms\text{-}of\text{-}ms\ A))\ T * 2$
$+\ card\ (set\text{-}mset\ (clauses_{NOT}\ T))$

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 $\mu_{CDCL}{}'\ A\ T < \mu_{CDCL}{}'\ A\ S$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rtranclp-$cdcl_{NOT}$-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*
⟨*proof*⟩

lemma *rtranclp-$cdcl_{NOT}$-card-clauses-bound*:
assumes
$cdcl_{NOT}^{**}$ S T **and**
inv S **and**
atms-of-mm (*$clauses_{NOT}$ S*) $\subseteq$ *A* **and**
atm-of ‘(*lits-of-l* (*trail S*)) $\subseteq$ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *card* (*set-mset* (*$clauses_{NOT}$ T*)) $\leq$ *card* (*set-mset* (*$clauses_{NOT}$ S*)) + *3* ^ (*card A*)
⟨*proof*⟩

lemma *rtranclp-$cdcl_{NOT}$-card-clauses-bound'*:
assumes
$cdcl_{NOT}^{**}$ S T **and**
inv S **and**
atms-of-mm (*$clauses_{NOT}$ S*) $\subseteq$ *A* **and**
atm-of ‘(*lits-of-l* (*trail S*)) $\subseteq$ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *card* {*C*|*C*. *C* $\in\#$ *$clauses_{NOT}$ T* $\wedge$ (*tautology C* $\vee$ $\neg$*distinct-mset C*)}
$\leq$ *card* {*C*|*C*. *C*$\in\#$ *$clauses_{NOT}$ S* $\wedge$ (*tautology C* $\vee$ $\neg$*distinct-mset C*)} + *3* ^ (*card A*)
(**is** *card ?T* $\leq$ *card ?S* + -)
⟨*proof*⟩

lemma *rtranclp-$cdcl_{NOT}$-card-simple-clauses-bound*:
assumes
$cdcl_{NOT}^{**}$ S T **and**
inv S **and**
NA: *atms-of-mm* (*$clauses_{NOT}$ S*) $\subseteq$ *A* **and**
MA: *atm-of* ‘ (*lits-of-l* (*trail S*)) $\subseteq$ *A* **and**
n-d: *no-dup* (*trail S*) **and**
finite: *finite A*
shows *card* (*set-mset* (*$clauses_{NOT}$ T*))
$\leq$ *card* {*C*. *C* $\in\#$ *$clauses_{NOT}$ S* $\wedge$ (*tautology C* $\vee$ $\neg$*distinct-mset C*)} + *3* ^ (*card A*)

(**is** *card ?T* $\leq$ *card ?S* + -)
⟨*proof*⟩

definition μ_{CDCL}*'-bound* :: *'v literal multiset set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
μ_{CDCL}*'-bound A S* =
((*2* + *card* (*atms-of-ms A*)) ^ (*1* + *card* (*atms-of-ms A*))) $*$ (*1* + *3* ^ *card* (*atms-of-ms A*)) $*$ *2*
+ *2*$*$*3* ^ (*card* (*atms-of-ms A*))
+ *card* {*C*. *C* $\in$# *clauses*$_{NOT}$ *S* $\wedge$ (*tautology C* $\vee$ $\neg$*distinct-mset C*)} + *3* ^ (*card* (*atms-of-ms A*))

lemma μ_{CDCL}*'-bound-reduce-trail-to*$_{NOT}$[*simp*]:
μ_{CDCL}*'-bound A* (*reduce-trail-to*$_{NOT}$ *M S*) = μ_{CDCL}*'-bound A S*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-*μ_{CDCL}*'-bound-reduce-trail-to*$_{NOT}$:
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*) **and**
U: *U* $\sim$ *reduce-trail-to*$_{NOT}$ *M T*
shows μ_{CDCL}*' A U* $\leq$ μ_{CDCL}*'-bound A S*
⟨*proof*⟩

lemma *rtranclp-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 μ_{CDCL}*' A T* $\leq$ μ_{CDCL}*'-bound A S*
⟨*proof*⟩

lemma *rtranclp-*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 μ_{CDCL}*'-bound A T* $\leq$ μ_{CDCL}*'-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* ⇒ *'st* ⇒ *bool*
begin
inductive $cdcl_{NOT}$*-raw-restart* :: *'st* ⇒ *'st* ⇒ *bool* **where**
$cdcl_{NOT}$ *S T* ⟹ $cdcl_{NOT}$*-raw-restart S T* |
restart S T ⟹ $cdcl_{NOT}$*-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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
inv backjump-conds propagate-conds learn-cond forget-cond
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*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'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*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
learn-cond forget-cond :: *'cls* ⇒ *'st* ⇒ *bool*
begin

lemma $cdcl_{NOT}$*-iff-*$cdcl_{NOT}$*-raw-restart-no-restarts*:
$cdcl_{NOT}$ *S T* ⟷ *restart-ops.*$cdcl_{NOT}$*-raw-restart* $cdcl_{NOT}$ (λ- -. *False*) *S T*
(**is** *?C S T* ⟷ *?R S T*)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-*$cdcl_{NOT}$*-raw-restart*:
$cdcl_{NOT}$ *S T* ⟹ *restart-ops.*$cdcl_{NOT}$*-raw-restart* $cdcl_{NOT}$ *restart S T*
⟨*proof*⟩
end

16.5.2 Increasing restarts

To add restarts we needs 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*) ≤ *f n* for (*1*::*'a*) ≤ *n*: it means that between two consecutive restarts, at least one step will be done. This is necessary to avoid sequence. like: full – restart – full – ...
- a measure *μ*: it should decrease under the assumptions *bound-inv*, whenever a $cdcl_{NOT}$ or a *restart* is done. A parameter is given to *μ*: for conflict- driven clause learning, it is

an upper-bound of the clauses. We are assuming that such a bound can be found after a restart whenever the invariant holds.

- we also assume that the measure decrease after any $cdcl_{NOT}$ step.
- an invariant on the states $cdcl_{NOT}$*-inv* that also holds after restarts.
- it is *not required* that the measure decrease with respect to restarts, but the measure has to be bound by some function μ*-bound* taking the same parameter as μ and the initial state of the considered $cdcl_{NOT}$ chain.

locale $cdcl_{NOT}$*-increasing-restarts-ops* =
  restart-ops $cdcl_{NOT}$ *restart* **for**
    restart :: *'st* ⇒ *'st* ⇒ *bool* **and**
    $cdcl_{NOT}$ :: *'st* ⇒ *'st* ⇒ *bool* +
  fixes
    f :: *nat* ⇒ *nat* **and**
    bound-inv :: *'bound* ⇒ *'st* ⇒ *bool* **and**
    μ :: *'bound* ⇒ *'st* ⇒ *nat* **and**
    $cdcl_{NOT}$*-inv* :: *'st* ⇒ *bool* **and**
    μ*-bound* :: *'bound* ⇒ *'st* ⇒ *nat*
  assumes
    f: *unbounded f* **and**
    f-ge-1: $\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \neq 0$ **and**
    bound-inv: $\bigwedge A\ S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow$ *bound-inv* $A\ S \Longrightarrow$ $cdcl_{NOT}\ S\ T \Longrightarrow$ *bound-inv* $A\ T$ **and**
    $cdcl_{NOT}$*-measure*: $\bigwedge A\ S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow$ *bound-inv* $A\ S \Longrightarrow$ $cdcl_{NOT}\ S\ T \Longrightarrow \mu\ A\ T < \mu$
$A\ S$ **and**
    measure-bound2: $\bigwedge A\ T\ U.$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T \Longrightarrow {cdcl_{NOT}}^{**}\ T\ U$
      $\Longrightarrow \mu\ A\ U \leq \mu$*-bound* $A\ T$ **and**
    measure-bound4: $\bigwedge A\ T\ U.$ $cdcl_{NOT}$*-inv* $T \Longrightarrow$ *bound-inv* $A\ T \Longrightarrow {cdcl_{NOT}}^{**}\ T\ U$
      $\Longrightarrow \mu$*-bound* $A\ U \leq \mu$*-bound* $A\ T$ **and**
    $cdcl_{NOT}$*-restart-inv*: $\bigwedge A\ U\ V.$ $cdcl_{NOT}$*-inv* $U \Longrightarrow$ *restart* $U\ V \Longrightarrow$ *bound-inv* $A\ U \Longrightarrow$ *bound-inv*
$A\ V$
      and
    exists-bound: $\bigwedge R\ S.$ $cdcl_{NOT}$*-inv* $R \Longrightarrow$ *restart* $R\ S \Longrightarrow \exists A.$ *bound-inv* $A\ S$ **and**
    $cdcl_{NOT}$*-inv*: $\bigwedge S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow cdcl_{NOT}\ S\ T \Longrightarrow$ $cdcl_{NOT}$*-inv* T **and**
    $cdcl_{NOT}$*-inv-restart*: $\bigwedge S\ T.$ $cdcl_{NOT}$*-inv* $S \Longrightarrow$ *restart* $S\ T \Longrightarrow$ $cdcl_{NOT}$*-inv* T
begin

lemma $cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*:
  assumes
    $(cdcl_{NOT}\widehat{\ }\widehat{\ }n)\ S\ T$ **and**
    $cdcl_{NOT}$*-inv* S
  shows $cdcl_{NOT}$*-inv* T
  ⟨*proof*⟩

lemma $cdcl_{NOT}$*-bound-inv*:
  assumes
    $(cdcl_{NOT}\widehat{\ }\widehat{\ }n)\ S\ T$ **and**
    $cdcl_{NOT}$*-inv* S
    bound-inv $A\ S$
  shows *bound-inv* $A\ T$
  ⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-*$cdcl_{NOT}$*-inv*:

assumes
  $cdcl_{NOT}^{**}$ S T **and**
  $cdcl_{NOT}$-*inv* S
shows $cdcl_{NOT}$-*inv* T
⟨*proof*⟩

lemma *rtranclp*-$cdcl_{NOT}$-*bound-inv*:
assumes
  $cdcl_{NOT}^{**}$ S T **and**
  bound-inv A S **and**
  $cdcl_{NOT}$-*inv* S
shows *bound-inv* A T
⟨*proof*⟩

lemma $cdcl_{NOT}$-*comp-n-le*:
assumes
  $(cdcl_{NOT}{}^{\frown\frown}(Suc\ n))\ S\ T$ **and**
  bound-inv A S
  $cdcl_{NOT}$-*inv* S
shows $\mu\ A\ T < \mu\ A\ S - n$
⟨*proof*⟩

lemma *wf*-$cdcl_{NOT}$:
wf $\{(T, S).\ cdcl_{NOT}\ S\ T \wedge cdcl_{NOT}\text{-}inv\ S \wedge bound\text{-}inv\ A\ S\}$ (**is** *wf ?A*)
⟨*proof*⟩

lemma *rtranclp*-$cdcl_{NOT}$-*measure*:
assumes
  $cdcl_{NOT}^{**}$ S T **and**
  bound-inv A S **and**
  $cdcl_{NOT}$-*inv* S
shows $\mu\ A\ T \leq \mu\ A\ S$
⟨*proof*⟩

lemma $cdcl_{NOT}$-*comp-bounded*:
assumes
  bound-inv A S **and** $cdcl_{NOT}$-*inv* S **and** $m \geq 1+\mu\ A\ S$
shows $\neg(cdcl_{NOT}{}^{\frown\frown} m)\ S\ T$
⟨*proof*⟩

- $f\ n < m$ ensures that at least one step has been done.

inductive $cdcl_{NOT}$-*restart* **where**
restart-step: $(cdcl_{NOT}{}^{\frown\frown}m)\ S\ T \Longrightarrow m \geq f\ n \Longrightarrow restart\ T\ U$
  $\Longrightarrow cdcl_{NOT}\text{-}restart\ (S, n)\ (U, Suc\ n)$ |
restart-full: *full1* $cdcl_{NOT}$ S $T \Longrightarrow cdcl_{NOT}\text{-}restart\ (S, n)\ (T, Suc\ n)$

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 $T \Longrightarrow cdcl_{NOT}\text{-}raw\text{-}restart^{**}\ (fst\ S)\ (fst\ T)$
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-with-restart-*$cdcl_{NOT}$*-inv*:
assumes
  $cdcl_{NOT}$*-restart*** S T **and**
  $cdcl_{NOT}$*-inv* (*fst* S)
shows $cdcl_{NOT}$*-inv* (*fst* T)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-with-restart-bound-inv*:
assumes
  $cdcl_{NOT}$*-restart*** S T **and**
  $cdcl_{NOT}$*-inv* (*fst* S) **and**
  bound-inv A (*fst* S)
shows *bound-inv* A (*fst* T)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-with-restart-increasing-number*:
  $cdcl_{NOT}$*-restart* S $T \Longrightarrow$ *snd* $T = 1 +$ *snd* S
⟨*proof*⟩
end

locale $cdcl_{NOT}$*-increasing-restarts* =
  $cdcl_{NOT}$*-increasing-restarts-ops restart* $cdcl_{NOT}$ *f bound-inv* μ $cdcl_{NOT}$*-inv* μ*-bound* +
  dpll-state mset-cls 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* ⇒ *'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*) *ann-lits* **and**
  raw-clauses :: *'st* ⇒ *'clss* **and**
  prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
  tl-trail :: *'st* ⇒*'st* **and**
  add-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
  remove-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
  f :: *nat* ⇒ *nat* **and**

$restart :: 'st \Rightarrow 'st \Rightarrow bool$ **and**
$bound\text{-}inv :: 'bound \Rightarrow 'st \Rightarrow bool$ **and**
$\mu :: 'bound \Rightarrow 'st \Rightarrow nat$ **and**
$cdcl_{NOT} :: 'st \Rightarrow 'st \Rightarrow bool$ **and**
$cdcl_{NOT}\text{-}inv :: 'st \Rightarrow bool$ **and**
$\mu\text{-}bound :: 'bound \Rightarrow 'st \Rightarrow nat$ +
assumes
measure-bound: $\bigwedge A\ T\ V\ n.\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow cdcl_{NOT}\text{-}restart\ (T,\ n)\ (V,\ Suc\ n) \Longrightarrow \mu\ A\ V \leq \mu\text{-}bound\ A\ T$ **and**
$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}\text{-}restart\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\text{-}bound\ A\ V \leq \mu\text{-}bound\ A\ T$
begin

lemma *rtranclp-*$cdcl_{NOT}$*-raw-restart-*μ*-bound*:
$cdcl_{NOT}\text{-}restart^{**}\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\text{-}bound\ A\ V \leq \mu\text{-}bound\ A\ T$
⟨*proof*⟩

lemma $cdcl_{NOT}$*-raw-restart-measure-bound*:
$cdcl_{NOT}\text{-}restart\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu\text{-}bound\ A\ T$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-raw-restart-measure-bound*:
$cdcl_{NOT}\text{-}restart^{**}\ (T,\ a)\ (V,\ b) \Longrightarrow\ cdcl_{NOT}\text{-}inv\ T \Longrightarrow bound\text{-}inv\ A\ T$
$\Longrightarrow \mu\ A\ V \leq \mu\text{-}bound\ A\ T$
⟨*proof*⟩

lemma *wf-*$cdcl_{NOT}$*-restart*:
$wf\ \{(T,\ S).\ cdcl_{NOT}\text{-}restart\ S\ T \wedge cdcl_{NOT}\text{-}inv\ (fst\ S)\}$ (**is** $wf\ ?A$)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-steps-bigger-than-bound*:
assumes
$cdcl_{NOT}\text{-}restart\ S\ T$ **and**
$bound\text{-}inv\ A\ (fst\ S)$ **and**
$cdcl_{NOT}\text{-}inv\ (fst\ S)$ **and**
$f\ (snd\ S) > \mu\text{-}bound\ A\ (fst\ S)$
shows $full1\ cdcl_{NOT}\ (fst\ S)\ (fst\ T)$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-with-inv-inv-rtranclp-*$cdcl_{NOT}$:
assumes
inv: $cdcl_{NOT}\text{-}inv\ S$ **and**
binv: $bound\text{-}inv\ A\ S$
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** $?A^{**}\ S\ T \longleftrightarrow ?B^{**}\ S\ T$)
⟨*proof*⟩

lemma *no-step-*$cdcl_{NOT}$*-restart-no-step-*$cdcl_{NOT}$:
assumes
n-s: $no\text{-}step\ cdcl_{NOT}\text{-}restart\ S$ **and**
inv: $cdcl_{NOT}\text{-}inv\ (fst\ S)$ **and**
binv: $bound\text{-}inv\ A\ (fst\ S)$

shows *no-step* $cdcl_{NOT}$ (*fst S*)
⟨*proof*⟩

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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ +
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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$ *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*) *ann-lits* **and**
raw-clauses :: *'st* ⇒ *'clss* **and**
prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒*'st* **and**
$add\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
$remove\text{-}cls_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
forget-cond :: *'cls* ⇒ *'st* ⇒ *bool* +
fixes *backjump-l-cond* :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool*
begin

We have a new backjump that combines the backjumping on the trail and the learning of the used clause (called C'' below)

inductive *backjump-l* **where**
backjump-l: *trail S* = *F'* @ *Decided K* () # *F*
⟹ *no-dup* (*trail S*)
⟹ *T* ∼ *prepend-trail* (*Propagated L* ()) ($reduce\text{-}trail\text{-}to_{NOT}$ *F* ($add\text{-}cls_{NOT}$ C'' *S*))
⟹ *C* ∈# $clauses_{NOT}$ *S*
⟹ *trail S* ⊨*as CNot C*
⟹ *undefined-lit F L*
⟹ *atm-of L* ∈ *atms-of-mm* ($clauses_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*trail S*))
⟹ $clauses_{NOT}$ *S* ⊨*pm* C' + {#*L*#}
⟹ *mset-cls* C'' = C' + {#*L*#}
⟹ *F* ⊨*as CNot* C'
⟹ *backjump-l-cond C C' L S T*
⟹ *backjump-l S T*

Avoid (meaningless) simplification in the theorem generated by *inductive-cases*:

declare $reduce\text{-}trail\text{-}to_{NOT}$*-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-to*$_{NOT}$*-length-ne*[*simp*] *Set.Un-iff*[*simp*] *Set.insert-iff*[*simp*]

inductive *cdcl*$_{NOT}$*-merged-bj-learn* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
cdcl$_{NOT}$*-merged-bj-learn-decide*$_{NOT}$: *decide*$_{NOT}$ *S S'* ⟹ *cdcl*$_{NOT}$*-merged-bj-learn S S'* |
cdcl$_{NOT}$*-merged-bj-learn-propagate*$_{NOT}$: *propagate*$_{NOT}$ *S S'* ⟹ *cdcl*$_{NOT}$*-merged-bj-learn S S'* |
cdcl$_{NOT}$*-merged-bj-learn-backjump-l*: *backjump-l S S'* ⟹ *cdcl*$_{NOT}$*-merged-bj-learn S S'* |
cdcl$_{NOT}$*-merged-bj-learn-forget*$_{NOT}$: *forget*$_{NOT}$ *S S'* ⟹ *cdcl*$_{NOT}$*-merged-bj-learn S S'*

lemma *cdcl*$_{NOT}$*-merged-bj-learn-no-dup-inv*:
  cdcl$_{NOT}$*-merged-bj-learn S T* ⟹ *no-dup* (*trail S*) ⟹ *no-dup* (*trail T*)
  ⟨*proof*⟩
end

locale *cdcl*$_{NOT}$*-merge-bj-learn-proxy* =
  cdcl$_{NOT}$*-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-cls$_{NOT}$ *remove-cls*$_{NOT}$ *propagate-conds*
    forget-cond
    $\lambda C\ C'\ L'\ S\ T.$ *backjump-l-cond C C' L' S T*
    ∧ *distinct-mset* (C' + {#L'#}) ∧ ¬*tautology* (C' + {#L'#})
  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*) *ann-lits* **and**
    raw-clauses :: *'st* ⇒ *'clss* **and**
    prepend-trail :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
    tl-trail :: *'st* ⇒*'st* **and**
    add-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
    remove-cls$_{NOT}$ :: *'cls* ⇒ *'st* ⇒ *'st* **and**
    propagate-conds :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
    forget-cond :: *'cls* ⇒ *'st* ⇒ *bool* **and**
    backjump-l-cond :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* +
  fixes
    inv :: *'st* ⇒ *bool*
  assumes
     bj-merge-can-jump:
     ⋀*S C F' K F L*.
       inv S
       ⟹ *trail S* = *F'* @ *Decided K* () # *F*
       ⟹ *C* ∈# *clauses*$_{NOT}$ *S*
       ⟹ *trail S* ⊨*as CNot C*
       ⟹ *undefined-lit F L*
       ⟹ *atm-of L* ∈ *atms-of-mm* (*clauses*$_{NOT}$ *S*) ∪ *atm-of* ‘ (*lits-of-l* (*F'* @ *Decided K* () # *F*))
       ⟹ *clauses*$_{NOT}$ *S* ⊨*pm C'* + {#*L*#}
       ⟹ *F* ⊨*as CNot C'*
       ⟹ ¬*no-step backjump-l S* **and**
     cdcl-merged-inv: ⋀*S T*. *cdcl*$_{NOT}$*-merged-bj-learn S T* ⟹ *inv S* ⟹ *inv T*

begin

abbreviation *backjump-conds* :: $'v$ *clause* $\Rightarrow$ $'v$ *clause* $\Rightarrow$ $'v$ *literal* $\Rightarrow$ $'st \Rightarrow 'st \Rightarrow$ *bool*
where
backjump-conds $\equiv \lambda C\ C'\ L'\ S\ T.$ *distinct-mset* $(C' + \{\#L'\#\}) \wedge \neg$*tautology* $(C' + \{\#L'\#\})$

Without additional knowledge on *backjump-l-cond*, it is impossible to have the same invariant.

sublocale *dpll-with-backjumping-ops 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
⟨*proof*⟩

end

locale *cdcl*$_{NOT}$*-merge-bj-learn-proxy2* =
cdcl$_{NOT}$*-merge-bj-learn-proxy 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 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*) *ann-lits* **and**
raw-clauses :: $'st \Rightarrow 'clss$ **and**
prepend-trail :: $('v,$ *unit*, *unit*) *ann-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*) *ann-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-cls$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds
λC -. *distinct-mset* (*mset-cls* C) $\wedge \neg$*tautology* (*mset-cls* C)
forget-cond
⟨*proof*⟩
end

locale *cdcl*$_{NOT}$*-merge-bj-learn* =
cdcl$_{NOT}$*-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-cls$_{NOT}$ *remove-cls*$_{NOT}$
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 :: $'st \Rightarrow ('v, unit, unit)$ *ann-lits* **and**
raw-clauses :: $'st \Rightarrow 'clss$ **and**
prepend-trail :: $('v, unit, unit)$ *ann-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**
backjump-l-cond :: $'v$ *clause* $\Rightarrow 'v$ *clause* $\Rightarrow 'v$ *literal* $\Rightarrow 'st \Rightarrow 'st \Rightarrow$ *bool* **and**
propagate-conds :: $('v, unit, unit)$ *ann-lit* $\Rightarrow 'st \Rightarrow$ *bool* **and**
forget-cond :: $'cls \Rightarrow 'st \Rightarrow$ *bool* **and**
inv :: $'st \Rightarrow$ *bool* +
assumes
dpll-merge-bj-inv: $\bigwedge S\ T.\ \ dpll\text{-}bj\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$ **and**
learn-inv: $\bigwedge S\ T.\ learn\ S\ T \Longrightarrow inv\ S \Longrightarrow inv\ T$
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-cls$_{NOT}$ *remove-cls*$_{NOT}$
inv backjump-conds propagate-conds
λC -. *distinct-mset* (*mset-cls* C) $\wedge$ $\neg$*tautology* (*mset-cls* 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 $\exists\, C'\ L\ D.$ *learn* S (*add-cls*$_{NOT}$ $D\ S$)
$\wedge$ *mset-cls* $D = (C' + \{\#L\#\})$
$\wedge$ *backjump* (*add-cls*$_{NOT}$ $D\ S$) T
$\wedge$ *atms-of* $(C' + \{\#L\#\}) \subseteq$ *atms-of-mm* (*clauses*$_{NOT}$ S) $\cup$ *atm-of* ‘ (*lits-of-l* (*trail* S))
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-merged-bj-learn-is-tranclp-cdcl*$_{NOT}$:
cdcl$_{NOT}$*-merged-bj-learn* $S\ T \Longrightarrow inv\ S \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow$ *cdcl*$_{NOT}{}^{++}$ $S\ T$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-is-rtranclp-cdcl*$_{NOT}$*-and-inv*:
cdcl$_{NOT}$*-merged-bj-learn*** $S\ T \Longrightarrow inv\ S \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow$ *cdcl*$_{NOT}{}^{**}$ $S\ T \wedge inv\ T$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-is-rtranclp-cdcl*$_{NOT}$:
cdcl$_{NOT}$*-merged-bj-learn*** $S\ T \Longrightarrow inv\ S \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow$ *cdcl*$_{NOT}{}^{**}$ $S\ T$
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-inv*:
cdcl$_{NOT}$*-merged-bj-learn*** $S\ T \Longrightarrow inv\ S \Longrightarrow$ *no-dup* (*trail* S) $\Longrightarrow inv\ T$

$\langle proof \rangle$

definition $\mu_C{}'$ :: *'v literal multiset set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_C{}'$ *A T* $\equiv$ μ_C (*1+card* (*atms-of-ms A*)) (*2+card* (*atms-of-ms A*)) (*trail-weight T*)

definition $\mu_{CDCL}{}'$*-merged* :: *'v literal multiset set* $\Rightarrow$ *'st* $\Rightarrow$ *nat* **where**
$\mu_{CDCL}{}'$*-merged A T* $\equiv$
((*2+card* (*atms-of-ms A*)) ^ (*1+card* (*atms-of-ms A*)) $-$ $\mu_C{}'$ *A T*) $*$ *2* + *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 $\mu_{CDCL}{}'$*-merged A T* $<$ $\mu_{CDCL}{}'$*-merged A S*
$\langle proof \rangle$

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*$_{NOT}$*-merged-bj-learn S T*}
$\langle proof \rangle$

lemma *tranclp-cdcl*$_{NOT}$*-cdcl*$_{NOT}$*-tranclp*:
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[*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*$_{NOT}$*-merged-bj-learn S T*}$^+$ (**is** - $\in$ *?P*$^+$)
$\langle proof \rangle$

lemma *wf-tranclp-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*$_{NOT}$*-merged-bj-learn*$^{++}$ *S T*}
$\langle proof \rangle$

lemma *backjump-no-step-backjump-l*:
backjump S T $\Longrightarrow$ *inv S* $\Longrightarrow$ $\neg$*no-step backjump-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) $\subseteq$ *atms-of-ms* A **and**
    atms-trail: *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
    n-d: *no-dup* (*trail* S) **and**
    finite A **and**
    inv: *inv* S **and**
    decomp: *all-decomposition-implies-m* (*clauses$_{NOT}$* S) (*get-all-ann-decomposition* (*trail* S))
  shows *unsatisfiable* (*set-mset* (*clauses$_{NOT}$* S))
    $\vee$ (*trail* S $\models$*asm* *clauses$_{NOT}$* S $\wedge$ *satisfiable* (*set-mset* (*clauses$_{NOT}$* S)))
⟨*proof*⟩

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) $\subseteq$ *atms-of-ms* A **and**
    atms-trail: *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
    n-d: *no-dup* (*trail* S) **and**
    finite A **and**
    inv: *inv* S **and**
    decomp: *all-decomposition-implies-m* (*clauses$_{NOT}$* S) (*get-all-ann-decomposition* (*trail* S))
  shows *unsatisfiable* (*set-mset* (*clauses$_{NOT}$* T))
    $\vee$ (*trail* T $\models$*asm* *clauses$_{NOT}$* T $\wedge$ *satisfiable* (*set-mset* (*clauses$_{NOT}$* T)))
⟨*proof*⟩

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*) *ann-lits* **and**
    raw-clauses :: *'st* $\Rightarrow$ *'clss* **and**
    prepend-trail :: (*'v*, *unit*, *unit*) *ann-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* ⇒ *'st* ⇒ *'st* **and**
inv :: *'st* ⇒ *bool* **and**
backjump-conds :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **and**
propagate-conds :: (*'v*, *unit*, *unit*) *ann-lit* ⇒ *'st* ⇒ *bool* **and**
learn-restrictions forget-restrictions :: *'cls* ⇒ *'st* ⇒ *bool*
+
fixes *f* :: *nat* ⇒ *nat*
assumes
unbounded: *unbounded f* **and** *f-ge-1*: $\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \geq 1$ **and**
inv-restart: $\bigwedge S\ T.$ *inv S* ⟹ *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::*'a list*) *S* ⟹ *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*) ⊆ *atms-of-ms A* **and**
atms-trail-S-A: *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* **and**
finite A **and**
cdcl$_{NOT}$: *cdcl*$_{NOT}$ *S T*
shows
atms-of-mm (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A* **and**
atm-of ' *lits-of-l* (*trail T*) ⊆ *atms-of-ms A* **and**
finite A
⟨*proof*⟩

sublocale *cdcl*$_{NOT}$*-increasing-restarts-ops* λ*S T*. *T* ∼ *reduce-trail-to*$_{NOT}$ ([]::*'a list*) *S cdcl*$_{NOT}$ *f*
λ*A S*. *atms-of-mm* (*clauses*$_{NOT}$ *S*) ⊆ *atms-of-ms A* ∧ *atm-of* ' *lits-of-l* (*trail S*) ⊆ *atms-of-ms A* ∧
finite A
μ_{CDCL}' λ*S*. *inv S* ∧ *no-dup* (*trail S*)
μ_{CDCL}'*-bound*
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-with-restart-*μ_{CDCL}'*-le-*μ_{CDCL}'*-bound*:
assumes
cdcl$_{NOT}$: *cdcl*$_{NOT}$*-restart* (*T*, *a*) (*V*, *b*) **and**
cdcl$_{NOT}$*-inv*:
inv T
no-dup (*trail T*) **and**
bound-inv:
atms-of-mm (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A*
atm-of ' *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*
finite A
shows μ_{CDCL}' *A V* ≤ μ_{CDCL}'*-bound A T*
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-with-restart-*μ_{CDCL}'*-bound-le-*μ_{CDCL}'*-bound*:
assumes
cdcl$_{NOT}$: *cdcl*$_{NOT}$*-restart* (*T*, *a*) (*V*, *b*) **and**
cdcl$_{NOT}$*-inv*:
inv T
no-dup (*trail T*) **and**
bound-inv:
atms-of-mm (*clauses*$_{NOT}$ *T*) ⊆ *atms-of-ms A*
atm-of ' *lits-of-l* (*trail T*) ⊆ *atms-of-ms A*

finite A
shows μ_{CDCL}'*-bound A V* $\leq$ μ_{CDCL}'*-bound A T*
⟨*proof*⟩

sublocale $cdcl_{NOT}$*-increasing-restarts* - - - - - - - - - - - - - -
f
$\lambda S\ T.\ T \sim$ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) *S*
$\lambda A\ S.$ *atms-of-mm* (*clauses*$_{NOT}$ *S*) $\subseteq$ *atms-of-ms A*
$\wedge$ *atm-of* ' *lits-of-l* (*trail S*) $\subseteq$ *atms-of-ms A* $\wedge$ *finite A*
μ_{CDCL}' $cdcl_{NOT}$
$\lambda S.$ *inv S* $\wedge$ *no-dup* (*trail S*)
μ_{CDCL}'*-bound*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-all-decomposition-implies*:
assumes $cdcl_{NOT}$*-restart S T* **and**
inv (*fst S*) **and**
no-dup (*trail* (*fst S*))
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst S*)) (*get-all-ann-decomposition* (*trail* (*fst S*)))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst T*)) (*get-all-ann-decomposition* (*trail* (*fst T*)))
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-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-ann-decomposition* (*trail* (*fst S*)))
shows
all-decomposition-implies-m (*clauses*$_{NOT}$ (*fst T*)) (*get-all-ann-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* $\models$*sextm clauses*$_{NOT}$ (*fst S*) $\longleftrightarrow$ *I* $\models$*sextm clauses*$_{NOT}$ (*fst T*)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-restart-sat-ext-iff*:
fixes *S T* :: '*st* $\times$ *nat*
assumes
st: $cdcl_{NOT}$*-restart*** *S T* **and**
n-d: *no-dup* (*trail* (*fst S*)) **and**
inv: *inv* (*fst S*)
shows *I* $\models$*sextm clauses*$_{NOT}$ (*fst S*) $\longleftrightarrow$ *I* $\models$*sextm clauses*$_{NOT}$ (*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*) $\subseteq$ *atms-of-ms A* **and**

atms-trail: *atm-of* ' *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
fin-A[*simp*]: *finite* A **and**
inv: *inv* S **and**
decomp: *all-decomposition-implies-m* ($clauses_{NOT}$ S) (*get-all-ann-decomposition* (*trail* S))
shows *unsatisfiable* (*set-mset* ($clauses_{NOT}$ S))
$\vee$ (*lits-of-l* (*trail* T) $\models$*sextm* $clauses_{NOT}$ S $\wedge$ *satisfiable* (*set-mset* ($clauses_{NOT}$ S)))
⟨*proof*⟩
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 $add\text{-}cls_{NOT}$ $remove\text{-}cls_{NOT}$
$\lambda C\ C'\ L'\ S\ T.$ *distinct-mset* $(C' + \{\#L'\#\})$ $\wedge$ *backjump-l-cond* $C\ C'\ L'\ S\ T$
propagate-conds forget-conds 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*) *ann-lits* **and**
raw-clauses :: $'st \Rightarrow\ 'clss$ **and**
prepend-trail :: ($'v$, *unit*, *unit*) *ann-lit* $\Rightarrow\ 'st \Rightarrow\ 'st$ **and**
tl-trail :: $'st \Rightarrow 'st$ **and**
$add\text{-}cls_{NOT}$ :: $'cls \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
$remove\text{-}cls_{NOT}$ :: $'cls \Rightarrow\ 'st \Rightarrow\ 'st$ **and**
propagate-conds :: ($'v$, *unit*, *unit*) *ann-lit* $\Rightarrow\ 'st \Rightarrow$ *bool* **and**
inv :: $'st \Rightarrow$ *bool* **and**
forget-conds :: $'cls \Rightarrow\ 'st \Rightarrow$ *bool* **and**
backjump-l-cond :: $'v$ *clause* $\Rightarrow\ 'v$ *clause* $\Rightarrow\ 'v$ *literal* $\Rightarrow\ 'st \Rightarrow\ 'st \Rightarrow$ *bool*
+
fixes f :: *nat* $\Rightarrow$ *nat*
assumes
unbounded: *unbounded* f **and** *f-ge-1*: $\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \geq 1$ **and**
inv-restart:$\bigwedge S\ T.$ *inv* $S \Longrightarrow T \sim$ $reduce\text{-}trail\text{-}to_{NOT}$ [] $S \Longrightarrow$ *inv* T
begin

definition *not-simplified-cls* $A = \{\#C \in\# A.$ *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)
⟨*proof*⟩

lemma $cdcl_{NOT}$*-merged-bj-learn-clauses-bound*:
assumes
$cdcl_{NOT}$*-merged-bj-learn* S T **and**

inv: *inv* S **and**
atms-clss: *atms-of-mm* (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atms-trail: *atm-of* '(*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
fin-A[*simp*]: *finite* A
shows *set-mset* (*clauses*$_{NOT}$ T) $\subseteq$ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ S))
$\cup$ *simple-clss* (*atms-of-ms* A)
⟨*proof*⟩

lemma *cdcl*$_{NOT}$*-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl*$_{NOT}$*-merged-bj-learn* S T
shows (*not-simplified-cls* (*clauses*$_{NOT}$ T)) $\subseteq$# (*not-simplified-cls* (*clauses*$_{NOT}$ S))
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-not-simplified-decreasing*:
assumes *cdcl*$_{NOT}$*-merged-bj-learn*** S T
shows (*not-simplified-cls* (*clauses*$_{NOT}$ T)) $\subseteq$# (*not-simplified-cls* (*clauses*$_{NOT}$ S))
⟨*proof*⟩

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound*:
assumes
cdcl$_{NOT}$*-merged-bj-learn*** S T **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atm-of '(*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
finite[*simp*]: *finite* A
shows *set-mset* (*clauses*$_{NOT}$ T) $\subseteq$ *set-mset* (*not-simplified-cls* (*clauses*$_{NOT}$ S))
$\cup$ *simple-clss* (*atms-of-ms* A)
⟨*proof*⟩

abbreviation μ_{CDCL}'*-bound* **where**
μ_{CDCL}'*-bound* A T $\equiv$ ((*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A))) $*$ *2*
+ *card* (*set-mset* (*not-simplified-cls*(*clauses*$_{NOT}$ T)))
+ *3* ^ *card* (*atms-of-ms* A)

lemma *rtranclp-cdcl*$_{NOT}$*-merged-bj-learn-clauses-bound-card*:
assumes
cdcl$_{NOT}$*-merged-bj-learn*** S T **and**
inv S **and**
atms-of-mm (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A **and**
atm-of '(*lits-of-l* (*trail* S)) $\subseteq$ *atms-of-ms* A **and**
n-d: *no-dup* (*trail* S) **and**
finite: *finite* A
shows μ_{CDCL}'*-merged* A T $\leq$ μ_{CDCL}'*-bound* A S
⟨*proof*⟩

sublocale *cdcl*$_{NOT}$*-increasing-restarts-ops* λS T. $T \sim$ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) S
cdcl$_{NOT}$*-merged-bj-learn f*
λA S. *atms-of-mm* (*clauses*$_{NOT}$ S) $\subseteq$ *atms-of-ms* A
$\wedge$ *atm-of* ' *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A $\wedge$ *finite* A
μ_{CDCL}'*-merged*
λS. *inv* S $\wedge$ *no-dup* (*trail* S)
μ_{CDCL}'*-bound*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-*$\mu_{CDCL}{}'$*-merged-le-*$\mu_{CDCL}{}'$*-bound*:
  assumes
    $cdcl_{NOT}$*-restart* T V
    inv (*fst* T) **and**
    no-dup (*trail* (*fst* T)) **and**
    atms-of-mm ($clauses_{NOT}$ (*fst* T)) $\subseteq$ *atms-of-ms* A **and**
    atm-of ‘ *lits-of-l* (*trail* (*fst* T)) $\subseteq$ *atms-of-ms* A **and**
    finite A
  shows $\mu_{CDCL}{}'$*-merged* A (*fst* V) $\leq$ $\mu_{CDCL}{}'$*-bound* A (*fst* T)
  ⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-*$\mu_{CDCL}{}'$*-bound-le-*$\mu_{CDCL}{}'$*-bound*:
  assumes
    $cdcl_{NOT}$*-restart* T V **and**
    no-dup (*trail* (*fst* T)) **and**
    inv (*fst* T) **and**
    fin: *finite* A
  shows $\mu_{CDCL}{}'$*-bound* A (*fst* V) $\leq$ $\mu_{CDCL}{}'$*-bound* A (*fst* T)
  ⟨*proof*⟩

sublocale $cdcl_{NOT}$*-increasing-restarts* - - - - - - - - - - - - - - *f*
   λS T. $T \sim$ *reduce-trail-to*$_{NOT}$ ([]::'*a list*) S
   λA S. *atms-of-mm* ($clauses_{NOT}$ S) $\subseteq$ *atms-of-ms* A
     $\wedge$ *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-ms* A $\wedge$ *finite* A
   $\mu_{CDCL}{}'$*-merged* $cdcl_{NOT}$*-merged-bj-learn*
    λS. *inv* S $\wedge$ *no-dup* (*trail* S)
   λA T. ((*2*+*card* (*atms-of-ms* A)) ^ (*1*+*card* (*atms-of-ms* A))) $*$ *2*
     + *card* (*set-mset* (*not-simplified-cls*($clauses_{NOT}$ T)))
     + *3* ^ *card* (*atms-of-ms* A)
   ⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-eq-sat-iff*:
  assumes
    $cdcl_{NOT}$*-restart* S T **and**
    no-dup (*trail* (*fst* S))
    inv (*fst* S)
  shows $I$$\models$*sextm* $clauses_{NOT}$ (*fst* S) $\longleftrightarrow$ I $\models$*sextm* $clauses_{NOT}$ (*fst* T)
  ⟨*proof*⟩

lemma *rtranclp-*$cdcl_{NOT}$*-restart-eq-sat-iff*:
  assumes
    $cdcl_{NOT}$*-restart*** S T **and**
    inv: *inv* (*fst* S) **and** *n-d*: *no-dup*(*trail* (*fst* S))
  shows $I$$\models$*sextm* $clauses_{NOT}$ (*fst* S) $\longleftrightarrow$ I $\models$*sextm* $clauses_{NOT}$ (*fst* T)
  ⟨*proof*⟩

lemma $cdcl_{NOT}$*-restart-all-decomposition-implies-m*:
  assumes
    $cdcl_{NOT}$*-restart* S T **and**
    inv: *inv* (*fst* S) **and** *n-d*: *no-dup*(*trail* (*fst* S)) **and**
    all-decomposition-implies-m ($clauses_{NOT}$ (*fst* S))
      (*get-all-ann-decomposition* (*trail* (*fst* S)))
  shows *all-decomposition-implies-m* ($clauses_{NOT}$ (*fst* T))

$(get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ (fst\ T)))$
$\langle proof \rangle$

lemma *rtranclp-cdcl$_{NOT}$-restart-all-decomposition-implies-m*:
assumes
$cdcl_{NOT}\text{-}restart^{**}\ S\ T$ **and**
inv: $inv\ (fst\ S)$ **and** *n-d*: $no\text{-}dup(trail\ (fst\ S))$ **and**
decomp: $all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ (fst\ S))$
$(get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ (fst\ S)))$
shows $all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ (fst\ T))$
$(get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ (fst\ T)))$
$\langle proof \rangle$

lemma *full-cdcl$_{NOT}$-restart-normal-form*:
assumes
full: $full\ cdcl_{NOT}\text{-}restart\ S\ T$ **and**
inv: $inv\ (fst\ S)$ **and** *n-d*: $no\text{-}dup(trail\ (fst\ S))$ **and**
decomp: $all\text{-}decomposition\text{-}implies\text{-}m\ (clauses_{NOT}\ (fst\ S))$
$(get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ (fst\ S)))$ **and**
atms-cls: $atms\text{-}of\text{-}mm\ (clauses_{NOT}\ (fst\ S)) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
atms-trail: $atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ (fst\ S)) \subseteq atms\text{-}of\text{-}ms\ A$ **and**
fin: $finite\ A$
shows $unsatisfiable\ (set\text{-}mset\ (clauses_{NOT}\ (fst\ S)))$
$\lor\ lits\text{-}of\text{-}l\ (trail\ (fst\ T)) \models sextm\ clauses_{NOT}\ (fst\ S) \land satisfiable\ (set\text{-}mset\ (clauses_{NOT}\ (fst\ S)))$
$\langle proof \rangle$

corollary *full-cdcl$_{NOT}$-restart-normal-form-init-state*:
assumes
init-state: $trail\ S = []\ clauses_{NOT}\ S = N$ **and**
full: $full\ cdcl_{NOT}\text{-}restart\ (S,\ 0)\ T$ **and**
inv: $inv\ S$
shows $unsatisfiable\ (set\text{-}mset\ N)$
$\lor\ lits\text{-}of\text{-}l\ (trail\ (fst\ T)) \models sextm\ N \land satisfiable\ (set\text{-}mset\ N)$
$\langle proof \rangle$

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 $backtrack :: ('v,\ unit,\ unit)\ ann\text{-}lit\ list \times 'v\ clauses$
$\Rightarrow ('v,\ unit,\ unit)\ ann\text{-}lit\ list \times 'v\ clauses \Rightarrow bool$ **where**
$backtrack\text{-}split\ (fst\ S) = (M',\ L\ \#\ M) \Longrightarrow is\text{-}decided\ L \Longrightarrow D \in\#\ snd\ S$
$\Longrightarrow fst\ S \models as\ CNot\ D \Longrightarrow backtrack\ S\ (Propagated\ (-\ (lit\text{-}of\ L))\ ()\ \#\ M,\ snd\ S)$

inductive-cases *backtrackE*[*elim*]: $backtrack\ (M,\ N)\ (M',\ N')$

lemma *backtrack-is-backjump*:
  fixes $M\ M'$ :: ('v, unit, unit) ann-lit list
  assumes
    backtrack: *backtrack* (M, N) (M', N') **and**
    no-dup: (*no-dup* ∘ *fst*) (M, N) **and**
    decomp: *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
    shows
      $\exists\, C\ F'\ K\ F\ L\ l\ C'.$
        $M = F'$ @ *Decided* K () # F ∧
        M' = *Propagated* $L\ l$ # F ∧ $N = N'$ ∧ C ∈# N ∧ F' @ *Decided* $K\ d$ # F ⊨as *CNot* C ∧
        undefined-lit $F\ L$ ∧ *atm-of* L ∈ *atms-of-mm* N ∪ *atm-of* ‘ *lits-of-l* (F' @ *Decided* $K\ d$ # F) ∧
        N ⊨pm C' + {#L#} ∧ F ⊨as *CNot* C'
⟨*proof*⟩

lemma *backtrack-is-backjump'*:
  fixes $M\ M'$ :: ('v, unit, unit) ann-lit list
  assumes
    backtrack: *backtrack* $S\ T$ **and**
    no-dup: (*no-dup* ∘ *fst*) S **and**
    decomp: *all-decomposition-implies-m* (*snd* S) (*get-all-ann-decomposition* (*fst* S))
    shows
      $\exists\, C\ F'\ K\ F\ L\ l\ C'.$
        fst $S = F'$ @ *Decided* K () # F ∧
        T = (*Propagated* $L\ l$ # F, *snd* S) ∧ C ∈# *snd* S ∧ *fst* S ⊨as *CNot* C
        ∧ *undefined-lit* $F\ L$ ∧ *atm-of* L ∈ *atms-of-mm* (*snd* S) ∪ *atm-of* ‘ *lits-of-l* (*fst* S) ∧
        snd S ⊨pm C' + {#L#} ∧ F ⊨as *CNot* C'
  ⟨*proof*⟩

sublocale *dpll-state*
  id λL C. C + {#L#} *remove1-mset*
  id op + op ∈# λL C. C + {#L#} *remove1-mset*
  fst snd λL (M, N). (L # M, N) λ(M, N). (*tl* M, N)
  λC (M, N). (M, {#C#} + N) λC (M, N). (M, *removeAll-mset* C N)
  ⟨*proof*⟩

sublocale *backjumping-ops*
  id λL C. C + {#L#} *remove1-mset*
  id op + op ∈# λL C. C + {#L#} *remove1-mset*
  fst snd λL (M, N). (L # M, N) λ(M, N). (*tl* M, N)
  λC (M, N). (M, {#C#} + N) λC (M, N). (M, *removeAll-mset* C N) λ- - - S T. *backtrack* S T
  ⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$*-snd*:
  snd (*reduce-trail-to*$_{NOT}$ $F\ S$) = *snd* S
  ⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$:
  reduce-trail-to$_{NOT}$ $F\ S$ =
    (*if length* (*fst* S) ≥ *length* F
    then drop (*length* (*fst* S) − *length* F) (*fst* S)
    else [],
    snd S) (**is** ?R = ?C)
⟨*proof*⟩

lemma *backtrack-is-backjump''*:

fixes M M' :: $('v, unit, unit)$ *ann-lit list*
assumes
backtrack: *backtrack* S T **and**
no-dup: $(no\text{-}dup \circ fst)$ S **and**
decomp: *all-decomposition-implies-m* (*snd* S) (*get-all-ann-decomposition* (*fst* S))
shows *backjump* S T
⟨*proof*⟩

lemma *can-do-bt-step*:
assumes
M: *fst* $S = F'$ @ *Decided* K d # F **and**
$C \in\#$ *snd* S **and**
C: *fst* $S \models as$ *CNot* C
shows ¬ *no-step backtrack* S
⟨*proof*⟩

end

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping-ops*
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* $C\ N)$
$\lambda(M, N).$ *no-dup* M ∧ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True*
⟨*proof*⟩

sublocale *dpll-with-backtrack* ⊆ *dpll-with-backjumping*
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* $C\ N)$
$\lambda(M, N).$ *no-dup* M ∧ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True*
⟨*proof*⟩

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* $\{((M'::('v, unit, unit)\ ann\text{-}lits, N'::'v\ clauses), ([], N))|M'\ N'\ N.$
$dpll\text{-}bj^{++}$ $([], N)$ (M', N') ∧ *atms-of-mm* $N \subseteq$ *atms-of-ms* $A\}$
⟨*proof*⟩

corollary *full-dpll-final-state-conclusive*:
fixes M M' :: $('v, unit, unit)$ *ann-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))
⟨*proof*⟩

corollary *full-dpll-normal-form-from-init-state*:
fixes M M' :: ('*v*, *unit*, *unit*) *ann-lit list*
assumes
full: *full dpll-bj* ([], N) (M', N')
shows M' $\models$*asm* N $\longleftrightarrow$ *satisfiable* (*set-mset* N)
⟨*proof*⟩

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)
λ(M, N). (*tl* M, N) λC (M, N). (M, {#C#} + N) λC (M, N). (M, *removeAll-mset* C N)
λ(M, N). *no-dup* M $\wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True* λ- -. *False* λ- -. *False*
⟨*proof*⟩

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)
λ(M, N). (*tl* M, N) λC (M, N). (M, {#C#} + N) λC (M, N). (M, *removeAll-mset* C N)
λ(M, N). *no-dup* M $\wedge$ *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
λ- - - S T. *backtrack* S T
λ- -. *True* λ- -. *False* λ- -. *False*
⟨*proof*⟩

lemma $cdcl_{NOT}$*-is-dpll*:
$cdcl_{NOT}$ S T $\longleftrightarrow$ *dpll-bj* S T
⟨*proof*⟩

Another proof of termination:

lemma *wf* {(T, S). *dpll-bj* S T $\wedge$ $cdcl_{NOT}$*-NOT-all-inv* A S}
⟨*proof*⟩
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 **and** *f-ge-1*:$\bigwedge n.\ n \geq 1 \Longrightarrow f\ n \geq 1$
begin
sublocale $cdcl_{NOT}$*-increasing-restarts*
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) λ(M, N). (*tl* M, N)
λC (M, N). (M, {#C#} + N) λC (M, N). (M, *removeAll-mset* C N) f λ(-, N) S. S = ([], N)
λA (M, N). *atms-of-mm* N $\subseteq$ *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-ann-decomposition* M)

$\lambda A\ T.\ (2+card\ (atms\text{-}of\text{-}ms\ A))\ \widehat{}\ (1+card\ (atms\text{-}of\text{-}ms\ A))$
$\quad - \mu_C\ (1+card\ (atms\text{-}of\text{-}ms\ A))\ (2+card\ (atms\text{-}of\text{-}ms\ A))\ (trail\text{-}weight\ T)\ dpll\text{-}bj$
$\lambda(M,\ N).\ no\text{-}dup\ M \wedge all\text{-}decomposition\text{-}implies\text{-}m\ N\ (get\text{-}all\text{-}ann\text{-}decomposition\ M)$
$\lambda A\ \text{-}.\ (2+card\ (atms\text{-}of\text{-}ms\ A))\ \widehat{}\ (1+card\ (atms\text{-}of\text{-}ms\ A))$
⟨*proof*⟩
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 $'a\ dpll_W\text{-}ann\text{-}lit = ('a,\ unit,\ unit)\ ann\text{-}lit$
type-synonym $'a\ dpll_W\text{-}ann\text{-}lits = ('a,\ unit,\ unit)\ ann\text{-}lits$
type-synonym $'v\ dpll_W\text{-}state = 'v\ dpll_W\text{-}ann\text{-}lits \times 'v\ clauses$

abbreviation $trail :: 'v\ dpll_W\text{-}state \Rightarrow 'v\ dpll_W\text{-}ann\text{-}lits$ **where**
$trail \equiv fst$
abbreviation $clauses :: 'v\ dpll_W\text{-}state \Rightarrow 'v\ clauses$ **where**
$clauses \equiv snd$

inductive $dpll_W :: 'v\ dpll_W\text{-}state \Rightarrow 'v\ dpll_W\text{-}state \Rightarrow bool$ **where**
$propagate$: $C + \{\#L\#\} \in\# clauses\ S \Longrightarrow trail\ S \models as\ CNot\ C \Longrightarrow undefined\text{-}lit\ (trail\ S)\ L$
$\Longrightarrow dpll_W\ S\ (Propagated\ L\ ()\ \#\ trail\ S,\ clauses\ S)$ |
$decided$: $undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (clauses\ S)$
$\Longrightarrow dpll_W\ S\ (Decided\ L\ ()\ \#\ trail\ S,\ clauses\ S)$ |
$backtrack$: $backtrack\text{-}split\ (trail\ S) = (M',\ L\ \#\ M) \Longrightarrow is\text{-}decided\ L \Longrightarrow D \in\# clauses\ S$
$\Longrightarrow trail\ S \models as\ CNot\ D \Longrightarrow dpll_W\ S\ (Propagated\ (-\ (lit\text{-}of\ L))\ ()\ \#\ M,\ clauses\ S)$

18.2 Invariants

lemma $dpll_W\text{-}distinct\text{-}inv$:
assumes $dpll_W\ S\ S'$
and $no\text{-}dup\ (trail\ S)$
shows $no\text{-}dup\ (trail\ S')$
⟨*proof*⟩

lemma $dpll_W\text{-}consistent\text{-}interp\text{-}inv$:
assumes $dpll_W\ S\ S'$
and $consistent\text{-}interp\ (lits\text{-}of\text{-}l\ (trail\ S))$
and $no\text{-}dup\ (trail\ S)$
shows $consistent\text{-}interp\ (lits\text{-}of\text{-}l\ (trail\ S'))$
⟨*proof*⟩

lemma $dpll_W\text{-}vars\text{-}in\text{-}snd\text{-}inv$:
assumes $dpll_W\ S\ S'$
and $atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S)) \subseteq atms\text{-}of\text{-}mm\ (clauses\ S)$
shows $atm\text{-}of\ `\ (lits\text{-}of\text{-}l\ (trail\ S')) \subseteq atms\text{-}of\text{-}mm\ (clauses\ S')$
⟨*proof*⟩

lemma *atms-of-ms-lit-of-atms-of*: *atms-of-ms* $((\lambda a.\ \{\#\mathit{lit\text{-}of}\ a\#\})$ ‘ $c)$ = *atm-of* ‘ *lit-of* ‘ c
⟨*proof*⟩

theorem 2.8.2 page 73 of Weidenbach's book

lemma dpll_W*-propagate-is-conclusion*:
assumes dpll_W S S'
and *all-decomposition-implies-m* (*clauses* S) (*get-all-ann-decomposition* (*trail* S))
and *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-mm* (*clauses* S)
shows *all-decomposition-implies-m* (*clauses* S') (*get-all-ann-decomposition* (*trail* S'))
⟨*proof*⟩

theorem 2.8.3 page 73 of Weidenbach's book

theorem dpll_W*-propagate-is-conclusion-of-decided*:
assumes dpll_W S S'
and *all-decomposition-implies-m* (*clauses* S) (*get-all-ann-decomposition* (*trail* S))
and *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-mm* (*clauses* S)
shows *set-mset* (*clauses* S') $\cup$ $\{\{\#\mathit{lit\text{-}of}\ L\#\}\ |L.\ \mathit{is\text{-}decided}\ L \wedge L \in \mathit{set}\ (\mathit{trail}\ S')\}$
$\models ps$ $(\lambda a.\ \{\#\mathit{lit\text{-}of}\ a\#\})$ ‘ $\bigcup$(*set* ‘ *snd* ‘ *set* (*get-all-ann-decomposition* (*trail* S')))
⟨*proof*⟩

theorem 2.8.4 page 73 of Weidenbach's book

lemma *only-propagated-vars-unsat*:
assumes *decided*: $\forall x \in \mathit{set}\ M.\ \neg\ \mathit{is\text{-}decided}\ x$
and *DN*: $D \in N$ **and** *D*: $M \models as$ *CNot* D
and *inv*: *all-decomposition-implies* N (*get-all-ann-decomposition* M)
and *atm-incl*: *atm-of* ‘ *lits-of-l* M $\subseteq$ *atms-of-ms* N
shows *unsatisfiable* N
⟨*proof*⟩

lemma dpll_W*-same-clauses*:
assumes dpll_W S S'
shows *clauses* S = *clauses* S'
⟨*proof*⟩

lemma *rtranclp-*dpll_W*-inv*:
assumes *rtranclp* dpll_W S S'
and *inv*: *all-decomposition-implies-m* (*clauses* S) (*get-all-ann-decomposition* (*trail* S))
and *atm-incl*: *atm-of* ‘ *lits-of-l* (*trail* S) $\subseteq$ *atms-of-mm* (*clauses* S)
and *consistent-interp* (*lits-of-l* (*trail* S))
and *no-dup* (*trail* S)
shows *all-decomposition-implies-m* (*clauses* S') (*get-all-ann-decomposition* (*trail* S'))
and *atm-of* ‘ *lits-of-l* (*trail* S') $\subseteq$ *atms-of-mm* (*clauses* S')
and *clauses* S = *clauses* S'
and *consistent-interp* (*lits-of-l* (*trail* S'))
and *no-dup* (*trail* S')
⟨*proof*⟩

definition dpll_W*-all-inv* S $\equiv$
(*all-decomposition-implies-m* (*clauses* S) (*get-all-ann-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-ann-decomposition* (*trail S*))
and *atm-of* ‘ *lits-of-l* (*trail S*) $\subseteq$ *atms-of-mm* (*clauses S*)
and *consistent-interp* (*lits-of-l* (*trail S*)) $\wedge$ *no-dup* (*trail S*)
⟨*proof*⟩

lemma *rtranclp-dpll$_W$-all-inv*:
assumes *rtranclp dpll$_W$ S S'*
and *dpll$_W$-all-inv S*
shows *dpll$_W$-all-inv S'*
⟨*proof*⟩

lemma *dpll$_W$-all-inv*:
assumes *dpll$_W$ S S'*
and *dpll$_W$-all-inv S*
shows *dpll$_W$-all-inv S'*
⟨*proof*⟩

lemma *rtranclp-dpll$_W$-inv-starting-from-0*:
assumes *rtranclp dpll$_W$ S S'*
and *inv*: *trail S* = []
shows *dpll$_W$-all-inv S'*
⟨*proof*⟩

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. *Decided M* ()) *M*, *N*)
⟨*proof*⟩

definition *conclusive-dpll$_W$-state* (*S*:: *'v dpll$_W$-state*) $\longleftrightarrow$
(*trail S* $\models$*asm clauses S* $\vee$ (($\forall L \in$ *set* (*trail S*). $\neg$*is-decided L*)
$\wedge$ ($\exists C \in\#$ *clauses S*. *trail S* $\models$*as CNot C*)))

theorem 2.8.6 page 74 of Weidenbach's book

lemma *dpll$_W$-strong-completeness*:
assumes *set M* $\models$*sm N*
and *consistent-interp* (*set M*)
and *distinct M*
and *atm-of* ‘ (*set M*) $\subseteq$ *atms-of-mm N*
shows *dpll$_W$*** ([], *N*) (*map* (λM. *Decided M* ()) *M*, *N*)
and *conclusive-dpll$_W$-state* (*map* (λM. *Decided M* ()) *M*, *N*)
⟨*proof*⟩

theorem 2.8.5 page 73 of Weidenbach's book

lemma *dpll$_W$-sound*:
assumes
rtranclp dpll$_W$ ([], *N*) (*M*, *N*) **and**
$\forall$ *S*. $\neg$*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 ($\lambda l.$ *if is-decided 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*))
$\in$ *lexn* $\{(a, b).\ a < b\}$ (*card vars*)
⟨*proof*⟩

theorem 2.8.7 page 74 of Weidenbach's book

lemma *dpll$_W$-card-decrease′*:
assumes *dpll*: *dpll$_W$ S S′*
and *atm-incl*: *atm-of ‘ lits-of-l* (*trail S*) $\subseteq$ *atms-of-mm* (*clauses S*)
and *no-dup*: *no-dup* (*trail S*)
shows (*dpll$_W$-mes* (*trail S′*) (*card* (*atms-of-mm* (*clauses S′*))),
dpll$_W$-mes (*trail S*) (*card* (*atms-of-mm* (*clauses S*)))) $\in$ *lex* $\{(a, b).\ a < b\}$
⟨*proof*⟩

lemma *wf-lexn*: *wf* (*lexn* $\{(a, b).\ (a{::}nat) < b\}$ (*card* (*atms-of-mm* (*clauses S*))))
⟨*proof*⟩

lemma *dpll$_W$-wf*:
wf $\{(S', S).$ *dpll$_W$-all-inv S* $\wedge$ *dpll$_W$ S S′*$\}$
⟨*proof*⟩

lemma *dpll$_W$-tranclp-star-commute*:
$\{(S', S).$ *dpll$_W$-all-inv S* $\wedge$ *dpll$_W$ S S′*$\}^+ = \{(S', S).$ *dpll$_W$-all-inv S* $\wedge$ *tranclp dpll$_W$ S S′*$\}$
(**is** *?A* = *?B*)
⟨*proof*⟩

lemma *dpll$_W$-wf-tranclp*: *wf* $\{(S', S).$ *dpll$_W$-all-inv S* $\wedge$ *dpll$_W$*$^{++}$ *S S′*$\}$
⟨*proof*⟩

lemma *dpll$_W$-wf-plus*:
shows *wf* $\{(S', ([], N))|\ S'.$ *dpll$_W$*$^{++}$ ($[]$, *N*) *S′*$\}$ (**is** *wf ?P*)
⟨*proof*⟩

18.4 Final States

lemma *dpll$_W$-no-more-step-is-a-conclusive-state*:
assumes $\forall S'.\ \neg$*dpll$_W$ S S′*
shows *conclusive-dpll$_W$-state S*
⟨*proof*⟩

lemma *dpll$_W$-conclusive-state-correct*:

assumes $dpll_W{}^{**}$ ([], N) (M, N) **and** *conclusive-dpll*$_W$*-state* (M, N)
shows M $\models$*asm* N $\longleftrightarrow$ *satisfiable* (*set-mset* N) (**is** *?A* $\longleftrightarrow$ *?B*)
⟨*proof*⟩

18.5 Link with NOT's DPLL

interpretation *dpll*$_W$*-*$_{NOT}$: *dpll-with-backtrack* ⟨*proof*⟩

declare *dpll*$_W$*-*$_{NOT}$*.state-simp*$_{NOT}$[*simp del*]
lemma *state-eq*$_{NOT}$*-iff-eq*[*iff*, *simp*]: *dpll*$_W$*-*$_{NOT}$*.state-eq*$_{NOT}$ S T $\longleftrightarrow$ $S = T$
⟨*proof*⟩
lemma *dpll*$_W$*-dpll*$_W$*-bj*:
assumes *inv*: *dpll*$_W$*-all-inv* S **and** *dpll*: *dpll*$_W$ S T
shows *dpll*$_W$*-*$_{NOT}$*.dpll-bj* S T
⟨*proof*⟩

lemma *dpll*$_W$*-bj-dpll*:
assumes *inv*: *dpll*$_W$*-all-inv* S **and** *dpll*: *dpll*$_W$*-*$_{NOT}$*.dpll-bj* S T
shows *dpll*$_W$ S T
⟨*proof*⟩

lemma *rtranclp-dpll*$_W$*-rtranclp-dpll*$_W$*-*$_{NOT}$:
assumes $dpll_W{}^{**}$ S T **and** *dpll*$_W$*-all-inv* S
shows *dpll*$_W$*-*$_{NOT}$*.dpll-bj*** S T
⟨*proof*⟩

lemma *rtranclp-dpll-rtranclp-dpll*$_W$:
assumes *dpll*$_W$*-*$_{NOT}$*.dpll-bj*** S T **and** *dpll*$_W$*-all-inv* S
shows $dpll_W{}^{**}$ S T
⟨*proof*⟩

lemma *dpll-conclusive-state-correctness*:
assumes *dpll*$_W$*-*$_{NOT}$*.dpll-bj*** ([], N) (M, N) **and** *conclusive-dpll*$_W$*-state* (M, N)
shows M $\models$*asm* N $\longleftrightarrow$ *satisfiable* (*set-mset* N)
⟨*proof*⟩

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*) *ann-lits* ⇒ *nat* ⇒ '*v literal* ⇒ *nat* **where**
get-rev-level [] - - = *0* |
get-rev-level (*Decided 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* ≡ *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*

⟨*proof*⟩

lemma *atm-of-notin-get-rev-level-eq-0*:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
shows $get\text{-}rev\text{-}level\ M\ n\ L = 0$
⟨*proof*⟩

lemma *get-rev-level-ge-0-atm-of-in*:
assumes $get\text{-}rev\text{-}level\ M\ n\ L > n$
shows $atm\text{-}of\ L \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
⟨*proof*⟩

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 $atm\text{-}of\ L \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
shows $get\text{-}rev\text{-}level\ (M\ @\ Decided\ K\ i\ \#\ M')\ n\ L = get\text{-}rev\text{-}level\ (Decided\ K\ i\ \#\ M')\ i\ L$
⟨*proof*⟩

lemma *get-rev-level-notin-end*[*simp*]:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M'$
shows $get\text{-}rev\text{-}level\ (M\ @\ M')\ n\ L = get\text{-}rev\text{-}level\ M\ n\ L$
⟨*proof*⟩

If the literal is at the beginning, then the end can be skipped

lemma *get-rev-level-skip-end*[*simp*]:
assumes $atm\text{-}of\ L \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M$
shows $get\text{-}rev\text{-}level\ (M\ @\ M')\ n\ L = get\text{-}rev\text{-}level\ M\ n\ L$
⟨*proof*⟩

lemma *get-level-skip-beginning*:
assumes $atm\text{-}of\ L' \neq atm\text{-}of\ (lit\text{-}of\ K)$
shows $get\text{-}level\ (K\ \#\ M)\ L' = get\text{-}level\ M\ L'$
⟨*proof*⟩

lemma *get-level-skip-beginning-not-decided-rev*:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ `\ lit\text{-}of\ `(set\ S)$
and $\forall s{\in}set\ S.\ \neg is\text{-}decided\ s$
shows $get\text{-}level\ (M\ @\ rev\ S)\ L = get\text{-}level\ M\ L$
⟨*proof*⟩

lemma *get-level-skip-beginning-not-decided*[*simp*]:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ `\ lit\text{-}of\ `(set\ S)$
and $\forall s{\in}set\ S.\ \neg is\text{-}decided\ s$
shows $get\text{-}level\ (M\ @\ S)\ L = get\text{-}level\ M\ L$
⟨*proof*⟩

lemma *get-rev-level-skip-beginning-not-decided*[*simp*]:
assumes $atm\text{-}of\ L \notin atm\text{-}of\ `\ lit\text{-}of\ `(set\ S)$
and $\forall s{\in}set\ S.\ \neg is\text{-}decided\ s$
shows $get\text{-}rev\text{-}level\ (rev\ S\ @\ rev\ M)\ 0\ L = get\text{-}level\ M\ L$
⟨*proof*⟩

lemma *get-level-skip-in-all-not-decided*:
fixes $M :: ('a,\ nat,\ 'b)\ ann\text{-}lit\ list$ **and** $L :: 'a\ literal$

assumes $\forall\, m \in set\ M.\ \neg\ is\text{-}decided\ m$
and $atm\text{-}of\ L \in atm\text{-}of\ `\ lit\text{-}of\ `\ (set\ M)$
shows $get\text{-}rev\text{-}level\ M\ n\ L = n$
$\langle proof \rangle$

lemma *get-level-skip-all-not-decided*[*simp*]:
fixes M
defines $M' \equiv rev\ M$
assumes $\forall\, m \in set\ M.\ \neg\ is\text{-}decided\ m$
shows $get\text{-}level\ M\ L = 0$
$\langle proof \rangle$

abbreviation $MMax\ M \equiv Max\ (set\text{-}mset\ M)$

the $\{\#0::'a\#\}$ is there to ensures that the set is not empty.

definition $get\text{-}maximum\text{-}level :: ('a,\ nat,\ 'b)\ ann\text{-}lit\ list \Rightarrow 'a\ literal\ multiset \Rightarrow nat$
where
$get\text{-}maximum\text{-}level\ M\ D = MMax\ (\{\#0\#\} + image\text{-}mset\ (get\text{-}level\ M)\ D)$

lemma *get-maximum-level-ge-get-level*:
$L \in\# D \Longrightarrow get\text{-}maximum\text{-}level\ M\ D \geq get\text{-}level\ M\ L$
$\langle proof \rangle$

lemma *get-maximum-level-empty*[*simp*]:
$get\text{-}maximum\text{-}level\ M\ \{\#\} = 0$
$\langle proof \rangle$

lemma *get-maximum-level-exists-lit-of-max-level*:
$D \neq \{\#\} \Longrightarrow \exists\, L \in\# D.\ get\text{-}level\ M\ L = get\text{-}maximum\text{-}level\ M\ D$
$\langle proof \rangle$

lemma *get-maximum-level-empty-list*[*simp*]:
$get\text{-}maximum\text{-}level\ []\ D = 0$
$\langle proof \rangle$

lemma *get-maximum-level-single*[*simp*]:
$get\text{-}maximum\text{-}level\ M\ \{\#L\#\} = get\text{-}level\ M\ L$
$\langle proof \rangle$

lemma *get-maximum-level-plus*:
$get\text{-}maximum\text{-}level\ M\ (D + D') = max\ (get\text{-}maximum\text{-}level\ M\ D)\ (get\text{-}maximum\text{-}level\ M\ D')$
$\langle proof \rangle$

lemma *get-maximum-level-exists-lit*:
assumes n: $n > 0$
and max: $get\text{-}maximum\text{-}level\ M\ D = n$
shows $\exists\, L \in\# D.\ get\text{-}level\ M\ L = n$
$\langle proof \rangle$

lemma *get-maximum-level-skip-first*[*simp*]:
assumes $atm\text{-}of\ L \notin atms\text{-}of\ D$
shows $get\text{-}maximum\text{-}level\ (Propagated\ L\ C\ \#\ M)\ D = get\text{-}maximum\text{-}level\ M\ D$
$\langle proof \rangle$

lemma *get-maximum-level-skip-beginning*:
assumes DH: *atms-of* $D \subseteq$ *atm-of* ‘*lits-of-l* H
shows *get-maximum-level* (c @ *Decided* Kh i # H) D = *get-maximum-level* H D
⟨*proof*⟩

lemma *get-maximum-level-D-single-propagated*:
get-maximum-level [*Propagated* *x21* *x22*] D = *0*
⟨*proof*⟩

lemma *get-maximum-level-skip-notin*:
assumes D: $\forall L \in \# D$. *atm-of* $L \in$ *atm-of* ‘*lits-of-l* M
shows *get-maximum-level* M D = *get-maximum-level* (*Propagated* *x21* *x22* # M) D
⟨*proof*⟩

lemma *get-maximum-level-skip-un-decided-not-present*:
assumes $\forall L \in \# D$. *atm-of* $L \in$ *atm-of* ‘ *lits-of-l* *aa* **and**
$\forall m \in$ *set* M. $\neg$ *is-decided* m
shows *get-maximum-level* *aa* D = *get-maximum-level* (M @ *aa*) D
⟨*proof*⟩

lemma *get-maximum-level-union-mset*:
get-maximum-level M (A #∪ B) = *get-maximum-level* M ($A + B$)
⟨*proof*⟩

fun *get-maximum-possible-level*:: ('*b*, *nat*, '*c*) *ann-lit list* ⇒ *nat* **where**
get-maximum-possible-level [] = *0* |
get-maximum-possible-level (*Decided* 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 (*Decided* - - # L) = *get-all-mark-of-propagated* L |
get-all-mark-of-propagated (*Propagated* - *mark* # L) = *mark* # *get-all-mark-of-propagated* L

lemma *get-all-mark-of-propagated-append*[*simp*]:
get-all-mark-of-propagated $(A\ @\ B)$ = *get-all-mark-of-propagated* A @ *get-all-mark-of-propagated* B
⟨*proof*⟩

18.5.2 Properties about the levels

fun *get-all-levels-of-ann* :: $('b, 'a, 'c)$ *ann-lit list* ⇒ *'a list* **where**
get-all-levels-of-ann [] = [] |
get-all-levels-of-ann (*Decided l level* # *Ls*) = *level* # *get-all-levels-of-ann Ls* |
get-all-levels-of-ann (*Propagated - -* # *Ls*) = *get-all-levels-of-ann Ls*

lemma *get-all-levels-of-ann-nil-iff-not-is-decided*:
get-all-levels-of-ann xs = [] ⟷ (∀ *x* ∈ *set xs*. ¬*is-decided x*)
⟨*proof*⟩

lemma *get-all-levels-of-ann-cons*:
get-all-levels-of-ann (*a* # *b*) =
(*if is-decided a then* [*level-of a*] *else* []) @ *get-all-levels-of-ann b*
⟨*proof*⟩

lemma *get-all-levels-of-ann-append*[*simp*]:
get-all-levels-of-ann (*a* @ *b*) = *get-all-levels-of-ann a* @ *get-all-levels-of-ann b*
⟨*proof*⟩

lemma *in-get-all-levels-of-ann-iff-decomp*:
i ∈ *set* (*get-all-levels-of-ann M*) ⟷ (∃ *c K c'*. *M* = *c* @ *Decided K i* # *c'*) (**is** *?A* ⟷ *?B*)
⟨*proof*⟩

lemma *get-rev-level-less-max-get-all-levels-of-ann*:
get-rev-level M n L ≤ *Max* (*set* (*n* # *get-all-levels-of-ann M*))
⟨*proof*⟩

lemma *get-rev-level-ge-min-get-all-levels-of-ann*:
assumes *atm-of L* ∈ *atm-of* ‘ *lits-of-l M*
shows *get-rev-level M n L* ≥ *Min* (*set* (*n* # *get-all-levels-of-ann M*))
⟨*proof*⟩

lemma *get-all-levels-of-ann-rev-eq-rev-get-all-levels-of-ann*[*simp*]:
get-all-levels-of-ann (*rev M*) = *rev* (*get-all-levels-of-ann M*)
⟨*proof*⟩

lemma *get-maximum-possible-level-max-get-all-levels-of-ann*:
get-maximum-possible-level M = *Max* (*insert 0* (*set* (*get-all-levels-of-ann M*)))
⟨*proof*⟩

lemma *get-rev-level-in-levels-of-decided*:
get-rev-level M n L ∈ {*0*, *n*} ∪ *set* (*get-all-levels-of-ann M*)
⟨*proof*⟩

lemma *get-rev-level-in-atms-in-levels-of-decided*:
atm-of L ∈ *atm-of* ‘ (*lits-of-l M*) ⟹
get-rev-level M n L ∈ {*n*} ∪ *set* (*get-all-levels-of-ann M*)
⟨*proof*⟩

lemma *get-all-levels-of-ann-no-decided*:
(∀ *l*∈*set Ls*. ¬ *is-decided l*) ⟷ *get-all-levels-of-ann Ls* = []

⟨*proof*⟩

lemma *get-level-in-levels-of-decided*:
get-level M L $\in$ {*0*} $\cup$ *set* (*get-all-levels-of-ann M*)
⟨*proof*⟩

The zero is here to avoid empty-list issues with *last*:

lemma *get-level-get-rev-level-get-all-levels-of-ann*:
assumes *atm-of L* $\notin$ *atm-of* ‘ (*lits-of-l M*)
shows
get-level (*K* @ *M*) *L* = *get-rev-level* (*rev K*) (*last* (*0* # *get-all-levels-of-ann* (*rev M*))) *L*
⟨*proof*⟩

lemma *get-rev-level-can-skip-correctly-ordered*:
assumes
no-dup M **and**
atm-of L $\notin$ *atm-of* ‘ (*lits-of-l M*) **and**
get-all-levels-of-ann M = *rev* [*Suc 0*..<*Suc* (*length* (*get-all-levels-of-ann M*))]
shows *get-rev-level* (*rev M* @ *K*) *0 L* = *get-rev-level K* (*length* (*get-all-levels-of-ann M*)) *L*
⟨*proof*⟩

lemma *get-level-skip-beginning-hd-get-all-levels-of-ann*:
assumes *atm-of L* $\notin$ *atm-of* ‘ *lits-of-l S* **and** *get-all-levels-of-ann S* $\neq$ []
shows *get-level* (*M*@ *S*) *L* = *get-rev-level* (*rev M*) (*hd* (*get-all-levels-of-ann S*)) *L*
⟨*proof*⟩

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**

$union\text{-}clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
$in\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow bool$ **and**
$insert\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
$remove\text{-}from\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

$mset\text{-}ccls :: 'ccls \Rightarrow 'v\ clause$ **and**
$union\text{-}ccls :: 'ccls \Rightarrow 'ccls \Rightarrow 'ccls$ **and**
$insert\text{-}ccls :: 'v\ literal \Rightarrow 'ccls \Rightarrow 'ccls$ **and**
$remove\text{-}clit :: 'v\ literal \Rightarrow 'ccls \Rightarrow 'ccls$
$+$
fixes
$ccls\text{-}of\text{-}cls :: 'cls \Rightarrow 'ccls$ **and**
$cls\text{-}of\text{-}ccls :: 'ccls \Rightarrow 'cls$ **and**

$trail :: 'st \Rightarrow ('v, nat, 'v\ clause)\ ann\text{-}lits$ **and**
$hd\text{-}raw\text{-}trail :: 'st \Rightarrow ('v, nat, 'cls)\ ann\text{-}lit$ **and**
$raw\text{-}init\text{-}clss :: 'st \Rightarrow 'clss$ **and**
$raw\text{-}learned\text{-}clss :: 'st \Rightarrow 'clss$ **and**
$backtrack\text{-}lvl :: 'st \Rightarrow nat$ **and**
$raw\text{-}conflicting :: 'st \Rightarrow 'ccls\ option$ **and**

$cons\text{-}trail :: ('v, nat, 'cls)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
$tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
$add\text{-}init\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$add\text{-}learned\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}backtrack\text{-}lvl :: nat \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}conflicting :: 'ccls\ option \Rightarrow 'st \Rightarrow 'st$ **and**

$init\text{-}state :: 'clss \Rightarrow 'st$ **and**
$restart\text{-}state :: 'st \Rightarrow 'st$
assumes
mset-ccls-ccls-of-cls[*simp*]:
$mset\text{-}ccls\ (ccls\text{-}of\text{-}cls\ C) = mset\text{-}cls\ C$ **and**
mset-cls-cls-of-ccls[*simp*]:
$mset\text{-}cls\ (cls\text{-}of\text{-}ccls\ D) = mset\text{-}ccls\ D$ **and**
ex-mset-cls: $\exists\, a.\ mset\text{-}cls\ a = E$
begin
fun $mmset\text{-}of\text{-}mlit :: ('a, 'b, 'cls)\ ann\text{-}lit \Rightarrow ('a, 'b, 'v\ clause)\ ann\text{-}lit$
where
$mmset\text{-}of\text{-}mlit\ (Propagated\ L\ C) = Propagated\ L\ (mset\text{-}cls\ C)$ |
$mmset\text{-}of\text{-}mlit\ (Decided\ L\ i) = Decided\ L\ i$

lemma *lit-of-mmset-of-mlit*[*simp*]:
$lit\text{-}of\ (mmset\text{-}of\text{-}mlit\ a) = lit\text{-}of\ a$
⟨*proof*⟩

lemma *lit-of-mmset-of-mlit-set-lit-of-l*[*simp*]:
$lit\text{-}of\ `\ mmset\text{-}of\text{-}mlit\ `\ set\ M' = lits\text{-}of\text{-}l\ M'$
⟨*proof*⟩

lemma *map-mmset-of-mlit-true-annots-true-cls*[*simp*]:
$map\ mmset\text{-}of\text{-}mlit\ M' \models as\ C \longleftrightarrow M' \models as\ C$
⟨*proof*⟩

abbreviation *init-clss* $\equiv \lambda S.$ *mset-clss* (*raw-init-clss* S)
abbreviation *learned-clss* $\equiv \lambda S.$ *mset-clss* (*raw-learned-clss* S)
abbreviation *conflicting* $\equiv \lambda S.$ *map-option mset-ccls* (*raw-conflicting* S)

notation *insert-cls* (**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 $S \equiv$ *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 decided literals;
2. the initial set of clauses (that is not changed during the whole calculus);
3. the learned clauses (clauses can be added or remove);
4. the maximum level of the trail;
5. the conflicting clause (if any has been found so far).

There are two different clause representation: one for the conflicting clause (*'ccls*, standing for conflicting clause) and one for the initial and learned clauses (*'cls*, standing for clause). The representation of the clauses annotating literals in the trail is slightly different: being able to convert it to *'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
$mset\text{-}cls :: 'cls \Rightarrow 'v\ clause$ **and**
$insert\text{-}cls :: 'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**
$remove\text{-}lit :: 'v\ literal \Rightarrow 'cls \Rightarrow 'cls$ **and**

$mset\text{-}clss :: 'clss \Rightarrow 'v\ clauses$ **and**
$union\text{-}clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
$in\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow bool$ **and**
$insert\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
$remove\text{-}from\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

$mset\text{-}ccls :: 'ccls \Rightarrow 'v\ clause$ **and**
$union\text{-}ccls :: 'ccls \Rightarrow 'ccls \Rightarrow 'ccls$ **and**
$insert\text{-}ccls :: 'v\ literal \Rightarrow 'ccls \Rightarrow 'ccls$ **and**
$remove\text{-}clit :: 'v\ literal \Rightarrow 'ccls \Rightarrow 'ccls$ **and**

$ccls\text{-}of\text{-}cls :: 'cls \Rightarrow 'ccls$ **and**
$cls\text{-}of\text{-}ccls :: 'ccls \Rightarrow 'cls$ **and**

$trail :: 'st \Rightarrow ('v,\ nat,\ 'v\ clause)\ ann\text{-}lits$ **and**
$hd\text{-}raw\text{-}trail :: 'st \Rightarrow ('v,\ nat,\ 'cls)\ ann\text{-}lit$ **and**
$raw\text{-}init\text{-}clss :: 'st \Rightarrow 'clss$ **and**
$raw\text{-}learned\text{-}clss :: 'st \Rightarrow 'clss$ **and**
$backtrack\text{-}lvl :: 'st \Rightarrow nat$ **and**
$raw\text{-}conflicting :: 'st \Rightarrow 'ccls\ option$ **and**

$cons\text{-}trail :: ('v,\ nat,\ 'cls)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
$tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
$add\text{-}init\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$add\text{-}learned\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}backtrack\text{-}lvl :: nat \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}conflicting :: 'ccls\ option \Rightarrow 'st \Rightarrow 'st$ **and**

$init\text{-}state :: 'clss \Rightarrow 'st$ **and**
$restart\text{-}state :: 'st \Rightarrow 'st$ +
assumes
$hd\text{-}raw\text{-}trail$: $trail\ S \neq [] \Longrightarrow mmset\text{-}of\text{-}mlit\ (hd\text{-}raw\text{-}trail\ S) = hd\ (trail\ S)$ **and**
$trail\text{-}cons\text{-}trail[simp]$:
$\bigwedge L\ st.\ undefined\text{-}lit\ (trail\ st)\ (lit\text{-}of\ L) \Longrightarrow$
$trail\ (cons\text{-}trail\ L\ st) = mmset\text{-}of\text{-}mlit\ L\ \#\ trail\ st$ **and**
$trail\text{-}tl\text{-}trail[simp]$: $\bigwedge st.\ trail\ (tl\text{-}trail\ st) = tl\ (trail\ st)$ **and**
$trail\text{-}add\text{-}init\text{-}cls[simp]$:
$\bigwedge st\ C.\ no\text{-}dup\ (trail\ st) \Longrightarrow trail\ (add\text{-}init\text{-}cls\ C\ st) = trail\ st$ **and**
$trail\text{-}add\text{-}learned\text{-}cls[simp]$:

$\bigwedge C\ st.\ \textit{no-dup}\ (\textit{trail}\ st) \Longrightarrow \textit{trail}\ (\textit{add-learned-cls}\ C\ st) = \textit{trail}\ st$ **and**
trail-remove-cls[*simp*]:
$\bigwedge C\ st.\ \textit{trail}\ (\textit{remove-cls}\ C\ st) = \textit{trail}\ st$ **and**
trail-update-backtrack-lvl[*simp*]: $\bigwedge st\ C.\ \textit{trail}\ (\textit{update-backtrack-lvl}\ C\ st) = \textit{trail}\ st$ **and**
trail-update-conflicting[*simp*]: $\bigwedge C\ st.\ \textit{trail}\ (\textit{update-conflicting}\ C\ st) = \textit{trail}\ st$ **and**

init-clss-cons-trail[*simp*]:
$\bigwedge M\ st.\ \textit{undefined-lit}\ (\textit{trail}\ st)\ (\textit{lit-of}\ M) \Longrightarrow$
$\textit{init-clss}\ (\textit{cons-trail}\ M\ st) = \textit{init-clss}\ st$
and
init-clss-tl-trail[*simp*]:
$\bigwedge st.\ \textit{init-clss}\ (\textit{tl-trail}\ st) = \textit{init-clss}\ st$ **and**
init-clss-add-init-cls[*simp*]:
$\bigwedge st\ C.\ \textit{no-dup}\ (\textit{trail}\ st) \Longrightarrow \textit{init-clss}\ (\textit{add-init-cls}\ C\ st) = \{\#\textit{mset-cls}\ C\#\} + \textit{init-clss}\ st$
and
init-clss-add-learned-cls[*simp*]:
$\bigwedge C\ st.\ \textit{no-dup}\ (\textit{trail}\ st) \Longrightarrow \textit{init-clss}\ (\textit{add-learned-cls}\ C\ st) = \textit{init-clss}\ st$ **and**
init-clss-remove-cls[*simp*]:
$\bigwedge C\ st.\ \textit{init-clss}\ (\textit{remove-cls}\ C\ st) = \textit{removeAll-mset}\ (\textit{mset-cls}\ C)\ (\textit{init-clss}\ st)$ **and**
init-clss-update-backtrack-lvl[*simp*]:
$\bigwedge st\ C.\ \textit{init-clss}\ (\textit{update-backtrack-lvl}\ C\ st) = \textit{init-clss}\ st$ **and**
init-clss-update-conflicting[*simp*]:
$\bigwedge C\ st.\ \textit{init-clss}\ (\textit{update-conflicting}\ C\ st) = \textit{init-clss}\ st$ **and**

learned-clss-cons-trail[*simp*]:
$\bigwedge M\ st.\ \textit{undefined-lit}\ (\textit{trail}\ st)\ (\textit{lit-of}\ M) \Longrightarrow$
$\textit{learned-clss}\ (\textit{cons-trail}\ M\ st) = \textit{learned-clss}\ st$ **and**
learned-clss-tl-trail[*simp*]:
$\bigwedge st.\ \textit{learned-clss}\ (\textit{tl-trail}\ st) = \textit{learned-clss}\ st$ **and**
learned-clss-add-init-cls[*simp*]:
$\bigwedge st\ C.\ \textit{no-dup}\ (\textit{trail}\ st) \Longrightarrow \textit{learned-clss}\ (\textit{add-init-cls}\ C\ st) = \textit{learned-clss}\ st$ **and**
learned-clss-add-learned-cls[*simp*]:
$\bigwedge C\ st.\ \textit{no-dup}\ (\textit{trail}\ st) \Longrightarrow$
$\textit{learned-clss}\ (\textit{add-learned-cls}\ C\ st) = \{\#\textit{mset-cls}\ C\#\} + \textit{learned-clss}\ st$ **and**
learned-clss-remove-cls[*simp*]:
$\bigwedge C\ st.\ \textit{learned-clss}\ (\textit{remove-cls}\ C\ st) = \textit{removeAll-mset}\ (\textit{mset-cls}\ C)\ (\textit{learned-clss}\ st)$ **and**
learned-clss-update-backtrack-lvl[*simp*]:
$\bigwedge st\ C.\ \textit{learned-clss}\ (\textit{update-backtrack-lvl}\ C\ st) = \textit{learned-clss}\ st$ **and**
learned-clss-update-conflicting[*simp*]:
$\bigwedge C\ st.\ \textit{learned-clss}\ (\textit{update-conflicting}\ C\ st) = \textit{learned-clss}\ st$ **and**

backtrack-lvl-cons-trail[*simp*]:
$\bigwedge M\ st.\ \textit{undefined-lit}\ (\textit{trail}\ st)\ (\textit{lit-of}\ M) \Longrightarrow$
$\textit{backtrack-lvl}\ (\textit{cons-trail}\ M\ st) = \textit{backtrack-lvl}\ st$ **and**
backtrack-lvl-tl-trail[*simp*]:
$\bigwedge st.\ \textit{backtrack-lvl}\ (\textit{tl-trail}\ st) = \textit{backtrack-lvl}\ st$ **and**
backtrack-lvl-add-init-cls[*simp*]:
$\bigwedge st\ C.\ \textit{no-dup}\ (\textit{trail}\ st) \Longrightarrow \textit{backtrack-lvl}\ (\textit{add-init-cls}\ C\ st) = \textit{backtrack-lvl}\ st$ **and**
backtrack-lvl-add-learned-cls[*simp*]:
$\bigwedge C\ st.\ \textit{no-dup}\ (\textit{trail}\ st) \Longrightarrow \textit{backtrack-lvl}\ (\textit{add-learned-cls}\ C\ st) = \textit{backtrack-lvl}\ st$ **and**
backtrack-lvl-remove-cls[*simp*]:
$\bigwedge C\ st.\ \textit{backtrack-lvl}\ (\textit{remove-cls}\ C\ st) = \textit{backtrack-lvl}\ st$ **and**
backtrack-lvl-update-backtrack-lvl[*simp*]:
$\bigwedge st\ k.\ \textit{backtrack-lvl}\ (\textit{update-backtrack-lvl}\ k\ st) = k$ **and**
backtrack-lvl-update-conflicting[*simp*]:

$\bigwedge C\ st.$ *backtrack-lvl* (*update-conflicting* C *st*) = *backtrack-lvl st* **and**

conflicting-cons-trail[*simp*]:
$\bigwedge M\ st.$ *undefined-lit* (*trail st*) (*lit-of* M) $\Longrightarrow$
conflicting (*cons-trail* M *st*) = *conflicting st* **and**
conflicting-tl-trail[*simp*]:
$\bigwedge st.$ *conflicting* (*tl-trail st*) = *conflicting st* **and**
conflicting-add-init-cls[*simp*]:
$\bigwedge st\ C.$ *no-dup* (*trail st*) $\Longrightarrow$ *conflicting* (*add-init-cls* C *st*) = *conflicting st* **and**
conflicting-add-learned-cls[*simp*]:
$\bigwedge C\ st.$ *no-dup* (*trail st*) $\Longrightarrow$ *conflicting* (*add-learned-cls* C *st*) = *conflicting st*
and
conflicting-remove-cls[*simp*]:
$\bigwedge C\ st.$ *conflicting* (*remove-cls* C *st*) = *conflicting st* **and**
conflicting-update-backtrack-lvl[*simp*]:
$\bigwedge st\ C.$ *conflicting* (*update-backtrack-lvl* C *st*) = *conflicting st* **and**
conflicting-update-conflicting[*simp*]:
$\bigwedge C\ st.$ *raw-conflicting* (*update-conflicting* C *st*) = C **and**

init-state-trail[*simp*]: $\bigwedge N.$ *trail* (*init-state* N) = [] **and**
init-state-clss[*simp*]: $\bigwedge N.$ (*init-clss* (*init-state* N)) = *mset-clss* N **and**
init-state-learned-clss[*simp*]: $\bigwedge N.$ *learned-clss* (*init-state* N) = {#} **and**
init-state-backtrack-lvl[*simp*]: $\bigwedge N.$ *backtrack-lvl* (*init-state* N) = *0* **and**
init-state-conflicting[*simp*]: $\bigwedge N.$ *conflicting* (*init-state* N) = *None* **and**

trail-restart-state[*simp*]: *trail* (*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) ⊆# *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) $\Longrightarrow$ *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) $\Longrightarrow$ *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) $\Longrightarrow$
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) $\Longrightarrow$ *clauses* (*add-learned-cls* C S) = {#*mset-cls* C#} + *clauses* S **and**
clauses-restart[*simp*]: *clauses* (*restart-state* S) ⊆# *clauses* S **and**
clauses-init-state[*simp*]: $\bigwedge N.$ *clauses* (*init-state* N) = *mset-clss* N
⟨*proof*⟩

abbreviation *state* :: $'st \Rightarrow ('v, nat, 'v\ clause)\ ann\text{-}lit\ list \times 'v\ clauses \times 'v\ clauses \times nat \times 'v\ clause\ option$ **where**
state S $\equiv$ (*trail S*, *init-clss S*, *learned-clss S*, *backtrack-lvl S*, *conflicting S*)

abbreviation *incr-lvl* :: $'st \Rightarrow 'st$ **where**
incr-lvl S $\equiv$ *update-backtrack-lvl* (*backtrack-lvl S* + *1*) *S*

definition *state-eq* :: $'st \Rightarrow 'st \Rightarrow bool$ (**infix** $\sim$ *50*) **where**
$S \sim T \longleftrightarrow state\ S = state\ T$

lemma *state-eq-ref*[*simp*, *intro*]:
$S \sim S$
⟨*proof*⟩

lemma *state-eq-sym*:
$S \sim T \longleftrightarrow T \sim S$
⟨*proof*⟩

lemma *state-eq-trans*:
$S \sim T \Longrightarrow T \sim U \Longrightarrow S \sim U$
⟨*proof*⟩

lemma
shows
state-eq-trail: $S \sim T \Longrightarrow trail\ S = trail\ T$ **and**
state-eq-init-clss: $S \sim T \Longrightarrow init\text{-}clss\ S = init\text{-}clss\ T$ **and**
state-eq-learned-clss: $S \sim T \Longrightarrow learned\text{-}clss\ S = learned\text{-}clss\ T$ **and**
state-eq-backtrack-lvl: $S \sim T \Longrightarrow backtrack\text{-}lvl\ S = backtrack\text{-}lvl\ T$ **and**
state-eq-conflicting: $S \sim T \Longrightarrow conflicting\ S = conflicting\ T$ **and**
state-eq-clauses: $S \sim T \Longrightarrow clauses\ S = clauses\ T$ **and**
state-eq-undefined-lit: $S \sim T \Longrightarrow undefined\text{-}lit\ (trail\ S)\ L = undefined\text{-}lit\ (trail\ T)\ L$
⟨*proof*⟩

lemma *state-eq-raw-conflicting-None*:
$S \sim T \Longrightarrow conflicting\ T = None \Longrightarrow raw\text{-}conflicting\ S = None$
⟨*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*]:
$x \in atms\text{-}of\text{-}mm\ (learned\text{-}clss\ (restart\text{-}state\ S)) \Longrightarrow x \in atms\text{-}of\text{-}mm\ (learned\text{-}clss\ S)$
⟨*proof*⟩

function *reduce-trail-to* :: $'a\ list \Rightarrow 'st \Rightarrow 'st$ **where**
reduce-trail-to F S =
(*if length* (*trail S*) = *length F* $\vee$ *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*]: $trail\ S = [] \Longrightarrow reduce\text{-}trail\text{-}to\ F\ S = S$ **and**
    reduce-trail-to-eq-length[*simp*]: $length\ (trail\ S) = length\ F \Longrightarrow reduce\text{-}trail\text{-}to\ F\ S = S$
  ⟨*proof*⟩

lemma *reduce-trail-to-length-ne*:
  $length\ (trail\ S) \neq length\ F \Longrightarrow trail\ S \neq [] \Longrightarrow$
    $reduce\text{-}trail\text{-}to\ F\ S = reduce\text{-}trail\text{-}to\ F\ (tl\text{-}trail\ S)$
  ⟨*proof*⟩

lemma *trail-reduce-trail-to-length-le*:
  assumes $length\ F > length\ (trail\ S)$
  shows $trail\ (reduce\text{-}trail\text{-}to\ F\ S) = []$
  ⟨*proof*⟩

lemma *trail-reduce-trail-to-nil*[*simp*]:
  $trail\ (reduce\text{-}trail\text{-}to\ []\ S) = []$
  ⟨*proof*⟩

lemma *clauses-reduce-trail-to-nil*:
  $clauses\ (reduce\text{-}trail\text{-}to\ []\ S) = clauses\ S$
⟨*proof*⟩

lemma *reduce-trail-to-skip-beginning*:
  assumes $trail\ S = F'\ @\ F$
  shows $trail\ (reduce\text{-}trail\text{-}to\ F\ S) = F$
  ⟨*proof*⟩

lemma *clauses-reduce-trail-to*[*simp*]:
  $clauses\ (reduce\text{-}trail\text{-}to\ F\ S) = clauses\ S$
  ⟨*proof*⟩

lemma *conflicting-update-trail*[*simp*]:
  $conflicting\ (reduce\text{-}trail\text{-}to\ F\ S) = conflicting\ S$
  ⟨*proof*⟩

lemma *backtrack-lvl-update-trail*[*simp*]:
  $backtrack\text{-}lvl\ (reduce\text{-}trail\text{-}to\ F\ S) = backtrack\text{-}lvl\ S$
  ⟨*proof*⟩

lemma *init-clss-update-trail*[*simp*]:
  $init\text{-}clss\ (reduce\text{-}trail\text{-}to\ F\ S) = init\text{-}clss\ S$
  ⟨*proof*⟩

lemma *learned-clss-update-trail*[*simp*]:
  $learned\text{-}clss\ (reduce\text{-}trail\text{-}to\ F\ S) = learned\text{-}clss\ S$
  ⟨*proof*⟩

lemma *raw-conflicting-reduce-trail-to*[*simp*]:
  $raw\text{-}conflicting\ (reduce\text{-}trail\text{-}to\ F\ S) = None \longleftrightarrow raw\text{-}conflicting\ S = None$

⟨proof⟩

lemma *trail-eq-reduce-trail-to-eq*:
trail S = *trail T* ⟹ *trail* (*reduce-trail-to F S*) = *trail* (*reduce-trail-to F T*)
⟨proof⟩

lemma *reduce-trail-to-state-eq*$_{NOT}$*-compatible*:
assumes *ST*: *S* ∼ *T*
shows *reduce-trail-to F S* ∼ *reduce-trail-to F T*
⟨proof⟩

lemma *reduce-trail-to-trail-tl-trail-decomp*[*simp*]:
trail S = *F'* @ *Decided K d* # *F* ⟹ (*trail* (*reduce-trail-to F S*)) = *F*
⟨proof⟩

lemma *reduce-trail-to-add-learned-cls*[*simp*]:
no-dup (*trail S*) ⟹
trail (*reduce-trail-to F* (*add-learned-cls C S*)) = *trail* (*reduce-trail-to F S*)
⟨proof⟩

lemma *reduce-trail-to-add-init-cls*[*simp*]:
no-dup (*trail S*) ⟹
trail (*reduce-trail-to F* (*add-init-cls C S*)) = *trail* (*reduce-trail-to F S*)
⟨proof⟩

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-ann-decomposition-decided-or-empty*:
assumes (*a*, *b*) ∈ *set* (*get-all-ann-decomposition M*)
shows *a* = [] ∨ (*is-decided* (*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-ann-decomposition-trail-update-trail*[*simp*]:
assumes *H*: (*L* # *M1*, *M2*) ∈ *set* (*get-all-ann-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-backtracl-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 $conflict\text{-}driven\text{-}clause\text{-}learning_W$ =
$state_W$
— 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-raw-trail raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting
— changing state:
cons-trail tl-trail add-init-cls add-learned-cls remove-cls update-backtrack-lvl
update-conflicting

— get state:
init-state
restart-state
for
mset-cls :: *'cls* ⇒ *'v clause* **and**
insert-cls :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**
remove-lit :: *'v literal* ⇒ *'cls* ⇒ *'cls* **and**

$mset\text{-}clss :: 'clss \Rightarrow 'v\ clauses$ **and**
$union\text{-}clss :: 'clss \Rightarrow 'clss \Rightarrow 'clss$ **and**
$in\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow bool$ **and**
$insert\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**
$remove\text{-}from\text{-}clss :: 'cls \Rightarrow 'clss \Rightarrow 'clss$ **and**

$mset\text{-}ccls :: 'ccls \Rightarrow 'v\ clause$ **and**
$union\text{-}ccls :: 'ccls \Rightarrow 'ccls \Rightarrow 'ccls$ **and**
$insert\text{-}ccls :: 'v\ literal \Rightarrow 'ccls \Rightarrow 'ccls$ **and**
$remove\text{-}clit :: 'v\ literal \Rightarrow 'ccls \Rightarrow 'ccls$ **and**

$ccls\text{-}of\text{-}cls :: 'cls \Rightarrow 'ccls$ **and**
$cls\text{-}of\text{-}ccls :: 'ccls \Rightarrow 'cls$ **and**

$trail :: 'st \Rightarrow ('v, nat, 'v\ clause)\ ann\text{-}lits$ **and**
$hd\text{-}raw\text{-}trail :: 'st \Rightarrow ('v, nat, 'cls)\ ann\text{-}lit$ **and**
$raw\text{-}init\text{-}clss :: 'st \Rightarrow 'clss$ **and**
$raw\text{-}learned\text{-}clss :: 'st \Rightarrow 'clss$ **and**
$backtrack\text{-}lvl :: 'st \Rightarrow nat$ **and**
$raw\text{-}conflicting :: 'st \Rightarrow 'ccls\ option$ **and**

$cons\text{-}trail :: ('v, nat, 'cls)\ ann\text{-}lit \Rightarrow 'st \Rightarrow 'st$ **and**
$tl\text{-}trail :: 'st \Rightarrow 'st$ **and**
$add\text{-}init\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$add\text{-}learned\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$remove\text{-}cls :: 'cls \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}backtrack\text{-}lvl :: nat \Rightarrow 'st \Rightarrow 'st$ **and**
$update\text{-}conflicting :: 'ccls\ option \Rightarrow 'st \Rightarrow 'st$ **and**

$init\text{-}state :: 'clss \Rightarrow 'st$ **and**
$restart\text{-}state :: 'st \Rightarrow 'st$
begin

inductive $propagate :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**
$propagate\text{-}rule$: $conflicting\ S = None \Longrightarrow$
$E\ !\in!\ raw\text{-}clauses\ S \Longrightarrow$
$L \in\#\ mset\text{-}cls\ E \Longrightarrow$
$trail\ S \models as\ CNot\ (mset\text{-}cls\ (remove\text{-}lit\ L\ E)) \Longrightarrow$
$undefined\text{-}lit\ (trail\ S)\ L \Longrightarrow$
$T \sim cons\text{-}trail\ (Propagated\ L\ E)\ S \Longrightarrow$
$propagate\ S\ T$

inductive-cases $propagateE$: $propagate\ S\ T$

inductive $conflict :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**
$conflict\text{-}rule$:
$conflicting\ S = None \Longrightarrow$
$D\ !\in!\ raw\text{-}clauses\ S \Longrightarrow$
$trail\ S \models as\ CNot\ (mset\text{-}cls\ D) \Longrightarrow$
$T \sim update\text{-}conflicting\ (Some\ (ccls\text{-}of\text{-}cls\ D))\ S \Longrightarrow$
$conflict\ S\ T$

inductive-cases $conflictE$: $conflict\ S\ T$

inductive $backtrack :: 'st \Rightarrow 'st \Rightarrow bool$ **for** $S :: 'st$ **where**

backtrack-rule:
  raw-conflicting S = *Some* $D \Longrightarrow$
  $L \in\#$ *mset-ccls* $D \Longrightarrow$
  (*Decided* K $(i+1)$ # $M1$, $M2$) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$
  get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
  get-level (*trail* S) L = *get-maximum-level* (*trail* S) (*mset-ccls* D) $\Longrightarrow$
  get-maximum-level (*trail* S) (*mset-ccls* (*remove-clit* L D)) $\equiv i \Longrightarrow$
  $T \sim$ *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)))) $\Longrightarrow$
  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* $\Longrightarrow$
  undefined-lit (*trail* S) $L \Longrightarrow$
  atm-of $L \in$ *atms-of-mm* (*init-clss* S) $\Longrightarrow$
  $T \sim$ *cons-trail* (*Decided* L (*backtrack-lvl* S $+1$)) (*incr-lvl* S) $\Longrightarrow$
  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' # $M \Longrightarrow$
   raw-conflicting S = *Some* $E \Longrightarrow$
   $-L \notin\#$ *mset-ccls* $E \Longrightarrow$
   mset-ccls $E \neq \{\#\} \Longrightarrow$
   $T \sim$ *tl-trail* $S \Longrightarrow$
   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* $S \neq [] \Longrightarrow$
  hd-raw-trail S = *Propagated* L $E \Longrightarrow$
  $L \in\#$ *mset-cls* $E \Longrightarrow$
  raw-conflicting S = *Some* $D' \Longrightarrow$
  $-L \in\#$ *mset-ccls* $D' \Longrightarrow$
  get-maximum-level (*trail* S) (*mset-ccls* (*remove-clit* $(-L)$ D')) = *backtrack-lvl* $S \Longrightarrow$
  $T \sim$ *update-conflicting* (*Some* (*union-ccls* (*remove-clit* $(-L)$ D') (*ccls-of-cls* (*remove-lit* L E))))
   (*tl-trail* S) $\Longrightarrow$
  resolve S T

inductive-cases *resolveE*: *resolve* S T

inductive *restart* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **for** S :: *'st* **where**

restart: *state* $S = (M, N, U, k, None) \Longrightarrow \neg M \models asm\ clauses\ S$
$\Longrightarrow T \sim restart\text{-}state\ S$
$\Longrightarrow restart\ S\ T$

inductive-cases *restartE*: *restart S T*

We add the condition $C \notin\#$ *init-clss* S, to maintain consistency even without the strategy.

inductive *forget* :: *'st* ⇒ *'st* ⇒ *bool* **where**
forget-rule:
$conflicting\ S = None \Longrightarrow$
$C\ !{\in}!\ raw\text{-}learned\text{-}clss\ S \Longrightarrow$
$\neg(trail\ S) \models asm\ clauses\ S \Longrightarrow$
$mset\text{-}cls\ C \notin set\ (get\text{-}all\text{-}mark\text{-}of\text{-}propagated\ (trail\ S)) \Longrightarrow$
$mset\text{-}cls\ C \notin\#\ init\text{-}clss\ S \Longrightarrow$
$T \sim remove\text{-}cls\ C\ S \Longrightarrow$
forget S T

inductive-cases *forgetE*: *forget S T*

inductive $cdcl_W$*-rf* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
restart: $restart\ S\ T \Longrightarrow cdcl_W\text{-}rf\ S\ T$ |
forget: $forget\ S\ T \Longrightarrow cdcl_W\text{-}rf\ S\ T$

inductive $cdcl_W$*-bj* :: *'st* ⇒ *'st* ⇒ *bool* **where**
skip: $skip\ S\ S' \Longrightarrow cdcl_W\text{-}bj\ S\ S'$ |
resolve: $resolve\ S\ S' \Longrightarrow cdcl_W\text{-}bj\ S\ S'$ |
backtrack: $backtrack\ S\ S' \Longrightarrow cdcl_W\text{-}bj\ S\ S'$

inductive-cases $cdcl_W$*-bjE*: $cdcl_W\text{-}bj\ S\ T$

inductive $cdcl_W$*-o* :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
decide: $decide\ S\ S' \Longrightarrow cdcl_W\text{-}o\ S\ S'$ |
bj: $cdcl_W\text{-}bj\ S\ S' \Longrightarrow cdcl_W\text{-}o\ S\ S'$

inductive $cdcl_W$ :: *'st* ⇒ *'st* ⇒ *bool* **for** *S* :: *'st* **where**
propagate: $propagate\ S\ S' \Longrightarrow cdcl_W\ S\ S'$ |
conflict: $conflict\ S\ S' \Longrightarrow cdcl_W\ S\ S'$ |
other: $cdcl_W\text{-}o\ S\ S' \Longrightarrow cdcl_W\ S\ S'$|
rf: $cdcl_W\text{-}rf\ S\ S' \Longrightarrow cdcl_W\ S\ S'$

lemma *rtranclp-propagate-is-rtranclp-cdcl*$_W$:
$propagate^{**}\ S\ S' \Longrightarrow cdcl_W^{**}\ S\ S'$
⟨*proof*⟩

lemma $cdcl_W$*-all-rules-induct*[*consumes 1*, *case-names propagate conflict forget restart decide skip resolve backtrack*]:
fixes *S* :: *'st*
assumes
$cdcl_W$: $cdcl_W\ S\ S'$ **and**
propagate: $\bigwedge T.\ propagate\ S\ T \Longrightarrow P\ S\ T$ **and**
conflict: $\bigwedge T.\ conflict\ S\ T \Longrightarrow P\ S\ T$ **and**
forget: $\bigwedge T.\ forget\ S\ T \Longrightarrow P\ S\ T$ **and**
restart: $\bigwedge T.\ restart\ S\ T \Longrightarrow P\ S\ T$ **and**
decide: $\bigwedge T.\ decide\ S\ T \Longrightarrow P\ S\ T$ **and**
skip: $\bigwedge T.\ skip\ S\ T \Longrightarrow P\ S\ T$ **and**

resolve: $\bigwedge T.$ *resolve* S $T \Longrightarrow P$ S T **and**
backtrack: $\bigwedge T.$ *backtrack* S $T \Longrightarrow P$ S T
shows P S S'
⟨*proof*⟩

lemma $cdcl_W$*-all-induct*[*consumes 1*, *case-names propagate conflict forget restart decide skip resolve backtrack*]:
fixes S :: *'st*
assumes
$cdcl_W$: $cdcl_W$ S S' **and**
propagateH: $\bigwedge C$ L T. *conflicting* S = *None* $\Longrightarrow$
C !∈! *raw-clauses* $S \Longrightarrow$
$L \in\#$ *mset-cls* $C \Longrightarrow$
trail $S \models as$ *CNot* (*remove1-mset* L (*mset-cls* C)) $\Longrightarrow$
undefined-lit (*trail* S) $L \Longrightarrow$
$T \sim$ *cons-trail* (*Propagated* L C) $S \Longrightarrow$
P S T **and**
conflictH: $\bigwedge D$ T. *conflicting* S = *None* $\Longrightarrow$
D !∈! *raw-clauses* $S \Longrightarrow$
trail $S \models as$ *CNot* (*mset-cls* D) $\Longrightarrow$
$T \sim$ *update-conflicting* (*Some* (*ccls-of-cls* D)) $S \Longrightarrow$
P S T **and**
forgetH: $\bigwedge C$ U T. *conflicting* S = *None* $\Longrightarrow$
C !∈! *raw-learned-clss* $S \Longrightarrow$
$\neg$(*trail* S) $\models asm$ *clauses* $S \Longrightarrow$
mset-cls $C \notin$ *set* (*get-all-mark-of-propagated* (*trail* S)) $\Longrightarrow$
mset-cls $C \notin\#$ *init-clss* $S \Longrightarrow$
$T \sim$ *remove-cls* C $S \Longrightarrow$
P S T **and**
restartH: $\bigwedge T$. $\neg$*trail* $S \models asm$ *clauses* $S \Longrightarrow$
conflicting S = *None* $\Longrightarrow$
$T \sim$ *restart-state* $S \Longrightarrow$
P S T **and**
decideH: $\bigwedge L$ T. *conflicting* S = *None* $\Longrightarrow$
undefined-lit (*trail* S) $L \Longrightarrow$
atm-of $L \in$ *atms-of-mm* (*init-clss* S) $\Longrightarrow$
$T \sim$ *cons-trail* (*Decided* L (*backtrack-lvl* S +*1*)) (*incr-lvl* S) $\Longrightarrow$
P S T **and**
skipH: $\bigwedge L$ C' M E T.
trail S = *Propagated* L C' # $M \Longrightarrow$
raw-conflicting S = *Some* $E \Longrightarrow$
$-L \notin\#$ *mset-ccls* $E \Longrightarrow$ *mset-ccls* $E \neq \{\#\} \Longrightarrow$
$T \sim$ *tl-trail* $S \Longrightarrow$
P S T **and**
resolveH: $\bigwedge L$ E M D T.
trail S = *Propagated* L (*mset-cls* E) # $M \Longrightarrow$
$L \in\#$ *mset-cls* $E \Longrightarrow$
hd-raw-trail S = *Propagated* L $E \Longrightarrow$
raw-conflicting S = *Some* $D \Longrightarrow$
$-L \in\#$ *mset-ccls* $D \Longrightarrow$
get-maximum-level (*trail* S) (*mset-ccls* (*remove-clit* $(-L)$ D)) = *backtrack-lvl* $S \Longrightarrow$
$T \sim$ *update-conflicting*
(*Some* (*union-ccls* (*remove-clit* $(-L)$ D) (*ccls-of-cls* (*remove-lit* L E)))) (*tl-trail* S) $\Longrightarrow$
P S T **and**
backtrackH: $\bigwedge L$ D K i *M1* *M2* T.

$raw\text{-}conflicting\ S = Some\ D \Longrightarrow$
$L \in\#\ mset\text{-}ccls\ D \Longrightarrow$
$(Decided\ K\ (i{+}1)\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ S)) \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = backtrack\text{-}lvl\ S \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ D) \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (remove1\text{-}mset\ L\ (mset\text{-}ccls\ D)) \equiv i \Longrightarrow$
$T \sim cons\text{-}trail\ (Propagated\ L\ (cls\text{-}of\text{-}ccls\ D))$
$(reduce\text{-}trail\text{-}to\ M1$
$(add\text{-}learned\text{-}cls\ (cls\text{-}of\text{-}ccls\ D)$
$(update\text{-}backtrack\text{-}lvl\ i$
$(update\text{-}conflicting\ None\ S)))) \Longrightarrow$
$P\ S\ T$
shows $P\ S\ S'$
⟨*proof*⟩

lemma $cdcl_W\text{-}o\text{-}induct[consumes\ 1,\ case\text{-}names\ decide\ skip\ resolve\ backtrack]$:
fixes $S\ ::\ 'st$
assumes $cdcl_W$: $cdcl_W\text{-}o\ S\ T$ **and**
$decideH$: $\bigwedge L\ T.\ conflicting\ S = None \Longrightarrow\ undefined\text{-}lit\ (trail\ S)\ L$
$\Longrightarrow atm\text{-}of\ L \in atms\text{-}of\text{-}mm\ (init\text{-}clss\ S)$
$\Longrightarrow T \sim cons\text{-}trail\ (Decided\ L\ (backtrack\text{-}lvl\ S\ {+}1))\ (incr\text{-}lvl\ S)$
$\Longrightarrow P\ S\ T$ **and**
$skipH$: $\bigwedge L\ C'\ M\ E\ T.$
$trail\ S = Propagated\ L\ C'\ \#\ M \Longrightarrow$
$raw\text{-}conflicting\ S = Some\ E \Longrightarrow$
$-L \notin\#\ mset\text{-}ccls\ E \Longrightarrow\ mset\text{-}ccls\ E \neq \{\#\} \Longrightarrow$
$T \sim tl\text{-}trail\ S \Longrightarrow$
$P\ S\ T$ **and**
$resolveH$: $\bigwedge L\ E\ M\ D\ T.$
$trail\ S = Propagated\ L\ (mset\text{-}cls\ E)\ \#\ M \Longrightarrow$
$L \in\#\ mset\text{-}cls\ E \Longrightarrow$
$hd\text{-}raw\text{-}trail\ S = Propagated\ L\ E \Longrightarrow$
$raw\text{-}conflicting\ S = Some\ D \Longrightarrow$
$-L \in\#\ mset\text{-}ccls\ D \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ (remove\text{-}clit\ (-L)\ D)) = backtrack\text{-}lvl\ S \Longrightarrow$
$T \sim update\text{-}conflicting$
$(Some\ (union\text{-}ccls\ (remove\text{-}clit\ (-L)\ D)\ (ccls\text{-}of\text{-}cls\ (remove\text{-}lit\ L\ E))))\ (tl\text{-}trail\ S) \Longrightarrow$
$P\ S\ T$ **and**
$backtrackH$: $\bigwedge L\ D\ K\ i\ M1\ M2\ T.$
$raw\text{-}conflicting\ S = Some\ D \Longrightarrow$
$L \in\#\ mset\text{-}ccls\ D \Longrightarrow$
$(Decided\ K\ (i{+}1)\ \#\ M1,\ M2) \in set\ (get\text{-}all\text{-}ann\text{-}decomposition\ (trail\ S)) \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = backtrack\text{-}lvl\ S \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ D) \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (remove1\text{-}mset\ L\ (mset\text{-}ccls\ D)) \equiv i \Longrightarrow$
$T \sim cons\text{-}trail\ (Propagated\ L\ (cls\text{-}of\text{-}ccls\ D))$
$(reduce\text{-}trail\text{-}to\ M1$
$(add\text{-}learned\text{-}cls\ (cls\text{-}of\text{-}ccls\ D)$
$(update\text{-}backtrack\text{-}lvl\ i$
$(update\text{-}conflicting\ None\ S)))) \Longrightarrow$
$P\ S\ T$
shows $P\ S\ T$
⟨*proof*⟩

thm $cdcl_W\text{-}o.induct$

lemma *cdcl*$_W$*-o-all-rules-induct*[*consumes 1*, *case-names decide backtrack skip resolve*]:
fixes S T :: *'st*
assumes
cdcl$_W$*-o* S T **and**
$\bigwedge T$. *decide* S T $\Longrightarrow$ P S T **and**
$\bigwedge T$. *backtrack* S T $\Longrightarrow$ P S T **and**
$\bigwedge T$. *skip* S T $\Longrightarrow$ P S T **and**
$\bigwedge T$. *resolve* S T $\Longrightarrow$ P S T
shows P S T
⟨*proof*⟩

lemma *cdcl*$_W$*-o-rule-cases*[*consumes 1*, *case-names decide backtrack skip resolve*]:
fixes S T :: *'st*
assumes
cdcl$_W$*-o* S T **and**
decide S T $\Longrightarrow$ P **and**
backtrack S T $\Longrightarrow$ P **and**
skip S T $\Longrightarrow$ P **and**
resolve S T $\Longrightarrow$ P
shows P
⟨*proof*⟩

19.3 Invariants

19.3.1 Properties of the trail

We here establish that:

- the marks are exactly [*1..<Suc k*] where k is the level;
- the consistency of the trail;
- the fact that there is no duplicate in the trail.

lemma *backtrack-lit-skiped*:
assumes
L: *get-level* (*trail S*) *L* = *backtrack-lvl S* **and**
M1: (*Decided K* (*i* + *1*) # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
no-dup: *no-dup* (*trail S*) **and**
bt-l: *backtrack-lvl S* = *length* (*get-all-levels-of-ann* (*trail S*)) **and**
order: *get-all-levels-of-ann* (*trail S*)
= *rev* [*1..<1+length* (*get-all-levels-of-ann* (*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-ann* (*trail S*)) **and**
get-all-levels-of-ann (*trail S*) = *rev* [*1..<1+length* (*get-all-levels-of-ann* (*trail S*))]
shows *no-dup* (*trail S'*)
⟨*proof*⟩

Item 1 page 81 of Weidenbach’s book

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-ann* (*trail S*)) **and**
    get-all-levels-of-ann (*trail S*) = *rev* [*1..<1+length* (*get-all-levels-of-ann* (*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-ann* (*trail S*)) **and**
    get-all-levels-of-ann (*trail S*) =
      rev [*1..<1+length* (*get-all-levels-of-ann* (*trail S*))] **and**
    n-d[*simp*]: *no-dup* (*trail S*)
  shows *backtrack-lvl S'* = *length* (*get-all-levels-of-ann* (*trail S'*))
  ⟨*proof*⟩

lemma $cdcl_W$*-rf-bt*:
  assumes
    $cdcl_W$*-rf S S'* **and**
    backtrack-lvl S = *length* (*get-all-levels-of-ann* (*trail S*)) **and**
    get-all-levels-of-ann (*trail S*) = *rev* [*1..<1+length* (*get-all-levels-of-ann* (*trail S*))]
  shows *backtrack-lvl S'* = *length* (*get-all-levels-of-ann* (*trail S'*))
  ⟨*proof*⟩

Item 7 page 81 of Weidenbach's book

lemma $cdcl_W$*-bt*:
  assumes
    $cdcl_W$ *S S'* **and**
    backtrack-lvl S = *length* (*get-all-levels-of-ann* (*trail S*)) **and**
    get-all-levels-of-ann (*trail S*)
    = *rev* ([*1..<1+length* (*get-all-levels-of-ann* (*trail S*))]) **and**
    no-dup (*trail S*)
  shows *backtrack-lvl S'* = *length* (*get-all-levels-of-ann* (*trail S'*))
  ⟨*proof*⟩

Stated in proof of Item 7 page 81 of Weidenbach's book

lemma $cdcl_W$*-bt-level'*:
  assumes
    $cdcl_W$ *S S'* **and**
    backtrack-lvl S = *length* (*get-all-levels-of-ann* (*trail S*)) **and**
    get-all-levels-of-ann (*trail S*)
      = *rev* ([*1..<1+length* (*get-all-levels-of-ann* (*trail S*))]) **and**
    n-d: *no-dup* (*trail S*)
  shows *get-all-levels-of-ann* (*trail S'*)
    = *rev* [*1..<1+length* (*get-all-levels-of-ann* (*trail S'*))]
  ⟨*proof*⟩

We write *1* + *length* (*get-all-levels-of-ann* (*trail S*)) instead of *backtrack-lvl S* to avoid non termination of rewriting.

definition $cdcl_W$*-M-level-inv* :: *'st* ⇒ *bool* **where**
$cdcl_W$*-M-level-inv S* ⟷
  consistent-interp (*lits-of-l* (*trail S*))

$\wedge$ *no-dup* (*trail S*)
$\wedge$ *backtrack-lvl S* = *length* (*get-all-levels-of-ann* (*trail S*))
$\wedge$ *get-all-levels-of-ann* (*trail S*)
= *rev* [*1..<1+length* (*get-all-levels-of-ann* (*trail S*))]

lemma $cdcl_W$*-M-level-inv-decomp*:
assumes $cdcl_W$*-M-level-inv S*
shows
consistent-interp (*lits-of-l* (*trail S*)) **and**
no-dup (*trail S*)
⟨*proof*⟩

lemma $cdcl_W$*-consistent-inv*:
fixes *S S'* :: *'st*
assumes
$cdcl_W$ *S S'* **and**
$cdcl_W$*-M-level-inv S*
shows $cdcl_W$*-M-level-inv S'*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-consistent-inv*:
assumes
$cdcl_W^{**}$ *S S'* **and**
$cdcl_W$*-M-level-inv S*
shows $cdcl_W$*-M-level-inv S'*
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-consistent-inv*:
assumes
$cdcl_W^{++}$ *S S'* **and**
$cdcl_W$*-M-level-inv S*
shows $cdcl_W$*-M-level-inv S'*
⟨*proof*⟩

lemma $cdcl_W$*-M-level-inv-S0-*$cdcl_W$[*simp*]:
$cdcl_W$*-M-level-inv* (*init-state N*)
⟨*proof*⟩

lemma $cdcl_W$*-M-level-inv-get-level-le-backtrack-lvl*:
assumes *inv*: $cdcl_W$*-M-level-inv S*
shows *get-level* (*trail S*) *L* $\leq$ *backtrack-lvl S*
⟨*proof*⟩

lemma *backtrack-ex-decomp*:
assumes
M-l: $cdcl_W$*-M-level-inv S* **and**
i-S: *i* < *backtrack-lvl S*
shows $\exists$ *K M1 M2*. (*Decided K* (*i* + *1*) # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail S*))
⟨*proof*⟩

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 *backtrack-induction-lev*[*consumes 1*, *case-names M-devel-inv backtrack*]:
  assumes
    bt: *backtrack S T* **and**
    inv: $cdcl_W$*-M-level-inv S* **and**
    backtrackH: $\bigwedge$*K i M1 M2 L D T*.
      raw-conflicting S = *Some D* $\Longrightarrow$
      L $\in\#$ *mset-ccls D* $\Longrightarrow$
      (*Decided K* (*Suc i*) # *M1*, *M2*) $\in$ *set* (*get-all-ann-decomposition* (*trail S*)) $\Longrightarrow$
      get-level (*trail S*) *L* = *backtrack-lvl S* $\Longrightarrow$
      get-level (*trail S*) *L* = *get-maximum-level* (*trail S*) (*mset-ccls D*) $\Longrightarrow$
      get-maximum-level (*trail S*) (*remove1-mset L* (*mset-ccls D*)) $\equiv$ *i* $\Longrightarrow$
      undefined-lit M1 L $\Longrightarrow$
      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*)))) $\Longrightarrow$
      P S T
  shows *P S T*
$\langle$*proof*$\rangle$

lemmas *backtrack-induction-lev2* = *backtrack-induction-lev*[*consumes 2*, *case-names backtrack*]

lemma $cdcl_W$*-all-induct-lev-full*:
  fixes *S* :: *'st*
  assumes
    $cdcl_W$: $cdcl_W$ *S S'* **and**
    inv[*simp*]: $cdcl_W$*-M-level-inv S* **and**
    propagateH: $\bigwedge$*C L T*. *conflicting S* = *None* $\Longrightarrow$
       C !$\in$! *raw-clauses S* $\Longrightarrow$
       L $\in\#$ *mset-cls C* $\Longrightarrow$
       trail S $\models$*as CNot* (*remove1-mset L* (*mset-cls C*)) $\Longrightarrow$
       undefined-lit (*trail S*) *L* $\Longrightarrow$
       T $\sim$ *cons-trail* (*Propagated L C*) *S* $\Longrightarrow$
       P S T **and**
    conflictH: $\bigwedge$*D T*. *conflicting S* = *None* $\Longrightarrow$
       D !$\in$! *raw-clauses S* $\Longrightarrow$
       trail S $\models$*as CNot* (*mset-cls D*) $\Longrightarrow$
       T $\sim$ *update-conflicting* (*Some* (*ccls-of-cls D*)) *S* $\Longrightarrow$
       P S T **and**
    forgetH: $\bigwedge$*C T*. *conflicting S* = *None* $\Longrightarrow$
      C !$\in$! *raw-learned-clss S* $\Longrightarrow$
      $\neg$(*trail S*) $\models$*asm clauses S* $\Longrightarrow$
      mset-cls C $\notin$ *set* (*get-all-mark-of-propagated* (*trail S*)) $\Longrightarrow$
      mset-cls C $\notin\#$ *init-clss S* $\Longrightarrow$
      T $\sim$ *remove-cls C S* $\Longrightarrow$
      P S T **and**
    restartH: $\bigwedge$*T*. $\neg$*trail S* $\models$*asm clauses S* $\Longrightarrow$
      conflicting S = *None* $\Longrightarrow$
      T $\sim$ *restart-state S* $\Longrightarrow$
      P S T **and**
    decideH: $\bigwedge$*L T*. *conflicting S* = *None* $\Longrightarrow$
      undefined-lit (*trail S*) *L* $\Longrightarrow$
      atm-of L $\in$ *atms-of-mm* (*init-clss S*) $\Longrightarrow$
      T $\sim$ *cons-trail* (*Decided L* (*backtrack-lvl S* +*1*)) (*incr-lvl S*) $\Longrightarrow$

$P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
trail S = *Propagated* $L\ C'$ # $M \Longrightarrow$
raw-conflicting S = *Some* $E \Longrightarrow$
$-L \notin\#$ *mset-ccls* $E \Longrightarrow$ *mset-ccls* $E \neq \{\#\} \Longrightarrow$
$T \sim$ *tl-trail* $S \Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T.$
trail S = *Propagated* L (*mset-cls* E) # $M \Longrightarrow$
$L \in\#$ *mset-cls* $E \Longrightarrow$
hd-raw-trail S = *Propagated* $L\ E \Longrightarrow$
raw-conflicting S = *Some* $D \Longrightarrow$
$-L \in\#$ *mset-ccls* $D \Longrightarrow$
get-maximum-level (*trail* S) (*mset-ccls* (*remove-clit* $(-L)$ D)) = *backtrack-lvl* $S \Longrightarrow$
$T \sim$ *update-conflicting*
(*Some* (*union-ccls* (*remove-clit* $(-L)$ D) (*ccls-of-cls* (*remove-lit* $L\ E$)))) (*tl-trail* S) $\Longrightarrow$
$P\ S\ T$ **and**
backtrackH: $\bigwedge K\ i\ M1\ M2\ L\ D\ T.$
raw-conflicting S = *Some* $D \Longrightarrow$
$L \in\#$ *mset-ccls* $D \Longrightarrow$
(*Decided* K (*Suc* i) # $M1$, $M2$) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$
get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
get-level (*trail* S) L = *get-maximum-level* (*trail* S) (*mset-ccls* D) $\Longrightarrow$
get-maximum-level (*trail* S) (*remove1-mset* L (*mset-ccls* D)) $\equiv i \Longrightarrow$
undefined-lit $M1\ L \Longrightarrow$
$T \sim$ *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)))) $\Longrightarrow$
$P\ S\ T$
shows $P\ S\ S'$
⟨*proof*⟩

lemmas *cdcl*$_W$*-all-induct-lev2* = *cdcl*$_W$*-all-induct-lev-full*[*consumes 2*, *case-names propagate conflict forget restart decide skip resolve backtrack*]

lemmas *cdcl*$_W$*-all-induct-lev* = *cdcl*$_W$*-all-induct-lev-full*[*consumes 1*, *case-names lev-inv propagate conflict forget restart decide skip resolve backtrack*]

thm *cdcl*$_W$*-o-induct*
lemma *cdcl*$_W$*-o-induct-lev*[*consumes 1*, *case-names M-lev decide skip resolve backtrack*]:
fixes S :: *'st*
assumes
cdcl$_W$: *cdcl*$_W$*-o* $S\ T$ **and**
inv[*simp*]: *cdcl*$_W$*-M-level-inv* S **and**
decideH: $\bigwedge L\ T.$ *conflicting* S = *None* $\Longrightarrow$
undefined-lit (*trail* S) $L \Longrightarrow$
atm-of $L \in$ *atms-of-mm* (*init-clss* S) $\Longrightarrow$
$T \sim$ *cons-trail* (*Decided* L (*backtrack-lvl* S *+1*)) (*incr-lvl* S) $\Longrightarrow$
$P\ S\ T$ **and**
skipH: $\bigwedge L\ C'\ M\ E\ T.$
trail S = *Propagated* $L\ C'$ # $M \Longrightarrow$
raw-conflicting S = *Some* $E \Longrightarrow$
$-L \notin\#$ *mset-ccls* $E \Longrightarrow$ *mset-ccls* $E \neq \{\#\} \Longrightarrow$

$T \sim$ *tl-trail* $S \Longrightarrow$
$P\ S\ T$ **and**
resolveH: $\bigwedge L\ E\ M\ D\ T$.
trail $S =$ *Propagated* L (*mset-cls* E) # $M \Longrightarrow$
$L \in\#$ *mset-cls* $E \Longrightarrow$
hd-raw-trail $S =$ *Propagated* $L\ E \Longrightarrow$
raw-conflicting $S =$ *Some* $D \Longrightarrow$
$-L \in\#$ *mset-ccls* $D \Longrightarrow$
get-maximum-level (*trail* S) (*mset-ccls* (*remove-clit* $(-L)$ D)) = *backtrack-lvl* $S \Longrightarrow$
$T \sim$ *update-conflicting*
(*Some* (*union-ccls* (*remove-clit* $(-L)$ D) (*ccls-of-cls* (*remove-lit* $L\ E$)))) (*tl-trail* S) $\Longrightarrow$
$P\ S\ T$ **and**
backtrackH: $\bigwedge K\ i\ M1\ M2\ L\ D\ T$.
raw-conflicting $S =$ *Some* $D \Longrightarrow$
$L \in\#$ *mset-ccls* $D \Longrightarrow$
(*Decided* K (*Suc* i) # $M1$, $M2$) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$
get-level (*trail* S) L = *backtrack-lvl* $S \Longrightarrow$
get-level (*trail* S) L = *get-maximum-level* (*trail* S) (*mset-ccls* D) $\Longrightarrow$
get-maximum-level (*trail* S) (*remove1-mset* L (*mset-ccls* D)) $\equiv i \Longrightarrow$
undefined-lit $M1\ L \Longrightarrow$
$T \sim$ *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)))) $\Longrightarrow$
$P\ S\ T$
shows $P\ S\ T$
⟨*proof*⟩

lemmas $cdcl_W$*-o-induct-lev2* = $cdcl_W$*-o-induct-lev*[*consumes 2*, *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* $S'\ T'$
⟨*proof*⟩

lemma *conflict-state-eq-compatible*:
assumes
confl: *conflict* $S\ T$ **and**
TT': $T \sim T'$ **and**
SS': $S \sim S'$
shows *conflict* $S'\ T'$
⟨*proof*⟩

lemma *backtrack-levE*[*consumes 2*]:
backtrack $S\ S' \Longrightarrow$ $cdcl_W$*-M-level-inv* $S \Longrightarrow$
($\bigwedge K\ i\ M1\ M2\ L\ D$.
raw-conflicting $S =$ *Some* $D \Longrightarrow$
$L \in\#$ *mset-ccls* $D \Longrightarrow$
(*Decided* K (*Suc* i) # $M1$, $M2$) $\in$ *set* (*get-all-ann-decomposition* (*trail* S)) $\Longrightarrow$

$get\text{-}level\ (trail\ S)\ L = backtrack\text{-}lvl\ S \Longrightarrow$
$get\text{-}level\ (trail\ S)\ L = get\text{-}maximum\text{-}level\ (trail\ S)\ (mset\text{-}ccls\ D) \Longrightarrow$
$get\text{-}maximum\text{-}level\ (trail\ S)\ (remove1\text{-}mset\ L\ (mset\text{-}ccls\ D)) \equiv i \Longrightarrow$
$undefined\text{-}lit\ M1\ L \Longrightarrow$
$S' \sim cons\text{-}trail\ (Propagated\ L\ (cls\text{-}of\text{-}ccls\ D))$
$(reduce\text{-}trail\text{-}to\ M1$
$(add\text{-}learned\text{-}cls\ (cls\text{-}of\text{-}ccls\ D)$
$(update\text{-}backtrack\text{-}lvl\ i$
$(update\text{-}conflicting\ None\ S)))) \Longrightarrow P) \Longrightarrow$
P
⟨*proof*⟩

lemma *backtrack-state-eq-compatible*:
assumes
bt: *backtrack S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$ **and**
inv: $cdcl_W$*-M-level-inv S*
shows *backtrack S' T'*
⟨*proof*⟩

lemma *decide-state-eq-compatible*:
assumes
decide S T **and**
$S \sim S'$ **and**
$T \sim T'$
shows *decide S' T'*
⟨*proof*⟩

lemma *skip-state-eq-compatible*:
assumes
skip: *skip S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$
shows *skip S' T'*
⟨*proof*⟩

lemma *resolve-state-eq-compatible*:
assumes
res: *resolve S T* **and**
TT': $T \sim T'$ **and**
SS': $S \sim S'$
shows *resolve S' T'*
⟨*proof*⟩

lemma *forget-state-eq-compatible*:
assumes
forget: *forget S T* **and**
SS': $S \sim S'$ **and**
TT': $T \sim T'$
shows *forget S' T'*
⟨*proof*⟩

lemma $cdcl_W$*-state-eq-compatible*:

assumes
$cdcl_W$ S T **and** $\neg$*restart* S T **and**
$S \sim S'$
$T \sim T'$ **and**
$cdcl_W$*-M-level-inv* S
shows $cdcl_W$ S' T'
⟨*proof*⟩

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'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-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'
⟨*proof*⟩

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'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-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 *rtranclp-*$cdcl_W$*-o-no-more-init-clss*:
assumes
$cdcl_W$*-o*** S S' **and**
inv: $cdcl_W$*-M-level-inv* S
shows *init-clss* S = *init-clss* S'
⟨*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 *rtranclp-*$cdcl_W$*-init-clss*:
$cdcl_W{}^{**}$ S $T \Longrightarrow$ $cdcl_W$*-M-level-inv* $S \Longrightarrow$ *init-clss* S = *init-clss* T
⟨*proof*⟩

lemma *tranclp-cdcl*$_W$*-init-clss*:
$cdcl_W{}^{++}$ S T $\Longrightarrow$ $cdcl_W$*-M-level-inv* S $\Longrightarrow$ *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 decided 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. *conflicting* S = *Some* T $\longrightarrow$ *init-clss* S $\models$*pm* T)
$\wedge$ *set* (*get-all-mark-of-propagated* (*trail* S)) $\subseteq$ *set-mset* (*clauses* S))

lemma $cdcl_W$*-learned-clause-S0-cdcl*$_W$[*simp*]:
$cdcl_W$*-learned-clause* (*init-state* N)
⟨*proof*⟩

Item 4 page 81 of Weidenbach's book and Item 4 page 81 of Weidenbach's book

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'
⟨*proof*⟩

lemma *rtranclp-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'
⟨*proof*⟩

19.3.6 No alien atom in the state

This invariant means that all the literals are in the set of clauses. They are implicit in Weidenbach's book.

definition *no-strange-atm* S' $\longleftrightarrow$ (
($\forall$ T. *conflicting* S' = *Some* T $\longrightarrow$ *atms-of* T $\subseteq$ *atms-of-mm* (*init-clss* S'))
$\wedge$ ($\forall$ L *mark*. *Propagated* L *mark* $\in$ *set* (*trail* S')
$\longrightarrow$ *atms-of* (*mark*) $\subseteq$ *atms-of-mm* (*init-clss* S'))
$\wedge$ *atms-of-mm* (*learned-clss* S') $\subseteq$ *atms-of-mm* (*init-clss* S')
$\wedge$ *atm-of* ` (*lits-of-l* (*trail* S')) $\subseteq$ *atms-of-mm* (*init-clss* S'))

lemma *no-strange-atm-decomp*:
assumes *no-strange-atm S*
shows *conflicting S = Some T* $\Longrightarrow$ *atms-of T* $\subseteq$ *atms-of-mm* (*init-clss S*)
and ($\forall$ *L mark. Propagated L mark* $\in$ *set* (*trail S*)
$\longrightarrow$ *atms-of* (*mark*) $\subseteq$ *atms-of-mm* (*init-clss S*))
and *atms-of-mm* (*learned-clss S*) $\subseteq$ *atms-of-mm* (*init-clss S*)
and *atm-of* ‘ (*lits-of-l* (*trail S*)) $\subseteq$ *atms-of-mm* (*init-clss S*)
⟨*proof*⟩

lemma *no-strange-atm-S0* [*simp*]: *no-strange-atm* (*init-state N*)
⟨*proof*⟩

lemma *in-atms-of-implies-atm-of-on-atms-of-ms*:
C + {#*L*#} $\in$# *A* $\Longrightarrow$ *x* $\in$ *atms-of C* $\Longrightarrow$ *x* $\in$ *atms-of-mm A*
⟨*proof*⟩

lemma *propagate-no-strange-atm-inv*:
assumes
propagate S T **and**
alien: *no-strange-atm S*
shows *no-strange-atm T*
⟨*proof*⟩

lemma *in-atms-of-remove1-mset-in-atms-of*:
x $\in$ *atms-of* (*remove1-mset L C*) $\Longrightarrow$ *x* $\in$ *atms-of C*
⟨*proof*⟩

lemma *cdcl*$_W$*-no-strange-atm-explicit*:
assumes
cdcl$_W$ *S S'* **and**
lev: *cdcl*$_W$*-M-level-inv S* **and**
conf: $\forall$ *T. conflicting S = Some T* $\longrightarrow$ *atms-of T* $\subseteq$ *atms-of-mm* (*init-clss S*) **and**
decided: $\forall$ *L mark. Propagated L mark* $\in$ *set* (*trail S*)
$\longrightarrow$ *atms-of mark* $\subseteq$ *atms-of-mm* (*init-clss S*) **and**
learned: *atms-of-mm* (*learned-clss S*) $\subseteq$ *atms-of-mm* (*init-clss S*) **and**
trail: *atm-of* ‘ (*lits-of-l* (*trail S*)) $\subseteq$ *atms-of-mm* (*init-clss S*)
shows
($\forall$ *T. conflicting S' = Some T* $\longrightarrow$ *atms-of T* $\subseteq$ *atms-of-mm* (*init-clss S'*)) $\wedge$
($\forall$ *L mark. Propagated L mark* $\in$ *set* (*trail S'*)
$\longrightarrow$ *atms-of* (*mark*) $\subseteq$ *atms-of-mm* (*init-clss S'*)) $\wedge$
atms-of-mm (*learned-clss S'*) $\subseteq$ *atms-of-mm* (*init-clss S'*) $\wedge$
atm-of ‘ (*lits-of-l* (*trail S'*)) $\subseteq$ *atms-of-mm* (*init-clss S'*)
(**is** *?C S'* $\wedge$ *?M S'* $\wedge$ *?U S'* $\wedge$ *?V S'*)
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-no-strange-atm-inv*:
assumes *cdcl*$_W$** *S S'* **and** *no-strange-atm S* **and** *cdcl*$_W$*-M-level-inv S*
shows *no-strange-atm S'*
⟨*proof*⟩

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.

definition *distinct-cdcl$_W$-state* $(S :: 'st)$
$\longleftrightarrow$ $((\forall\, T.\ \textit{conflicting}\ S = \textit{Some}\ T \longrightarrow \textit{distinct-mset}\ T)$
$\wedge$ *distinct-mset-mset* (*learned-clss* S)
$\wedge$ *distinct-mset-mset* (*init-clss* S)
$\wedge$ $(\forall\, L\ \textit{mark}.\ (\textit{Propagated}\ L\ \textit{mark} \in \textit{set}\ (\textit{trail}\ S) \longrightarrow \textit{distinct-mset}\ \textit{mark})))$

lemma *distinct-cdcl$_W$-state-decomp*:
assumes *distinct-cdcl$_W$-state* $(S :: 'st)$
shows
$\forall\, T.\ \textit{conflicting}\ S = \textit{Some}\ T \longrightarrow \textit{distinct-mset}\ T$ **and**
distinct-mset-mset (*learned-clss* S) **and**
distinct-mset-mset (*init-clss* S) **and**
$\forall\, L\ \textit{mark}.\ (\textit{Propagated}\ L\ \textit{mark} \in \textit{set}\ (\textit{trail}\ S) \longrightarrow \textit{distinct-mset}\ (\ \textit{mark}))$
⟨*proof*⟩

lemma *distinct-cdcl$_W$-state-decomp-2*:
assumes *distinct-cdcl$_W$-state* $(S :: 'st)$ **and** *conflicting* $S = $ *Some* T
shows *distinct-mset* T
⟨*proof*⟩

lemma *distinct-cdcl$_W$-state-S0-cdcl$_W$* [*simp*]:
distinct-mset-mset (*mset-clss* N) $\Longrightarrow$ *distinct-cdcl$_W$-state* (*init-state* N)
⟨*proof*⟩

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'
⟨*proof*⟩

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

This invariant shows that each mark contains a contradiction only related to the previously defined variable.

abbreviation *every-mark-is-a-conflict* :: $'st \Rightarrow \textit{bool}$ **where**
every-mark-is-a-conflict $S \equiv$
$\forall\, L\ \textit{mark}\ a\ b.\ a\ @\ \textit{Propagated}\ L\ \textit{mark}\ \#\ b = (\textit{trail}\ S)$
$\longrightarrow (b \models\!as\ \textit{CNot}\ (\textit{mark} - \{\#L\#\}) \wedge L \in\!\#\ \ \textit{mark})$

definition *cdcl$_W$-conflicting* $S \longleftrightarrow$

($\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*as CNot T*)
$\wedge$ *every-mark-is-a-conflict S*

lemma *backtrack-atms-of-D-in-M1*:
fixes *M1* :: (*'v*, *nat*, *'v clause*) *ann-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: (*Decided K* (*Suc i*) # *M1*, *M2*)
$\in$ *set* (*get-all-ann-decomposition* (*trail S*)) **and**
S-lvl: *backtrack-lvl S* = *get-maximum-level* (*trail S*) (*mset-ccls D*) **and**
S-confl: *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**
confl: $\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*as 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*⟩

Item 5 page 81 of Weidenbach's book

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-clss S*) (*get-all-ann-decomposition* (*trail S*)) **and**
learned: *cdcl*$_W$*-learned-clause S* **and**
confl: $\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*as CNot T* **and**
alien: *no-strange-atm S*
shows *all-decomposition-implies-m* (*init-clss S'*) (*get-all-ann-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-ann-decomposition* (*trail S*)) **and**
confl: $\forall$ *T*. *conflicting S* = *Some T* $\longrightarrow$ *trail S* $\models$*as CNot T* **and**
alien: *no-strange-atm S* **and**
mark-confl: *every-mark-is-a-conflict S*
shows *every-mark-is-a-conflict S'*
⟨*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. *conflicting* S = *Some* T $\longrightarrow$ *trail* S $\models$*as CNot* T **and**
decided-confl: $\forall$ L *mark* a b. a @ *Propagated* L *mark* # b = (*trail* S)
$\longrightarrow$ (b $\models$*as CNot* (*mark* − {#L#}) $\wedge$ L $\in$# *mark*) **and**
dist: *distinct-cdcl*$_W$*-state* S
shows $\forall$ T. *conflicting* S' = *Some* T $\longrightarrow$ *trail* S' $\models$*as CNot* T
⟨*proof*⟩

lemma *cdcl*$_W$*-conflicting-decomp*:
assumes *cdcl*$_W$*-conflicting* S
shows $\forall$ T. *conflicting* S = *Some* T $\longrightarrow$ *trail* S $\models$*as CNot* T
and $\forall$ L *mark* a b. a @ *Propagated* L *mark* # b = (*trail* S)
$\longrightarrow$ (b $\models$*as CNot* (*mark* − {#L#}) $\wedge$ L $\in$# *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-ann-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-ann-decomposition* (*trail* S')) **and**
cdcl$_W$*-learned-clause* S' **and**
cdcl$_W$*-M-level-inv* S' **and**
no-strange-atm S' **and**
distinct-cdcl$_W$*-state* S' **and**
cdcl$_W$*-conflicting* S'
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-all-inv*:
assumes
cdcl$_W$: *rtranclp cdcl*$_W$ S S' **and**
1: *all-decomposition-implies-m* (*init-clss* S) (*get-all-ann-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-ann-decomposition* (*trail* S')) **and**
$cdcl_W$*-learned-clause* S' **and**
$cdcl_W$*-M-level-inv* S' **and**
no-strange-atm S' **and**
distinct-$cdcl_W$*-state* S' **and**
$cdcl_W$*-conflicting* S'
⟨*proof*⟩

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-ann-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 b. a* @ *Propagated L mark* # *b* = *trail* (*init-state* N) $\longrightarrow$
($b \models$*as CNot* (*mark* $-$ {#L#}) $\wedge$ $L \in\#$ *mark*) **and**
distinct-$cdcl_W$*-state* (*init-state* N)
⟨*proof*⟩

Item 6 page 81 of Weidenbach's book

lemma $cdcl_W$*-only-propagated-vars-unsat*:
assumes
decided: $\forall x \in$ *set* M. $\neg$ *is-decided* x **and**
DN: $D \in\#$ *clauses* S **and**
D: $M \models$*as CNot* D **and**
inv: *all-decomposition-implies-m* N (*get-all-ann-decomposition* M) **and**
state: *state* S = (M, N, U, k, C) **and**
learned-cl: $cdcl_W$*-learned-clause* S **and**
atm-incl: *no-strange-atm* S
shows *unsatisfiable* (*set-mset* N)
⟨*proof*⟩

Item 5 page 81 of Weidenbach's book

We have actually a much stronger theorem, namely *all-decomposition-implies ?N* (*get-all-ann-decomposition ?M*) $\Longrightarrow$ *?N* $\cup$ {*unmark L* |*L. is-decided L* $\wedge$ $L \in$ *set ?M*} $\models$*ps unmark-l ?M*, that show that the only choices we made are decided in the formula

lemma
assumes *all-decomposition-implies-m* N (*get-all-ann-decomposition* M)
and $\forall m \in$ *set* M. $\neg$*is-decided* m
shows *set-mset* $N \models$*ps unmark-l* M
⟨*proof*⟩

Item 7 page 81 of Weidenbach's book (part 1)

lemma *conflict-with-false-implies-unsat*:
assumes
$cdcl_W$: $cdcl_W$ S S' **and**
lev: $cdcl_W$*-M-level-inv* S **and**
[*simp*]: *conflicting* S' = *Some* {#} **and**
learned: $cdcl_W$*-learned-clause* S
shows *unsatisfiable* (*set-mset* (*init-clss* S))

$\langle proof \rangle$

Item 7 page 81 of Weidenbach's book (part 2)

lemma *conflict-with-false-implies-terminated*:
assumes $cdcl_W\ S\ S'$
and *conflicting* $S = Some\ \{\#\}$
shows *False*
$\langle proof \rangle$

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
$cdcl_W\ S\ S'$ **and**
lev: $cdcl_W$*-M-level-inv* S **and**
conflicting: $cdcl_W$*-conflicting* S **and**
no-tauto: $\forall\ s \in\#\ learned\text{-}clss\ S.\ \neg tautology\ s$
shows $\forall\ s \in\#\ learned\text{-}clss\ S'.\ \neg tautology\ s$
$\langle proof \rangle$

definition *final-*$cdcl_W$*-state* $(S :: 'st)$
$\longleftrightarrow (trail\ S \models asm\ init\text{-}clss\ S$
$\lor ((\forall\ L \in set\ (trail\ S).\ \neg is\text{-}decided\ L) \land$
$(\exists\ C \in\#\ init\text{-}clss\ S.\ trail\ S \models as\ CNot\ C)))$

definition *termination-*$cdcl_W$*-state* $(S :: 'st)$
$\longleftrightarrow (trail\ S \models asm\ init\text{-}clss\ S$
$\lor ((\forall\ L \in atms\text{-}of\text{-}mm\ (init\text{-}clss\ S).\ L \in atm\text{-}of\ `\ lits\text{-}of\text{-}l\ (trail\ S))$
$\land (\exists\ C \in\#\ init\text{-}clss\ S.\ trail\ S \models as\ CNot\ C)))$

19.4 CDCL Strong Completeness

fun *mapi* $:: ('a \Rightarrow nat \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
$mapi\ \text{-}\ \text{-}\ [] = []\ |$
$mapi\ f\ n\ (x\ \#\ xs) = f\ x\ n\ \#\ mapi\ f\ (n - 1)\ xs$

lemma *mark-not-in-set-mapi*[*simp*]: $L \notin set\ M \Longrightarrow Decided\ L\ k \notin set\ (mapi\ Decided\ i\ M)$
$\langle proof \rangle$

lemma *propagated-not-in-set-mapi*[*simp*]: $L \notin set\ M \Longrightarrow Propagated\ L\ k \notin set\ (mapi\ Decided\ i\ M)$
$\langle proof \rangle$

lemma *image-set-mapi*:
$f\ `\ set\ (mapi\ g\ i\ M) = set\ (mapi\ (\lambda x\ i.\ f\ (g\ x\ i))\ i\ M)$
$\langle proof \rangle$

lemma *mapi-map-convert*:
$\forall\ x\ i\ j.\ f\ x\ i = f\ x\ j \Longrightarrow mapi\ f\ i\ M = map\ (\lambda x.\ f\ x\ 0)\ M$
$\langle proof \rangle$

lemma *defined-lit-mapi*: $defined\text{-}lit\ (mapi\ Decided\ i\ M)\ L \longleftrightarrow atm\text{-}of\ L \in atm\text{-}of\ `\ set\ M$
$\langle proof \rangle$

lemma *cdcl$_W$-can-do-step*:
assumes
consistent-interp (*set M*) **and**
distinct M **and**
atm-of ‘ (*set M*) $\subseteq$ *atms-of-mm* (*mset-clss N*)
shows $\exists$ *S*. *rtranclp cdcl$_W$* (*init-state N*) *S*
$\wedge$ *state S* = (*mapi Decided* (*length M*) *M*, *mset-clss N*, {#}, *length M*, *None*)
⟨*proof*⟩

theorem 2.9.11 page 84 of Weidenbach's book

lemma *cdcl$_W$-strong-completeness*:
assumes
MN: *set M* $\models$*sm mset-clss N* **and**
cons: *consistent-interp* (*set M*) **and**
dist: *distinct M* **and**
atm: *atm-of* ‘ (*set M*) $\subseteq$ *atms-of-mm* (*mset-clss N*)
obtains *S* **where**
state S = (*mapi Decided* (*length M*) *M*, *mset-clss N*, {#}, *length M*, *None*) **and**
rtranclp 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 *tranclp-conflict*:
tranclp conflict S S' $\Longrightarrow$ *conflict S S'*
⟨*proof*⟩

lemma *tranclp-conflict-iff*[*iff*]:
full1 conflict S S' $\longleftrightarrow$ *conflict S S'*
⟨*proof*⟩

inductive *cdcl$_W$-cp* :: *'st* $\Rightarrow$ *'st* $\Rightarrow$ *bool* **where**
conflict'[*intro*]: *conflict S S'* $\Longrightarrow$ *cdcl$_W$-cp S S'* |
propagate': *propagate S S'* $\Longrightarrow$ *cdcl$_W$-cp S S'*

lemma *rtranclp-cdcl$_W$-cp-rtranclp-cdcl$_W$*:
cdcl$_W$-cp** *S T* $\Longrightarrow$ *cdcl$_W$*** *S T*
⟨*proof*⟩

lemma *cdcl$_W$-cp-state-eq-compatible*:
assumes
cdcl$_W$-cp S T **and**
S $\sim$ *S'* **and**
T $\sim$ *T'*
shows *cdcl$_W$-cp S' T'*
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-state-eq-compatible*:
assumes
cdcl$_W$-cp^{++} *S T* **and**

$S \sim S'$ **and**
$T \sim T'$
shows $cdcl_W\text{-}cp^{++}\ S'\ T'$
⟨*proof*⟩

lemma *option-full-cdcl$_W$-cp*:
$conflicting\ S \neq None \Longrightarrow full\ cdcl_W\text{-}cp\ S\ S$
⟨*proof*⟩

lemma *skip-unique*:
$skip\ S\ T \Longrightarrow skip\ S\ T' \Longrightarrow T \sim T'$
⟨*proof*⟩

lemma *resolve-unique*:
$resolve\ S\ T \Longrightarrow resolve\ S\ T' \Longrightarrow T \sim T'$
⟨*proof*⟩

lemma *cdcl$_W$-cp-no-more-clauses*:
assumes $cdcl_W\text{-}cp\ S\ S'$
shows $clauses\ S = clauses\ S'$
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-no-more-clauses*:
assumes $cdcl_W\text{-}cp^{++}\ S\ S'$
shows $clauses\ S = clauses\ S'$
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-no-more-clauses*:
assumes $cdcl_W\text{-}cp^{**}\ S\ S'$
shows $clauses\ S = clauses\ S'$
⟨*proof*⟩

lemma *no-conflict-after-conflict*:
$conflict\ S\ T \Longrightarrow \neg conflict\ T\ U$
⟨*proof*⟩

lemma *no-propagate-after-conflict*:
$conflict\ S\ T \Longrightarrow \neg propagate\ T\ U$
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-propagate-with-conflict-or-not*:
assumes $cdcl_W\text{-}cp^{++}\ S\ U$
shows $(propagate^{++}\ S\ U \wedge conflicting\ U = None)$
$\vee (\exists\ T\ D.\ propagate^{**}\ S\ T \wedge conflict\ T\ U \wedge conflicting\ U = Some\ D)$
⟨*proof*⟩

lemma *cdcl$_W$-cp-conflicting-not-empty*[*simp*]: $conflicting\ S = Some\ D \Longrightarrow \neg cdcl_W\text{-}cp\ S\ S'$
⟨*proof*⟩

lemma *no-step-cdcl$_W$-cp-no-conflict-no-propagate*:
assumes $no\text{-}step\ cdcl_W\text{-}cp\ S$
shows $no\text{-}step\ conflict\ S$ **and** $no\text{-}step\ propagate\ S$
⟨*proof*⟩

CDCL with the reasonable strategy: we fully propagate the conflict and propagate, then we apply any other possible rule $cdcl_W\text{-}o\ S\ S'$ and re-apply conflict and propagate $cdcl_W\text{-}cp^{\downarrow}\ S'$

S''

inductive $cdcl_W$*-stgy* :: *'st* ⇒ *'st* ⇒ *bool* **for** S :: *'st* **where**
conflict': *full1* $cdcl_W$*-cp* S S' ⟹ $cdcl_W$*-stgy* S S' |
other': $cdcl_W$*-o* S S' ⟹ *no-step* $cdcl_W$*-cp* S ⟹ *full* $cdcl_W$*-cp* S' S'' ⟹ $cdcl_W$*-stgy* S S''

19.5.2 Invariants

These are the same invariants as before, but lifted

lemma $cdcl_W$*-cp-learned-clause-inv*:
assumes $cdcl_W$*-cp* S S'
shows *learned-clss* S = *learned-clss* S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-learned-clause-inv*:
assumes $cdcl_W$*-cp*** S S'
shows *learned-clss* S = *learned-clss* S'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-cp-learned-clause-inv*:
assumes $cdcl_W$*-cp*$^{++}$ S S'
shows *learned-clss* S = *learned-clss* S'
⟨*proof*⟩

lemma $cdcl_W$*-cp-backtrack-lvl*:
assumes $cdcl_W$*-cp* S S'
shows *backtrack-lvl* S = *backtrack-lvl* S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-backtrack-lvl*:
assumes $cdcl_W$*-cp*** S S'
shows *backtrack-lvl* S = *backtrack-lvl* S'
⟨*proof*⟩

lemma $cdcl_W$*-cp-consistent-inv*:
assumes $cdcl_W$*-cp* S S'
and $cdcl_W$*-M-level-inv* S
shows $cdcl_W$*-M-level-inv* S'
⟨*proof*⟩

lemma *full1-*$cdcl_W$*-cp-consistent-inv*:
assumes *full1* $cdcl_W$*-cp* S S'
and $cdcl_W$*-M-level-inv* S
shows $cdcl_W$*-M-level-inv* S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-consistent-inv*:
assumes *rtranclp* $cdcl_W$*-cp* S S'
and $cdcl_W$*-M-level-inv* S
shows $cdcl_W$*-M-level-inv* S'
⟨*proof*⟩

lemma $cdcl_W$*-stgy-consistent-inv*:
assumes $cdcl_W$*-stgy* S S'
and $cdcl_W$*-M-level-inv* S
shows $cdcl_W$*-M-level-inv* S'

⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-stgy-consistent-inv*:
assumes *cdcl$_W$-stgy** S S'*
and *cdcl$_W$-M-level-inv S*
shows *cdcl$_W$-M-level-inv S'*
⟨*proof*⟩

lemma *cdcl$_W$-cp-no-more-init-clss*:
assumes *cdcl$_W$-cp S S'*
shows *init-clss S = init-clss S'*
⟨*proof*⟩

lemma *tranclp-cdcl$_W$-cp-no-more-init-clss*:
assumes *cdcl$_W$-cp^{++} S S'*
shows *init-clss S = init-clss S'*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-no-more-init-clss*:
assumes *cdcl$_W$-stgy S S'* **and** *cdcl$_W$-M-level-inv S*
shows *init-clss S = init-clss S'*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-stgy-no-more-init-clss*:
assumes *cdcl$_W$-stgy** S S'* **and** *cdcl$_W$-M-level-inv S*
shows *init-clss S = init-clss S'*
⟨*proof*⟩

lemma *cdcl$_W$-cp-dropWhile-trail'*:
assumes *cdcl$_W$-cp S S'*
obtains *M* **where** *trail S' = M @ trail S* **and** (∀ *l* ∈ *set M*. ¬*is-decided l*)
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail'*:
assumes *cdcl$_W$-cp** S S'*
obtains *M* :: (*'v*, *nat*, *'v clause*) *ann-lit list* **where**
trail S' = M @ trail S **and** ∀ *l* ∈ *set M*. ¬*is-decided l*
⟨*proof*⟩

lemma *cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp S S'*
shows ∃ *M*. *trail S' = M @ trail S* ∧ (∀ *l* ∈ *set M*. ¬*is-decided l*)
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-dropWhile-trail*:
assumes *cdcl$_W$-cp** S S'*
shows ∃ *M*. *trail S' = M @ trail S* ∧ (∀ *l* ∈ *set M*. ¬*is-decided l*)
⟨*proof*⟩

This theorem can be seen a 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 *length* (*trail* S) $\leq$ *card* (*atms-of-mm* (*init-clss* S))
⟨*proof*⟩

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 (λS. *card* (*atms-of-mm* (*init-clss* S)) − *length* (*trail* S)
+ (*if conflicting* S = *None then 1 else 0*)) S
> (λS. *card* (*atms-of-mm* (*init-clss* S)) − *length* (*trail* S)
+ (*if conflicting* S = *None then 1 else 0*)) T
⟨*proof*⟩

lemma $cdcl_W$*-cp-wf*: *wf* {(b,a). ($cdcl_W$*-M-level-inv* a ∧ *no-strange-atm* a)
∧ $cdcl_W$*-cp* a b}
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-all-struct-inv-*$cdcl_W$*-cp-iff-rtranclp-*$cdcl_W$*-cp*:
assumes
lev: $cdcl_W$*-M-level-inv* S **and**
alien: *no-strange-atm* S
shows (λa b. ($cdcl_W$*-M-level-inv* a ∧ *no-strange-atm* a) ∧ $cdcl_W$*-cp* a b)** S T
⟷ $cdcl_W$*-cp*** S T
(**is** *?I* S T ⟷ *?C* S T)
⟨*proof*⟩

lemma $cdcl_W$*-cp-normalized-element*:
assumes
lev: $cdcl_W$*-M-level-inv* S **and**
no-strange-atm S
obtains T **where** *full* $cdcl_W$*-cp* S T
⟨*proof*⟩

lemma *always-exists-full-*$cdcl_W$*-cp-step*:
assumes *no-strange-atm* S
shows ∃ S''. *full* $cdcl_W$*-cp* S S''
⟨*proof*⟩

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* ⇒ *bool* **where**
no-clause-is-false ≡
λS. (*conflicting* S = *None* ⟶ (∀ D ∈# *clauses* S. ¬*trail* S ⊨*as CNot* D))

abbreviation *conflict-is-false-with-level* :: *'st* ⇒ *bool* **where**
conflict-is-false-with-level S ≡ ∀ D. *conflicting* S = *Some* D ⟶ D ≠ {#}
⟶ (∃ L ∈# D. *get-level* (*trail* S) L = *backtrack-lvl* S)

lemma *not-conflict-not-any-negated-init-clss*:
assumes ∀ S'. ¬*conflict* S S'
shows *no-clause-is-false* S

⟨*proof*⟩

lemma *full-cdcl*$_W$*-cp-not-any-negated-init-clss*:
assumes *full cdcl*$_W$*-cp S S'*
shows *no-clause-is-false S'*
⟨*proof*⟩

lemma *full1-cdcl*$_W$*-cp-not-any-negated-init-clss*:
assumes *full1 cdcl*$_W$*-cp S S'*
shows *no-clause-is-false S'*
⟨*proof*⟩

lemma *cdcl*$_W$*-stgy-not-non-negated-init-clss*:
assumes *cdcl*$_W$*-stgy S S'*
shows *no-clause-is-false S'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *no-chained-conflict*:
assumes *conflict S S'*
and *conflict S' S''*
shows *False*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-cp-propa-or-propa-confl*:
assumes *cdcl*$_W$*-cp*** *S U*
shows *propagate*** $S\ U \vee (\exists\, T.\ propagate^{**}\ S\ T\ \wedge\ conflict\ T\ U)$
⟨*proof*⟩

lemma *rtranclp-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 decided literals

definition *mark-is-false-with-level* :: *'st* ⇒ *bool* **where**
mark-is-false-with-level S' ≡
$\forall\, D\ M1\ M2\ L.\ \ M1\ @\ Propagated\ L\ D\ \#\ M2 = trail\ S' \longrightarrow\ \ D - \{\#L\#\} \neq \{\#\}$
$\longrightarrow (\exists\, L.\ L \in\!\#\ \ D \wedge get\text{-}level\ (trail\ S')\ L = get\text{-}maximum\text{-}possible\text{-}level\ M1)$

definition *no-more-propagation-to-do* :: *'st* ⇒ *bool* **where**

no-more-propagation-to-do $S \equiv$
$\forall D\ M\ M'\ L.\ D + \{\#L\#\} \in\#\ clauses\ S \longrightarrow trail\ S = M' \ @\ M \longrightarrow M \models as\ CNot\ D$
$\longrightarrow undefined\text{-}lit\ M\ L \longrightarrow get\text{-}maximum\text{-}possible\text{-}level\ M < backtrack\text{-}lvl\ S$
$\longrightarrow (\exists\, L.\ L \in\#\ D \wedge get\text{-}level\ (trail\ S)\ L = get\text{-}maximum\text{-}possible\text{-}level\ M)$

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 $\exists S'.\ cdcl_W\text{-}stgy\ S\ S'$
⟨*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 = backtrack-lvl S* **and**
D: *get-maximum-level* (*trail S*) (*remove1-mset L* (*mset-ccls E*)) < *backtrack-lvl S* **and**
bt: *backtrack-lvl S = get-maximum-level* (*trail S*) (*mset-ccls E*) **and**
M-L: $cdcl_W$*-M-level-inv S*
shows $\exists S'.\ cdcl_W\text{-}o\ S\ S'$
⟨*proof*⟩

lemma $cdcl_W$*-stgy-final-state-conclusive*:
assumes
termi: $\forall S'.\ \neg cdcl_W\text{-}stgy\ S\ S'$ **and**
decomp: *all-decomposition-implies-m* (*init-clss S*) (*get-all-ann-decomposition* (*trail S*)) **and**
learned: $cdcl_W$*-learned-clause S* **and**
level-inv: $cdcl_W$*-M-level-inv S* **and**
alien: *no-strange-atm S* **and**
no-dup: *distinct-*$cdcl_W$*-state S* **and**

confl: $cdcl_W$*-conflicting* S **and**
confl-k: *conflict-is-false-with-level* S
shows (*conflicting* S = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss* S)))
∨ (*conflicting* S = *None* ∧ *trail* S ⊨*as set-mset* (*init-clss* S))
⟨*proof*⟩

lemma $cdcl_W$*-cp-tranclp-*$cdcl_W$:
$cdcl_W$*-cp* S S' ⟹ $cdcl_W{}^{++}$ S S'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-cp-tranclp-*$cdcl_W$:
$cdcl_W$*-cp*$^{++}$ S S' ⟹ $cdcl_W{}^{++}$ S S'
⟨*proof*⟩

lemma $cdcl_W$*-stgy-tranclp-*$cdcl_W$:
$cdcl_W$*-stgy* S S' ⟹ $cdcl_W{}^{++}$ S S'
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-stgy-tranclp-*$cdcl_W$:
$cdcl_W$*-stgy*$^{++}$ S S' ⟹ $cdcl_W{}^{++}$ S S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-stgy-rtranclp-*$cdcl_W$:
$cdcl_W$*-stgy*** S S' ⟹ $cdcl_W{}^{**}$ S S'
⟨*proof*⟩

lemma *not-empty-get-maximum-level-exists-lit*:
assumes *n*: $D \neq$ {#}
and *max*: *get-maximum-level* M D = n
shows ∃ L ∈#D. *get-level* M L = n
⟨*proof*⟩

lemma $cdcl_W$*-o-conflict-is-false-with-level-inv*:
assumes
$cdcl_W$*-o* S S' **and**
lev: $cdcl_W$*-M-level-inv* S **and**
confl-inv: *conflict-is-false-with-level* S **and**
n-d: *distinct-*$cdcl_W$*-state* S **and**
conflicting: $cdcl_W$*-conflicting* S
shows *conflict-is-false-with-level* S'
⟨*proof*⟩

19.5.5 Strong completeness

lemma $cdcl_W$*-cp-propagate-confl*:
assumes $cdcl_W$*-cp* S T
shows *propagate*** S T ∨ (∃ S'. *propagate*** S S' ∧ *conflict* S' T)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-propagate-confl*:
assumes $cdcl_W$*-cp*** S T
shows *propagate*** S T ∨ (∃ S'. *propagate*** S S' ∧ *conflict* S' T)
⟨*proof*⟩

lemma *propagate-high-levelE*:
assumes *propagate* S T

obtains M' N' U k L C **where**
state $S = (M', N', U, k, None)$ **and**
state $T = (Propagated\ L\ (C + \{\#L\#\})\ \#\ M', N', U, k, None)$ **and**
$C + \{\#L\#\} \in\#$ *local.clauses* S **and**
$M' \models as$ *CNot* C **and**
undefined-lit (*trail* S) L
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-propagate-completeness*:
assumes *MN*: *set* $M \models s$ *set-mset* N **and**
cons: *consistent-interp* (*set* M) **and**
tot: *total-over-m* (*set* M) (*set-mset* N) **and**
lits-of-l (*trail* S) $\subseteq$ *set* M **and**
init-clss $S = N$ **and**
propagate** S S' **and**
learned-clss $S = \{\#\}$
shows *length* (*trail* S) $\leq$ *length* (*trail* S') $\wedge$ *lits-of-l* (*trail* S') $\subseteq$ *set* M
⟨*proof*⟩

lemma
assumes *propagate*** S X
shows
rtranclp-propagate-init-clss: *init-clss* X = *init-clss* S **and**
rtranclp-propagate-learned-clss: *learned-clss* X = *learned-clss* S
⟨*proof*⟩

lemma *completeness-is-a-full1-propagation*:
fixes S :: *'st* **and** M :: *'v literal list*
assumes *MN*: *set* $M \models s$ *set-mset* N
and *cons*: *consistent-interp* (*set* M)
and *tot*: *total-over-m* (*set* M) (*set-mset* N)
and *alien*: *no-strange-atm* S
and *learned*: *learned-clss* $S = \{\#\}$
and *clsS*[*simp*]: *init-clss* $S = N$
and *lits*: *lits-of-l* (*trail* S) $\subseteq$ *set* M
shows $\exists\, S'$. *propagate*** S S' $\wedge$ *full cdcl*$_W$*-cp* S S'
⟨*proof*⟩

See also *cdcl*$_W$*-cp*** *?S* *?S'* $\Longrightarrow \exists\, M$. *trail ?S'* $= M$ @ *trail ?S* $\wedge$ ($\forall\, l \in$*set* M. $\neg$ *is-decided* l)

lemma *rtranclp-propagate-is-trail-append*:
propagate** S $T \Longrightarrow \exists\, c$. *trail* $T = c$ @ *trail* S
⟨*proof*⟩

lemma *rtranclp-propagate-is-update-trail*:
propagate** S $T \Longrightarrow$ *cdcl*$_W$*-M-level-inv* $S \Longrightarrow$
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
$\exists M'\ k\ S.$ *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*⟩

theorem 2.9.11 page 84 of Weidenbach's book (with strategy)

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*) $\equiv$
($\forall M\ K\ i\ M'\ D.\ M'$ @ *Decided K i* # M = *trail S* $\longrightarrow D \in\#$ *clauses S*
$\longrightarrow \neg M \models$*as 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-confl S* **and**
no-f: *no-clause-is-false S*
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *conflict-no-smaller-confl-inv*:
assumes *conflict* $S\ S'$
and *no-smaller-confl S*
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *propagate-no-smaller-confl-inv*:

assumes *propagate*: *propagate* S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma $cdcl_W$*-cp-no-smaller-confl-inv*:
assumes *propagate*: $cdcl_W$*-cp* S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *rtrancp-*$cdcl_W$*-cp-no-smaller-confl-inv*:
assumes *propagate*: $cdcl_W$*-cp*** S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *trancp-*$cdcl_W$*-cp-no-smaller-confl-inv*:
assumes *propagate*: $cdcl_W$*-cp*$^{++}$ S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *full-*$cdcl_W$*-cp-no-smaller-confl-inv*:
assumes *full* $cdcl_W$*-cp* S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *full1-*$cdcl_W$*-cp-no-smaller-confl-inv*:
assumes *full1* $cdcl_W$*-cp* S S'
and *n-l*: *no-smaller-confl* S
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma $cdcl_W$*-stgy-no-smaller-confl-inv*:
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
shows *no-smaller-confl* S'
⟨*proof*⟩

lemma *is-conflicting-exists-conflict*:
assumes $\neg(\forall D \in \# init\text{-}clss\ S' + learned\text{-}clss\ S'.\ \neg\ trail\ S' \models as\ CNot\ D)$
and *conflicting* $S' = None$
shows $\exists S''.$ *conflict* S' S''
⟨*proof*⟩

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'
$\vee$ (*conflicting* $S' = None$
$\longrightarrow$ ($\forall D \in\#$ *clauses* S'. *trail* $S' \models as$ *CNot* D
$\longrightarrow$ ($\exists L$. $L \in\# D \wedge$ *get-level* (*trail* S') $L =$ *backtrack-lvl* S')))
⟨*proof*⟩

lemma *full1-cdcl*$_W$*-cp-exists-conflict-decompose*:
assumes
confl: $\exists D{\in}\#$*clauses* S. *trail* $S \models as$ *CNot* D **and**
full: *full cdcl*$_W$*-cp* S U **and**
no-confl: *conflicting* $S = None$ **and**
lev: *cdcl*$_W$*-M-level-inv* S
shows $\exists T$. *propagate*** S $T \wedge$ *conflict* T U
⟨*proof*⟩

lemma *full1-cdcl*$_W$*-cp-exists-conflict-full1-decompose*:
assumes
confl: $\exists D{\in}\#$*clauses* S. *trail* $S \models as$ *CNot* D **and**
full: *full cdcl*$_W$*-cp* S U **and**
no-confl: *conflicting* $S = None$**and**
lev: *cdcl*$_W$*-M-level-inv* S
shows $\exists T$ D. *propagate*** S $T \wedge$ *conflict* T U
$\wedge$ *trail* $T \models as$ *CNot* $D \wedge$ *conflicting* $U = Some$ $D \wedge D \in\#$ *clauses* S
⟨*proof*⟩

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'
⟨*proof*⟩

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'
⟨*proof*⟩

lemma *rtranclp-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-ann-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'*
⟨*proof*⟩

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*∈#*mset-clss N*. *D* $\neq$ {#}
shows (*conflicting S'* = *Some* {#} $\wedge$ *unsatisfiable* (*set-mset* (*init-clss S'*)))
$\vee$ (*conflicting S'* = *None* $\wedge$ *trail S'* $\models$*asm init-clss S'*)
⟨*proof*⟩

lemma *conflict-is-full1-cdcl$_W$-cp*:
assumes *cp*: *conflict S S'*
shows *full1 cdcl$_W$-cp S S'*
⟨*proof*⟩

lemma *cdcl$_W$-cp-fst-empty-conflicting-false*:
assumes
cdcl$_W$-cp S S' **and**
trail S = [] **and**
conflicting S $\neq$ *None*
shows *False*
⟨*proof*⟩

lemma *cdcl$_W$-o-fst-empty-conflicting-false*:
assumes *cdcl$_W$-o S S'*
and *trail S* = []
and *conflicting S* $\neq$ *None*
shows *False*
⟨*proof*⟩

lemma *cdcl$_W$-stgy-fst-empty-conflicting-false*:
assumes *cdcl$_W$-stgy S S'*
and *trail S* = []
and *conflicting S* $\neq$ *None*
shows *False*
⟨*proof*⟩
thm *cdcl$_W$-cp.induct*[*split-format*(*complete*)]

lemma *cdcl$_W$-cp-conflicting-is-false*:
cdcl$_W$-cp S S' $\Longrightarrow$ *conflicting S* = *Some* {#} $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-conflicting-is-false*:

$cdcl_W$*-cp*$^{++}$ *S S'* $\Longrightarrow$ *conflicting S* = *Some* {#} $\Longrightarrow$ *False*
⟨*proof*⟩

lemma $cdcl_W$*-o-conflicting-is-false*:
$cdcl_W$*-o S S'* $\Longrightarrow$ *conflicting S* = *Some* {#} $\Longrightarrow$ *False*
⟨*proof*⟩

lemma $cdcl_W$*-stgy-conflicting-is-false*:
$cdcl_W$*-stgy S S'* $\Longrightarrow$ *conflicting S* = *Some* {#} $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-stgy-conflicting-is-false*:
$cdcl_W$*-stgy*** *S S'* $\Longrightarrow$ *conflicting S* = *Some* {#} $\Longrightarrow$ *S'* = *S*
⟨*proof*⟩

lemma *full-*$cdcl_W$*-init-clss-with-false-normal-form*:
assumes
∀ *m*∈ *set M*. ¬*is-decided 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-ann-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*⟩

theorem 2.9.9 page 83 of Weidenbach's book

lemma *full-*$cdcl_W$*-stgy-final-state-conclusive*:
fixes *S'* :: *'st*
assumes *full*: *full* $cdcl_W$*-stgy* (*init-state N*) *S'* **and** *no-d*: *distinct-mset-mset* (*mset-clss N*)
shows (*conflicting S'* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*init-clss S'*)))
∨ (*conflicting S'* = *None* ∧ *trail S'* ⊨*asm init-clss S'*)
⟨*proof*⟩

theorem 2.9.9 page 83 of Weidenbach's book

lemma *full-*$cdcl_W$*-stgy-final-state-conclusive-from-init-state*:
fixes *S'* :: *'st*
assumes *full*: *full* $cdcl_W$*-stgy* (*init-state N*) *S'*
and *no-d*: *distinct-mset-mset* (*mset-clss N*)
shows (*conflicting S'* = *Some* {#} ∧ *unsatisfiable* (*set-mset* (*mset-clss N*)))
∨ (*conflicting S'* = *None* ∧ *trail S'* ⊨*asm* (*mset-clss N*) ∧ *satisfiable* (*set-mset* (*mset-clss N*)))
⟨*proof*⟩
end

end
theory *CDCL-W-Termination*
imports *CDCL-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 $cdcl_W$*-all-struct-inv* **where**
$cdcl_W$*-all-struct-inv* $S \longleftrightarrow$
no-strange-atm S $\wedge$
$cdcl_W$*-M-level-inv* S $\wedge$
$(\forall s \in\# \textit{learned-clss } S.\ \neg \textit{tautology } s)$ $\wedge$
distinct-$cdcl_W$*-state* S $\wedge$
$cdcl_W$*-conflicting* S $\wedge$
all-decomposition-implies-m (*init-clss* S) (*get-all-ann-decomposition* (*trail* S)) $\wedge$
$cdcl_W$*-learned-clause* S

lemma $cdcl_W$*-all-struct-inv-inv*:
assumes $cdcl_W$ S S' **and** $cdcl_W$*-all-struct-inv* S
shows $cdcl_W$*-all-struct-inv* S'
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-all-struct-inv-inv*:
assumes $cdcl_W^{**}$ S S' **and** $cdcl_W$*-all-struct-inv* S
shows $cdcl_W$*-all-struct-inv* S'
⟨*proof*⟩

lemma $cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*:
$cdcl_W$*-stgy* S T $\Longrightarrow$ $cdcl_W$*-all-struct-inv* S $\Longrightarrow$ $cdcl_W$*-all-struct-inv* T
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-stgy-*$cdcl_W$*-all-struct-inv*:
$cdcl_W$*-stgy*** S T $\Longrightarrow$ $cdcl_W$*-all-struct-inv* S $\Longrightarrow$ $cdcl_W$*-all-struct-inv* T
⟨*proof*⟩

19.7 No Relearning of a clause

lemma $cdcl_W$*-o-new-clause-learned-is-backtrack-step*:
assumes *learned*: $D \in\# \textit{learned-clss } T$ **and**
new: $D \notin\# \textit{learned-clss } S$ **and**
$cdcl_W$: $cdcl_W$*-o* S T **and**
lev: $cdcl_W$*-M-level-inv* S
shows *backtrack* S T $\wedge$ *conflicting* S = *Some* D
⟨*proof*⟩

lemma $cdcl_W$*-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: $D \in\# \textit{learned-clss } T$ **and**
new: $D \notin\# \textit{learned-clss } S$ **and**
$cdcl_W$: $cdcl_W$*-stgy* S T **and**

lev: $cdcl_W$*-M-level-inv S*
shows $\exists S'$. *backtrack S S'* $\wedge$ $cdcl_W$*-stgy*** *S' T* $\wedge$ *conflicting S = Some D*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-new-clause-learned-has-backtrack-step*:
assumes *learned*: $D \in\#$ *learned-clss T* **and**
new: $D \notin\#$ *learned-clss S* **and**
$cdcl_W$: $cdcl_W$*-stgy*** *S T* **and**
lev: $cdcl_W$*-M-level-inv S*
shows $\exists S' S''$. $cdcl_W$*-stgy*** *S S'* $\wedge$ *backtrack S' S''* $\wedge$ *conflicting S' = Some D* $\wedge$
$cdcl_W$*-stgy*** *S'' T*
⟨*proof*⟩

lemma *propagate-no-more-Decided-lit*:
assumes *propagate S S'*
shows *Decided K i* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K i* $\in$ *set* (*trail S'*)
⟨*proof*⟩

lemma *conflict-no-more-Decided-lit*:
assumes *conflict S S'*
shows *Decided K i* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K i* $\in$ *set* (*trail S'*)
⟨*proof*⟩

lemma $cdcl_W$*-cp-no-more-Decided-lit*:
assumes $cdcl_W$*-cp S S'*
shows *Decided K i* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K i* $\in$ *set* (*trail S'*)
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-no-more-Decided-lit*:
assumes $cdcl_W$*-cp*** *S S'*
shows *Decided K i* $\in$ *set* (*trail S*) $\longleftrightarrow$ *Decided K i* $\in$ *set* (*trail S'*)
⟨*proof*⟩

lemma $cdcl_W$*-o-no-more-Decided-lit*:
assumes $cdcl_W$*-o S S'* **and** *lev*: $cdcl_W$*-M-level-inv S* **and** $\neg$*decide S S'*
shows *Decided K i* $\in$ *set* (*trail S'*) $\longrightarrow$ *Decided K i* $\in$ *set* (*trail S*)
⟨*proof*⟩

lemma $cdcl_W$*-new-decided-at-beginning-is-decide*:
assumes $cdcl_W$*-stgy S S'* **and**
lev: $cdcl_W$*-M-level-inv S* **and**
trail S' = M' @ *Decided L i # M* **and**
trail S = M
shows $\exists T$. *decide S T* $\wedge$ *no-step* $cdcl_W$*-cp S*
⟨*proof*⟩

lemma $cdcl_W$*-o-is-decide*:
assumes $cdcl_W$*-o S T* **and** *lev*: $cdcl_W$*-M-level-inv S*
trail T = drop (*length* M_0) *M'* @ *Decided L i # H* @ *M***and**
$\neg$ ($\exists M'$. *trail S = M'* @ *Decided L i # H* @ *M*)
shows *decide S T*
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-new-decided-at-beginning-is-decide*:
assumes $cdcl_W$*-stgy*** *R U* **and**

$trail\ U = M' @ Decided\ L\ i\ \#\ H\ @\ M$ **and**
$trail\ R = M$ **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ R$
shows
$\exists S\ T\ T'.\ cdcl_W\text{-}stgy^{**}\ R\ S \wedge decide\ S\ T \wedge cdcl_W\text{-}stgy^{**}\ T\ U \wedge cdcl_W\text{-}stgy^{**}\ S\ U \wedge$
$no\text{-}step\ cdcl_W\text{-}cp\ S \wedge trail\ T = Decided\ L\ i\ \#\ H\ @\ M \wedge trail\ S = H\ @\ M \wedge cdcl_W\text{-}stgy\ S\ T' \wedge$
$cdcl_W\text{-}stgy^{**}\ T'\ U$
⟨proof⟩

lemma *rtranclp-cdcl$_W$-new-decided-at-beginning-is-decide'*:
assumes $cdcl_W\text{-}stgy^{**}\ R\ U$ **and**
$trail\ U = M' @ Decided\ L\ i\ \#\ H\ @\ M$ **and**
$trail\ R = M$ **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ R$
shows $\exists y\ y'.\ cdcl_W\text{-}stgy^{**}\ R\ y \wedge cdcl_W\text{-}stgy\ y\ y' \wedge \neg (\exists c.\ trail\ y = c\ @\ Decided\ L\ i\ \#\ H\ @\ M)$
$\wedge (\lambda a\ b.\ cdcl_W\text{-}stgy\ a\ b \wedge (\exists c.\ trail\ a = c\ @\ Decided\ L\ i\ \#\ H\ @\ M))^{**}\ y'\ U$
⟨proof⟩

lemma *beginning-not-decided-invert*:
assumes A: $M\ @\ A = M'\ @\ Decided\ K\ i\ \#\ H$ **and**
nm: $\forall m \in set\ M.\ \neg is\text{-}decided\ m$
shows $\exists M.\ A = M\ @\ Decided\ K\ i\ \#\ H$
⟨proof⟩

lemma *cdcl$_W$-stgy-trail-has-new-decided-is-decide-step*:
assumes $cdcl_W\text{-}stgy\ S\ T$
$\neg (\exists c.\ trail\ S = c\ @\ Decided\ L\ i\ \#\ H\ @\ M)$ **and**
$(\lambda a\ b.\ cdcl_W\text{-}stgy\ a\ b \wedge (\exists c.\ trail\ a = c\ @\ Decided\ L\ i\ \#\ H\ @\ M))^{**}\ T\ U$ **and**
$\exists M'.\ trail\ U = M'\ @\ Decided\ L\ i\ \#\ H\ @\ M$ **and**
lev: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $\exists S'.\ decide\ S\ S' \wedge full\ cdcl_W\text{-}cp\ S'\ T \wedge no\text{-}step\ cdcl_W\text{-}cp\ S$
⟨proof⟩

lemma *rtranclp-cdcl$_W$-stgy-with-trail-end-has-trail-end*:
assumes $(\lambda a\ b.\ cdcl_W\text{-}stgy\ a\ b \wedge (\exists c.\ trail\ a = c\ @\ Decided\ L\ i\ \#\ H\ @\ M))^{**}\ T\ U$ **and**
$\exists M'.\ trail\ U = M'\ @\ Decided\ L\ i\ \#\ H\ @\ M$
shows $\exists M'.\ trail\ T = M'\ @\ Decided\ L\ i\ \#\ H\ @\ M$
⟨proof⟩

lemma *remove1-mset-eq-remove1-mset-same*:
$remove1\text{-}mset\ L\ D = remove1\text{-}mset\ L'\ D \Longrightarrow L \in\#\ D \Longrightarrow L = L'$
⟨proof⟩

lemma *cdcl$_W$-o-cannot-learn*:
assumes
$cdcl_W\text{-}o\ y\ z$ **and**
lev: $cdcl_W\text{-}M\text{-}level\text{-}inv\ y$ **and**
trM: $trail\ y = c\ @\ Decided\ Kh\ i\ \#\ H$ **and**
DL: $D \notin\#\ learned\text{-}clss\ y$ **and**
LD: $L \in\#\ D$ **and**
DH: $atms\text{-}of\ (remove1\text{-}mset\ L\ D) \subseteq atm\text{-}of\ `lits\text{-}of\text{-}l\ H$ **and**
LH: $atm\text{-}of\ L \notin atm\text{-}of\ `lits\text{-}of\text{-}l\ H$ **and**
$learned$: $\forall T.\ conflicting\ y = Some\ T \longrightarrow trail\ y \models as\ CNot\ T$ **and**
z: $trail\ z = c'\ @\ Decided\ Kh\ i\ \#\ H$
shows $D \notin\#\ learned\text{-}clss\ z$

⟨*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 $y = c$ @ *Decided Kh* i # H **and**
$D \notin\#$ *learned-clss* y **and**
LD: $L \in\# D$ **and**
DH: *atms-of* (*remove1-mset* L D) $\subseteq$ *atm-of* ' *lits-of-l* H **and**
LH: *atm-of* $L \notin$ *atm-of* ' *lits-of-l* H **and**
$\forall T$. *conflicting* y = *Some* $T \longrightarrow$ *trail* $y \models as$ *CNot* T **and**
trail $z = c'$ @ *Decided Kh* i # H
shows $D \notin\#$ *learned-clss* z
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-stgy-with-trail-end-has-not-been-learned*:
assumes
(λa b. $cdcl_W$*-stgy* a b $\wedge$ ($\exists c$. *trail* $a = c$ @ *Decided* K i # H @ []))** S z **and**
$cdcl_W$*-all-struct-inv* S **and**
trail $S = c$ @ *Decided* K i # H **and**
$D \notin\#$ *learned-clss* S **and**
LD: $L \in\# D$ **and**
DH: *atms-of* (*remove1-mset* L D) $\subseteq$ *atm-of* ' *lits-of-l* H **and**
LH: *atm-of* $L \notin$ *atm-of* ' *lits-of-l* H **and**
$\exists c'$. *trail* $z = c'$ @ *Decided* K i # H
shows $D \notin\#$ *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**
$E \notin\#$ *learned-clss* S **and**
$E \in\#$ *learned-clss* T
shows $\exists S'$. *backtrack* S S' $\wedge$ *conflicting* S = *Some* E $\wedge$ *full* $cdcl_W$*-cp* S' T
⟨*proof*⟩

theorem 2.9.7 page 83 of Weidenbach's book

lemma *cdcl*$_W$*-stgy-no-relearned-clause*:
assumes
invR: $cdcl_W$*-all-struct-inv* R **and**
st': $cdcl_W$*-stgy*** R S **and**
bt: *backtrack* S T **and**
confl: *raw-conflicting* S = *Some* E **and**
already-learned: *mset-ccls* $E \in\#$ *clauses* S **and**
R: *trail* R = []
shows *False*
⟨*proof*⟩

lemma *rtranclp-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*::*nat*) ^ (*card* (*atms-of-mm* (*init-clss* S))) − *card* (*set-mset* (*learned-clss* S)),
if conflicting S = *None then 1 else 0*,
if conflicting S = *None then card* (*atms-of-mm* (*init-clss* S)) − *length* (*trail* S)
else length (*trail* S)
]

lemma *length-model-le-vars-all-inv*:
assumes $cdcl_W$*-all-struct-inv* S
shows *length* (*trail* S) ≤ *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**
∀ s ∈# *learned-clss* S. ¬*tautology* s
shows *card*(*set-mset* (*learned-clss* S)) ≤ *3* ^ *card* (*atms-of-mm* (*learned-clss* S))
⟨*proof*⟩

lemma $cdcl_W$*-measure-decreasing*:
fixes S :: *'st*
assumes
$cdcl_W$ S S' **and**
no-restart:
¬(*learned-clss* S ⊆# *learned-clss* S' ∧ [] = *trail* S' ∧ *conflicting* S' = *None*)
and
no-forget: *learned-clss* S ⊆# *learned-clss* S' **and**
no-relearn: $\bigwedge S'$. *backtrack* S S' ⟹ ∀ T. *conflicting* S = *Some* T ⟶ T ∉# *learned-clss* S
and
alien: *no-strange-atm* S **and**
M-level: $cdcl_W$*-M-level-inv* S **and**
no-taut: ∀ s ∈# *learned-clss* S. ¬*tautology* s **and**
no-dup: *distinct-*$cdcl_W$*-state* S **and**
confl: $cdcl_W$*-conflicting* S
shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) ∈ *lexn less-than 3*
⟨*proof*⟩

lemma *propagate-measure-decreasing*:
fixes S :: *'st*
assumes *propagate* S S' **and** $cdcl_W$*-all-struct-inv* S
shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *conflict-measure-decreasing*:
fixes S :: *'st*
assumes *conflict* S S' **and** $cdcl_W$*-all-struct-inv* S
shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *decide-measure-decreasing*:
fixes S :: *'st*
assumes *decide* S S' **and** $cdcl_W$*-all-struct-inv* S
shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma $cdcl_W$*-cp-measure-decreasing*:
fixes S :: *'st*
assumes $cdcl_W$*-cp* S S' **and** $cdcl_W$*-all-struct-inv* S
shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-cp-measure-decreasing*:
fixes S :: *'st*
assumes $cdcl_W$*-cp*$^{++}$ S S' **and** $cdcl_W$*-all-struct-inv* S
shows ($cdcl_W$*-measure* S', $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma $cdcl_W$*-stgy-step-decreasing*:
fixes R S T :: *'st*
assumes $cdcl_W$*-stgy* S T **and**
$cdcl_W$*-stgy*** R S
trail R = [] **and**
$cdcl_W$*-all-struct-inv* R
shows ($cdcl_W$*-measure* T, $cdcl_W$*-measure* S) $\in$ *lexn less-than 3*
⟨*proof*⟩

Roughly corresponds to theorem 2.9.15 page 86 of Weidenbach's book (using a different bound)

lemma *tranclp-*$cdcl_W$*-stgy-decreasing*:
fixes R S T :: *'st*
assumes $cdcl_W$*-stgy*$^{++}$ R S
trail R = [] **and**
$cdcl_W$*-all-struct-inv* R
shows ($cdcl_W$*-measure* S, $cdcl_W$*-measure* R) $\in$ *lexn less-than 3*
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-stgy-S0-decreasing*:
fixes R S T :: *'st*
assumes
pl: $cdcl_W$*-stgy*$^{++}$ (*init-state* N) S **and**
no-dup: *distinct-mset-mset* (*mset-clss* N)
shows ($cdcl_W$*-measure* S, $cdcl_W$*-measure* (*init-state* N)) $\in$ *lexn less-than 3*

⟨*proof*⟩

lemma *wf-tranclp-cdcl$_W$-stgy*:
  wf $\{(S::'st,\ init\text{-}state\ N)|$
    $S\ N.\ distinct\text{-}mset\text{-}mset\ (mset\text{-}clss\ N) \wedge cdcl_W\text{-}stgy^{++}\ (init\text{-}state\ N)\ S\}$
  ⟨*proof*⟩

lemma *cdcl$_W$-cp-wf-all-inv*:
  $wf\ \{(S',\ S).\ cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \wedge cdcl_W\text{-}cp\ S\ S'\}$
  (**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*]:
  $(\forall c \in set\ C.\ -c \in lits\text{-}of\text{-}l\ Ms) \longleftrightarrow Ms \models as\ CNot\ (mset\ C)$
  ⟨*proof*⟩

The right-hand version is written at a high-level, but only the left-hand side is executable.

definition *is-unit-clause* :: $'a\ literal\ list \Rightarrow ('a,\ 'b,\ 'c)\ ann\text{-}lit\ list \Rightarrow 'a\ literal\ option$
 where
 $is\text{-}unit\text{-}clause\ l\ M =$
  $(case\ List.filter\ (\lambda a.\ atm\text{-}of\ a \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M)\ l\ of$
    $a\ \#\ [] \Rightarrow if\ M \models as\ CNot\ (mset\ l - \{\#a\#\})\ then\ Some\ a\ else\ None$
  $|\ \text{-} \Rightarrow None)$

definition *is-unit-clause-code* :: $'a\ literal\ list \Rightarrow ('a,\ 'b,\ 'c)\ ann\text{-}lit\ list$
  $\Rightarrow 'a\ literal\ option$ **where**
 $is\text{-}unit\text{-}clause\text{-}code\ l\ M =$
  $(case\ List.filter\ (\lambda a.\ atm\text{-}of\ a \notin atm\text{-}of\ `\ lits\text{-}of\text{-}l\ M)\ l\ of$
    $a\ \#\ [] \Rightarrow if\ (\forall c \in set\ (remove1\ a\ l).\ -c \in lits\text{-}of\text{-}l\ M)\ then\ Some\ a\ else\ None$
  $|\ \text{-} \Rightarrow None)$

lemma *is-unit-clause-is-unit-clause-code*[*code*]:
  $is\text{-}unit\text{-}clause\ l\ M = is\text{-}unit\text{-}clause\text{-}code\ l\ M$
⟨*proof*⟩

lemma *is-unit-clause-some-undef*:
  assumes $is\text{-}unit\text{-}clause\ l\ M = Some\ a$
  shows $undefined\text{-}lit\ M\ a$
⟨*proof*⟩

lemma *is-unit-clause-some-CNot*: $is\text{-}unit\text{-}clause\ l\ M = Some\ a \Longrightarrow M \models as\ CNot\ (mset\ l - \{\#a\#\})$

$\langle proof \rangle$

lemma *is-unit-clause-some-in*: *is-unit-clause l M = Some a* $\Longrightarrow$ *a* $\in$ *set l*
$\langle proof \rangle$

lemma *is-unit-clause-nil*[*simp*]: *is-unit-clause* [] *M = None*
$\langle proof \rangle$

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*) *ann-lit list*
$\Rightarrow$ (*'a literal* $\times$ *'a literal list*) *option* **where**
find-first-unit-clause (*a* # *l*) *M* =
(*case is-unit-clause a M of*
None $\Rightarrow$ *find-first-unit-clause l M*
| *Some L* $\Rightarrow$ *Some* (*L*, *a*)) |
find-first-unit-clause [] - = *None*

lemma *find-first-unit-clause-some*:
find-first-unit-clause l M = Some (*a*, *c*)
$\Longrightarrow$ *c* $\in$ *set l* $\wedge$ *M* $\models$*as CNot* (*mset c* $-$ {#*a*#}) $\wedge$ *undefined-lit M a* $\wedge$ *a* $\in$ *set c*
$\langle proof \rangle$

lemma *propagate-is-unit-clause-not-None*:
assumes *dist*: *distinct c* **and**
M: *M* $\models$*as CNot* (*mset c* $-$ {#*a*#}) **and**
undef: *undefined-lit M a* **and**
ac: *a* $\in$ *set c*
shows *is-unit-clause c M* $\neq$ *None*
$\langle proof \rangle$

lemma *find-first-unit-clause-none*:
distinct c $\Longrightarrow$ *c* $\in$ *set l* $\Longrightarrow$ *M* $\models$*as CNot* (*mset c* $-$ {#*a*#}) $\Longrightarrow$ *undefined-lit M a* $\Longrightarrow$ *a* $\in$ *set c*
$\Longrightarrow$ *find-first-unit-clause l M* $\neq$ *None*
$\langle proof \rangle$

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* (λ*lit. lit* $\notin$ *M* $\wedge$ $-$*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 (λ*lit. lit* $\notin$ *M* $\wedge$ $-$*lit* $\notin$ *M*) *a = None* $\longleftrightarrow$ *atm-of ‘ set a* $\subseteq$ *atm-of ‘ M*
$\langle proof \rangle$

lemma *find-some*: *List.find* (λ*lit. lit* $\notin$ *M* $\wedge$ $-$*lit* $\notin$ *M*) *a = Some b* $\Longrightarrow$ *b* $\in$ *set a* $\wedge$ *b* $\notin$ *M* $\wedge$ $-$*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*)

⟨*proof*⟩

lemma *find-first-unused-var-Some-not-all-incl*:
assumes *find-first-unused-var l M = Some c*
shows $\neg(\forall a \in \textit{set } l.\ \textit{atm-of} ` \textit{set } a \subseteq \textit{atm-of} ` M)$
⟨*proof*⟩

lemma *find-first-unused-var-Some*:
$\textit{find-first-unused-var } l\ M = \textit{Some } a \Longrightarrow (\exists m \in \textit{set } l.\ a \in \textit{set } m \wedge a \notin M \wedge -a \notin M)$
⟨*proof*⟩

lemma *find-first-unused-var-undefined*:
$\textit{find-first-unused-var } l\ (\textit{lits-of-l } Ms) = \textit{Some } a \Longrightarrow \textit{undefined-lit } Ms\ a$
⟨*proof*⟩

end
theory *DPLL-W-Implementation*
imports *DPLL-CDCL-W-Implementation DPLL-W ~~/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* $\textit{dpll}_W$*-ann-lits* × *int literal list list*
⇒ *int* $\textit{dpll}_W$*-ann-lits* × *int literal list list* **where**
DPLL-step = (λ(*Ms*, *N*).
(*case find-first-unit-clause N Ms of*
Some (*L*, -) ⇒ (*Propagated L* () # *Ms*, *N*)
| - ⇒
if ∃ *C* ∈ *set N*. (∀ *c* ∈ *set C*. −*c* ∈ *lits-of-l Ms*)
then
(*case backtrack-split Ms of*
(-, *L* # *M*) ⇒ (*Propagated* (− (*lit-of L*)) () # *M*, *N*)
| (-, -) ⇒ (*Ms*, *N*)
)
else
(*case find-first-unused-var N* (*lits-of-l Ms*) *of*
Some a ⇒ (*Decided a* () # *Ms*, *N*)
| *None* ⇒ (*Ms*, *N*))))

Example of propagation:

value *DPLL-step* ([*Decided* (*Neg 1*) ()], [[*Pos* (*1*::*int*), *Neg 2*]])

We define the conversion function between the states as defined in *Prop-DPLL* (with multisets) and here (with lists).

abbreviation *toS* ≡ λ(*Ms*::(*int*, *unit*, *unit*) *ann-lit list*)
(*N*:: *int literal list list*). (*Ms*, *mset* (*map mset N*))
abbreviation *toS'* ≡ λ(*Ms*::(*int*, *unit*, *unit*) *ann-lit list*,
N:: *int literal list list*). (*Ms*, *mset* (*map mset N*))

Proof of correctness of *DPLL-step*

lemma *DPLL-step-is-a-*$\textit{dpll}_W$*-step*:
assumes *step*: (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*)
and *neq*: (*Ms*, *N*) ≠ (*Ms'*, *N'*)

shows $dpll_W$ (*toS Ms N*) (*toS Ms' N'*)
⟨*proof*⟩

lemma *DPLL-step-stuck-final-state*:
assumes *step*: (*Ms*, *N*) = *DPLL-step* (*Ms*, *N*)
shows *conclusive-*$dpll_W$*-state* (*toS Ms N*)
⟨*proof*⟩

20.2.2 Adding invariants

Invariant tested in the function **function** *DPLL-ci* :: *int* $dpll_W$*-ann-lits* ⇒ *int literal list list* ⇒ *int* $dpll_W$*-ann-lits* × *int literal list list* **where**
DPLL-ci Ms N =
(*if* ¬$dpll_W$*-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*)
⟨*proof*⟩
termination
⟨*proof*⟩

No invariant tested **function** (*domintros*) *DPLL-part*:: *int* $dpll_W$*-ann-lits* ⇒ *int literal list list* ⇒ *int* $dpll_W$*-ann-lits* × *int literal list list* **where**
DPLL-part Ms N =
(*let* (*Ms'*, *N'*) = *DPLL-step* (*Ms*, *N*) *in*
if (*Ms'*, *N'*) = (*Ms*, *N*) *then* (*Ms*, *N*) *else DPLL-part Ms' N*)
⟨*proof*⟩

lemma *snd-DPLL-step*[*simp*]:
snd (*DPLL-step* (*Ms*, *N*)) = *N*
⟨*proof*⟩

lemma $dpll_W$*-all-inv-implieS-2-eq3-and-dom*:
assumes $dpll_W$*-all-inv* (*Ms*, *mset* (*map mset N*))
shows *DPLL-ci Ms N* = *DPLL-part Ms N* ∧ *DPLL-part-dom* (*Ms*, *N*)
⟨*proof*⟩

lemma *DPLL-ci-*$dpll_W$*-rtranclp*:
assumes *DPLL-ci Ms N* = (*Ms'*, *N'*)
shows ${dpll_W}^{**}$ (*toS Ms N*) (*toS Ms' N*)
⟨*proof*⟩

lemma $dpll_W$*-all-inv-*$dpll_W$*-tranclp-irrefl*:
assumes $dpll_W$*-all-inv* (*Ms*, *N*)
and ${dpll_W}^{++}$ (*Ms*, *N*) (*Ms*, *N*)
shows *False*
⟨*proof*⟩

lemma *DPLL-ci-final-state*:
assumes *step*: *DPLL-ci Ms N* = (*Ms*, *N*)
and *inv*: $dpll_W$*-all-inv* (*toS Ms N*)
shows *conclusive-*$dpll_W$*-state* (*toS Ms N*)
⟨*proof*⟩

lemma *DPLL-step-obtains*:

obtains Ms' **where** $(Ms', N) = DPLL\text{-}step\ (Ms, N)$
⟨*proof*⟩

lemma *DPLL-ci-obtains*:
obtains Ms' **where** $(Ms', N) = DPLL\text{-}ci\ Ms\ N$
⟨*proof*⟩

lemma *DPLL-ci-no-more-step*:
assumes *step*: $DPLL\text{-}ci\ Ms\ N = (Ms', N')$
shows $DPLL\text{-}ci\ Ms'\ N' = (Ms', N')$
⟨*proof*⟩

lemma *DPLL-part-dpll$_W$-all-inv-final*:
fixes $M\ Ms'$:: (*int*, *unit*, *unit*) *ann-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 $conclusive\text{-}dpll_W\text{-}state\ (toS\ Ms'\ N) \land dpll_W{}^{**}\ (toS\ Ms\ N)\ (toS\ Ms'\ N)$
⟨*proof*⟩

Embedding the invariant into the type

Defining the type **typedef** $dpll_W\text{-}state$ =
$\{(M::(int, unit, unit)\ ann\text{-}lit\ list, N::int\ literal\ list\ list).$
$dpll_W\text{-}all\text{-}inv\ (toS\ M\ N)\}$
morphisms *rough-state-of state-of*
⟨*proof*⟩

lemma
$DPLL\text{-}part\text{-}dom\ ([], N)$
⟨*proof*⟩

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
⟨*proof*⟩
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-state-of-inverse*[*simp*]

lemma *DPLL-step-dpll$_W$-conc-inv*:
$DPLL\text{-}step\ (rough\text{-}state\text{-}of\ S) \in \{(M, N).\ dpll_W\text{-}all\text{-}inv\ (toS\ M\ N)\}$
⟨*proof*⟩

lemma *rough-state-of-DPLL-step'-DPLL-step*[*simp*]:
$rough\text{-}state\text{-}of\ (DPLL\text{-}step'\ S) = DPLL\text{-}step\ (rough\text{-}state\text{-}of\ S)$
⟨*proof*⟩

function *DPLL-tot*:: $dpll_W$*-state* ⇒ $dpll_W$*-state* **where**
DPLL-tot S =
  (*let S′* = *DPLL-step′ S in*
   if S′ = *S then S else DPLL-tot S′*)
  ⟨*proof*⟩
termination
⟨*proof*⟩

lemma [*code*]:
DPLL-tot S =
  (*let S′* = *DPLL-step′ S in*
   if S′ = *S then S else DPLL-tot S′*) ⟨*proof*⟩

lemma *DPLL-tot-DPLL-step-DPLL-tot*[*simp*]: *DPLL-tot* (*DPLL-step′ S*) = *DPLL-tot S*
  ⟨*proof*⟩

lemma *DOPLL-step′-DPLL-tot*[*simp*]:
  DPLL-step′ (*DPLL-tot S*) = *DPLL-tot S*
  ⟨*proof*⟩

lemma *DPLL-tot-final-state*:
  assumes *DPLL-tot S* = *S*
  shows *conclusive-*$dpll_W$*-state* (*toS′* (*rough-state-of S*))
⟨*proof*⟩

lemma *DPLL-tot-star*:
  assumes *rough-state-of* (*DPLL-tot S*) = *S′*
  shows $dpll_W{}^{**}$ (*toS′* (*rough-state-of S*)) (*toS′ S′*)
  ⟨*proof*⟩

lemma *rough-state-of-rough-state-of-nil*[*simp*]:
  rough-state-of (*state-of* ([], *N*)) = ([], *N*)
  ⟨*proof*⟩

Theorem of correctness

lemma *DPLL-tot-correct*:
  assumes *rough-state-of* (*DPLL-tot* (*state-of* (([], *N*)))) = (*M*, *N′*)
  and (*M′*, *N″*) = *toS′* (*M*, *N′*)
  shows *M′* ⊨*asm N″* ⟷ *satisfiable* (*set-mset N″*)
⟨*proof*⟩

20.2.3 Code export

A conversion to *DPLL-W-Implementation.*$dpll_W$*-state*   **definition** *Con* :: (*int*, *unit*, *unit*) *ann-lit list* × *int literal list list*
                    ⇒ $dpll_W$*-state* **where**
  Con xs = *state-of* (*if* $dpll_W$*-all-inv* (*toS* (*fst xs*) (*snd xs*)) *then xs else* ([], []))
lemma [*code abstype*]:
  Con (*rough-state-of S*) = *S*
  ⟨*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* (∀ *A* ∈ *set N*. (∃ *a*∈*set A*. *a* ∈ *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 $cdcl_W$*-decided-level* = *nat*

type-synonym *'v* $cdcl_W$*-ann-lit* = (*'v*, $cdcl_W$*-decided-level*, *'v* $cdcl_W$*-mark*) *ann-lit*
type-synonym *'v* $cdcl_W$*-ann-lits* = (*'v*, $cdcl_W$*-decided-level*, *'v* $cdcl_W$*-mark*) *ann-lits*
type-synonym *'v* $cdcl_W$*-state* =
'v $cdcl_W$*-ann-lits* × *'v literal list list* × *'v literal list list* × *nat* ×
'v literal list option

abbreviation *raw-trail* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'a* **where**
raw-trail ≡ (λ(*M*, -). *M*)

abbreviation *raw-cons-trail* :: *'a* ⇒ *'a list* × *'b* × *'c* × *'d* × *'e* ⇒ *'a list* × *'b* × *'c* × *'d* × *'e*
where
raw-cons-trail ≡ (λ*L* (*M*, *S*). (*L*#*M*, *S*))

abbreviation *raw-tl-trail* :: *'a list* × *'b* × *'c* × *'d* × *'e* ⇒ *'a list* × *'b* × *'c* × *'d* × *'e* **where**
raw-tl-trail ≡ (λ(*M*, *S*). (*tl M*, *S*))

abbreviation *raw-init-clss* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'b* **where**
raw-init-clss ≡ λ(*M*, *N*, -). *N*

abbreviation *raw-learned-clss* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'c* **where**
raw-learned-clss ≡ λ(*M*, *N*, *U*, -). *U*

abbreviation *raw-backtrack-lvl* :: *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'d* **where**
raw-backtrack-lvl ≡ λ(*M*, *N*, *U*, *k*, -). *k*

abbreviation *raw-update-backtrack-lvl* :: *'d* ⇒ *'a* × *'b* × *'c* × *'d* × *'e* ⇒ *'a* × *'b* × *'c* × *'d* × *'e*
where

raw-update-backtrack-lvl $\equiv \lambda k\ (M,\ N,\ U,\ \text{-},\ S).\ \ (M,\ N,\ U,\ k,\ S)$

abbreviation *raw-conflicting* :: $'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'e$ **where**
raw-conflicting $\equiv \lambda(M,\ N,\ U,\ k,\ D).\ D$

abbreviation *raw-update-conflicting* :: $'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a \times 'b \times 'c \times 'd \times 'e$
where
raw-update-conflicting $\equiv \lambda S\ (M,\ N,\ U,\ k,\ \text{-}).\ \ (M,\ N,\ U,\ k,\ S)$

abbreviation *raw-add-learned-cls* **where**
raw-add-learned-cls $\equiv \lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ \{\#C\#\}\ +\ U,\ S)$

abbreviation *raw-remove-cls* **where**
raw-remove-cls $\equiv \lambda C\ (M,\ N,\ U,\ S).\ (M,$ *removeAll-mset C N*, *removeAll-mset C U*, $S)$

type-synonym *'v cdcl$_W$-state-inv-st* = (*'v, nat, 'v literal list*) *ann-lit list* $\times$
'v literal list list $\times$ *'v literal list list* $\times$ *nat* $\times$ *'v literal list option*

abbreviation *raw-S0-cdcl$_W$ N* $\equiv$ (([], *N*, [], *0*, *None*):: *'v cdcl$_W$-state-inv-st*)

fun *mmset-of-mlit'* :: (*'v, nat, 'v literal list*) *ann-lit* $\Rightarrow$ (*'v, nat, 'v clause*) *ann-lit*
where
mmset-of-mlit' (*Propagated L C*) = *Propagated L* (*mset C*) |
mmset-of-mlit' (*Decided L i*) = *Decided L i*

lemma *lit-of-mmset-of-mlit'*[*simp*]:
lit-of (*mmset-of-mlit' xa*) = *lit-of xa*
⟨*proof*⟩

abbreviation *trail* **where**
trail S $\equiv$ *map mmset-of-mlit'* (*raw-trail S*)

abbreviation *clauses-of-l* **where**
clauses-of-l $\equiv \lambda L.$ *mset* (*map mset L*)

global-interpretation *state$_W$-ops*
mset::*'v literal list* $\Rightarrow$ *'v clause*
op # remove1

clauses-of-l op @ $\lambda L\ C.\ L \in$ *set C op #* $\lambda C.$ *remove1-cond* ($\lambda L.$ *mset L* = *mset C*)

mset $\lambda xs\ ys.$ *case-prod append* (*fold* (λx (*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*
$\lambda(\text{-},\ N,\ \text{-}).\ N$
$\lambda(\text{-},\ \text{-},\ U,\ \text{-}).\ U$
$\lambda(\text{-},\ \text{-},\ \text{-},\ k,\ \text{-}).\ k$
$\lambda(\text{-},\ \text{-},\ \text{-},\ \text{-},\ C).\ C$

$\lambda L\ (M,\ S).\ (L\ \#\ M,\ S)$
$\lambda(M,\ S).\ (tl\ M,\ S)$
$\lambda C\ (M,\ N,\ S).\ (M,\ C\ \#\ N,\ S)$

$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ C\ \#\ U,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ \mathit{filter}\ (\lambda L.\ \mathit{mset}\ L \neq \mathit{mset}\ C)\ N,\ \mathit{filter}\ (\lambda L.\ \mathit{mset}\ L \neq \mathit{mset}\ C)\ U,\ S)$
$\lambda(k{::}\mathit{nat})\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$
$\lambda D\ (M,\ N,\ U,\ k,\ \text{-}).\ (M,\ N,\ U,\ k,\ D)$
$\lambda N.\ ([],\ N,\ [],\ \mathit{0},\ \mathit{None})$
$\lambda(\text{-},\ N,\ U,\ \text{-}).\ ([],\ N,\ U,\ \mathit{0},\ \mathit{None})$
⟨*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 $[L{\leftarrow}a\ .\ \mathit{mset}\ L \neq \mathit{mset}\ C] = \mathit{mset}\ (\mathit{removeAll}\ (\mathit{mset}\ C)\ (\mathit{map}\ \mathit{mset}\ a))$
⟨*proof*⟩

interpretation state_W
mset::′*v literal list* ⇒ ′*v clause*
op # remove1

clauses-of-l op @ $\lambda L\ C.\ L \in \mathit{set}\ C$ *op #* $\lambda C.$ *remove1-cond* $(\lambda L.\ \mathit{mset}\ L = \mathit{mset}\ C)$

mset $\lambda xs\ ys.$ *case-prod append* $(\mathit{fold}\ (\lambda x\ (ys,\ zs).\ (\mathit{remove1}\ x\ ys,\ x\ \#\ zs))\ xs\ (ys,\ []))$
op # remove1

id id

$\lambda(M,\ \text{-}).$ *map mmset-of-mlit′ M* $\lambda(M,\ \text{-}).$ *hd M*
$\lambda(\text{-},\ N,\ \text{-}).\ N$
$\lambda(\text{-},\ \text{-},\ U,\ \text{-}).\ U$
$\lambda(\text{-},\ \text{-},\ \text{-},\ k,\ \text{-}).\ k$
$\lambda(\text{-},\ \text{-},\ \text{-},\ \text{-},\ C).\ C$

$\lambda L\ (M,\ S).\ (L\ \#\ M,\ S)$
$\lambda(M,\ S).\ (\mathit{tl}\ M,\ S)$
$\lambda C\ (M,\ N,\ S).\ (M,\ C\ \#\ N,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ N,\ C\ \#\ U,\ S)$
$\lambda C\ (M,\ N,\ U,\ S).\ (M,\ \mathit{filter}\ (\lambda L.\ \mathit{mset}\ L \neq \mathit{mset}\ C)\ N,\ \mathit{filter}\ (\lambda L.\ \mathit{mset}\ L \neq \mathit{mset}\ C)\ U,\ S)$
$\lambda(k{::}\mathit{nat})\ (M,\ N,\ U,\ \text{-},\ D).\ (M,\ N,\ U,\ k,\ D)$
$\lambda D\ (M,\ N,\ U,\ k,\ \text{-}).\ (M,\ N,\ U,\ k,\ D)$
$\lambda N.\ ([],\ N,\ [],\ \mathit{0},\ \mathit{None})$
$\lambda(\text{-},\ N,\ U,\ \text{-}).\ ([],\ N,\ U,\ \mathit{0},\ \mathit{None})$
⟨*proof*⟩

global-interpretation $\mathit{conflict\text{-}driven\text{-}clause\text{-}learning}_W$
mset::′*v literal list* ⇒ ′*v clause*
op # remove1

clauses-of-l op @ $\lambda L\ C.\ L \in \mathit{set}\ C$ *op #* $\lambda C.$ *remove1-cond* $(\lambda L.\ \mathit{mset}\ L = \mathit{mset}\ C)$

mset $\lambda xs\ ys.$ *case-prod append* $(\mathit{fold}\ (\lambda x\ (ys,\ zs).\ (\mathit{remove1}\ x\ ys,\ x\ \#\ zs))\ xs\ (ys,\ []))$
op # remove1

id id

$\lambda(M,\ \text{-}).$ *map mmset-of-mlit′ M* $\lambda(M,\ \text{-}).$ *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* $L \neq$ *mset* C) N, *filter* (λL. *mset* $L \neq$ *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*⟩

declare *state-simp*[*simp del*] *raw-clauses-def*[*simp*] *state-eq-def*[*simp*]
notation *state-eq* (**infix** $\sim$ *50*)
term *reduce-trail-to*

lemma *reduce-trail-to-map*[*simp*]:
reduce-trail-to (*map f M1*) = *reduce-trail-to M1*
⟨*proof*⟩

20.3 CDCL Implementation

20.3.1 Types and Additional Lemmas

lemma *true-clss-remdups*[*simp*]:
$I \models s$ (*mset* $\circ$ *remdups*) ‘ $N \longleftrightarrow I \models s$ *mset* ‘ N
⟨*proof*⟩

lemma *satisfiable-mset-remdups*[*simp*]:
satisfiable ((*mset* $\circ$ *remdups*) ‘ N) $\longleftrightarrow$ *satisfiable* (*mset* ‘ N)
⟨*proof*⟩

We need some functions to convert between our abstract state *nat* $cdcl_W$*-state* and the concrete state *'v* $cdcl_W$*-state-inv-st*.

abbreviation *convertC* :: *'a list option* $\Rightarrow$ *'a multiset option* **where**
convertC $\equiv$ *map-option mset*

lemma *convert-Propagated*[*elim!*]:
mmset-of-mlit' z = *Propagated* L $C \Longrightarrow (\exists C'. z =$ *Propagated* L $C' \wedge C =$ *mset* C')
⟨*proof*⟩

lemma *get-rev-level-map-convert*:
get-rev-level (*map mmset-of-mlit' M*) n x = *get-rev-level M n x*
⟨*proof*⟩

lemma *get-level-map-convert*[*simp*]:
get-level (*map mmset-of-mlit' M*) = *get-level M*
⟨*proof*⟩

lemma *get-rev-level-map-mmsetof-mlit*[*simp*]:
get-rev-level (*map mmset-of-mlit M*) = *get-rev-level M*
⟨*proof*⟩

lemma *get-level-map-mmsetof-mlit*[*simp*]:
get-level (*map mmset-of-mlit M*) = *get-level M*
⟨*proof*⟩

lemma *get-maximum-level-map-convert*[*simp*]:
get-maximum-level (*map mmset-of-mlit′ M*) *D* = *get-maximum-level M D*
⟨*proof*⟩

lemma *get-all-levels-of-ann-map-convert*[*simp*]:
get-all-levels-of-ann (*map mmset-of-mlit′ M*) = (*get-all-levels-of-ann M*)
⟨*proof*⟩

lemma *reduce-trail-to-empty-trail*[*simp*]:
reduce-trail-to F ([], *aa*, *ab*, *ac*, *b*) = ([], *aa*, *ab*, *ac*, *b*)
⟨*proof*⟩

lemma *raw-trail-reduce-trail-to-length-le*:
assumes *length F* > *length* (*raw-trail S*)
shows *raw-trail* (*reduce-trail-to F S*) = []
⟨*proof*⟩

lemma *reduce-trail-to*:
reduce-trail-to F S =
((*if length* (*raw-trail S*) ≥ *length F*
then drop (*length* (*raw-trail S*) − *length F*) (*raw-trail S*)
else []), *raw-init-clss S*, *raw-learned-clss S*, *raw-backtrack-lvl S*, *raw-conflicting S*)
(**is** *?S* = -)
⟨*proof*⟩

Definition an abstract type

typedef *′v* $cdcl_W$*-state-inv* = {*S*::*′v* $cdcl_W$*-state-inv-st*. $cdcl_W$*-all-struct-inv S*}
morphisms *rough-state-of state-of*
⟨*proof*⟩

instantiation $cdcl_W$*-state-inv* :: (*type*) *equal*
begin
definition *equal-*$cdcl_W$*-state-inv* :: *′v* $cdcl_W$*-state-inv* ⇒ *′v* $cdcl_W$*-state-inv* ⇒ *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* ⟷ *undefined-lit M L*
⟨*proof*⟩

lemma *true-annot-map-convert*[*simp*]: *map mmset-of-mlit′ M* ⊨*a N* ⟷ *M* ⊨*a N*
⟨*proof*⟩

lemma *true-annots-map-convert*[*simp*]: *map mmset-of-mlit′ M* ⊨*as N* ⟷ *M* ⊨*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-propgate-step*:
  do-propagate-step $S \neq S \Longrightarrow$ *propagate* S (*do-propagate-step* S)
  ⟨*proof*⟩

lemma *do-propagate-step-option*[*simp*]:
  conflicting $S \neq None \Longrightarrow$ *do-propagate-step* $S = S$
  ⟨*proof*⟩
thm *prod-cases*

lemma *do-propagate-step-no-step*:
  assumes *dist*: $\forall\, c \in set$ (*raw-clauses* S). *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 $M\ (N\ \#\ Ns) = (if\ (\forall\, c \in set\ N.\ -c \in lits\text{-}of\text{-}l\ M)\ then\ Some\ N\ else\ find\text{-}conflict\ M\ Ns)$

lemma *find-conflict-Some*:
  find-conflict $M\ Ns = Some\ N \Longrightarrow N \in set\ Ns \wedge M \models as\ CNot\ (mset\ N)$
  ⟨*proof*⟩

lemma *find-conflict-None*:
  find-conflict $M\ Ns = None \longleftrightarrow (\forall\, N \in set\ Ns.\ \neg M \models as\ CNot\ (mset\ N))$
  ⟨*proof*⟩

lemma *find-conflict-None-no-confl*:
  find-conflict $M\ (N@U) = None \longleftrightarrow$ *no-step conflict* $(M,\ N,\ U,\ k,\ None)$
  ⟨*proof*⟩

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 $a \Rightarrow (M,\ N,\ U,\ k,\ Some\ a)$
      | *None* $\Rightarrow (M,\ N,\ U,\ k,\ None))$

| $S \Rightarrow S$)

lemma *do-conflict-step*:
do-conflict-step $S \neq S \Longrightarrow$ *conflict* S (*do-conflict-step* S)
⟨*proof*⟩

lemma *do-conflict-step-no-step*:
do-conflict-step $S = S \Longrightarrow$ *no-step conflict* S
⟨*proof*⟩

lemma *do-conflict-step-option*[*simp*]:
conflicting $S \neq$ *None* $\Longrightarrow$ *do-conflict-step* $S = S$
⟨*proof*⟩

lemma *do-conflict-step-conflicting*[*dest*]:
do-conflict-step $S \neq S \Longrightarrow$ *conflicting* (*do-conflict-step* S) $\neq$ *None*
⟨*proof*⟩

definition *do-cp-step* **where**
do-cp-step S =
(*do-propagate-step o do-conflict-step*) S

lemma *cp-step-is-cdcl*$_W$*-cp*:
assumes H: *do-cp-step* $S \neq S$
shows *cdcl*$_W$*-cp* S (*do-cp-step* S)
⟨*proof*⟩

lemma *do-cp-step-eq-no-prop-no-confl*:
do-cp-step $S = S \Longrightarrow$ *do-conflict-step* $S = S \wedge$ *do-propagate-step* $S = S$
⟨*proof*⟩

lemma *no-cdcl*$_W$*-cp-iff-no-propagate-no-conflict*:
no-step cdcl$_W$*-cp* $S \longleftrightarrow$ *no-step propagate* $S \wedge$ *no-step conflict* S
⟨*proof*⟩

lemma *do-cp-step-eq-no-step*:
assumes
H: *do-cp-step* $S = S$ **and**
$\forall c \in$ *set* (*raw-init-clss* S @ *raw-learned-clss* S). *distinct* c
shows *no-step cdcl*$_W$*-cp* S
⟨*proof*⟩

lemma *cdcl*$_W$*-cp-cdcl*$_W$*-st*: *cdcl*$_W$*-cp* $S\ S' \Longrightarrow$ *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* $S \Longrightarrow$ *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* ⇒ *'v cdcl$_W$-state-inv-st* **where**
do-skip-step (*Propagated L C # Ls,N,U,k, Some D*) =
  (*if* −*L* ∉ *set D* ∧ *D* ≠ []
  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 ≠ *S* ⟹ *skip S* (*do-skip-step S*)
  ⟨*proof*⟩

lemma *do-skip-step-no*:
  do-skip-step S = *S* ⟹ *no-step skip S*
  ⟨*proof*⟩

lemma *do-skip-step-trail-is-None*[*iff*]:
  do-skip-step S = (*a, b, c, d, None*) ⟷ *S* = (*a, b, c, d, None*)
  ⟨*proof*⟩

Resolve **fun** *maximum-level-code*:: *'a literal list* ⇒ (*'a, nat, 'b*) *ann-lit list* ⇒ *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*) *ann-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* ⇒ *'v cdcl$_W$-state-inv-st* **where**
do-resolve-step (*Propagated L C # Ls, N, U, k, Some D*) =
  (*if* −*L* ∈ *set D* ∧ *maximum-level-code* (*remove1* (−*L*) *D*) (*Propagated L C # Ls*) = *k*
  then (*Ls, N, U, k, Some* (*remdups* (*remove1 L C* @ *remove1* (−*L*) *D*)))
  else (*Propagated L C # Ls, N, U, k, Some D*)) |
do-resolve-step S = *S*

lemma *do-resolve-step*:
  cdcl$_W$-all-struct-inv S ⟹ *do-resolve-step S* ≠ *S*
  ⟹ *resolve S* (*do-resolve-step S*)
⟨*proof*⟩

lemma *do-resolve-step-no*:
  do-resolve-step S = *S* ⟹ *no-step resolve S*
  ⟨*proof*⟩

lemma *rough-state-of-state-of-resolve*[*simp*]:
  cdcl$_W$-all-struct-inv S ⟹ *rough-state-of* (*state-of* (*do-resolve-step S*)) = *do-resolve-step S*
  ⟨*proof*⟩

lemma *do-resolve-step-trail-is-None*[*iff*]:
  do-resolve-step S = (*a, b, c, d, None*) ⟷ *S* = (*a, b, c, d, None*)

⟨*proof*⟩

Backjumping **fun** *find-level-decomp* **where**

find-level-decomp M [] *D k* = *None* |
find-level-decomp M (*L* # *Ls*) *D k* =
  (*case* (*get-level M L*, *maximum-level-code* (*D* @ *Ls*) *M*) *of*
    (*i*, *j*) ⇒ *if i* = *k* ∧ *j* < *i then Some* (*L*, *j*) *else find-level-decomp M Ls* (*L*#*D*) *k*
)

lemma *find-level-decomp-some*:
  assumes *find-level-decomp M Ls D k* = *Some* (*L*, *j*)
  shows *L* ∈ *set Ls* ∧ *get-maximum-level M* (*mset* (*remove1 L* (*Ls* @ *D*))) = *j* ∧ *get-level M L* = *k*
  ⟨*proof*⟩

lemma *find-level-decomp-none*:
  assumes *find-level-decomp M Ls E k* = *None* **and** *mset* (*L*#*D*) = *mset* (*Ls* @ *E*)
  shows ¬(*L* ∈ *set Ls* ∧ *get-maximum-level M* (*mset D*) < *k* ∧ *k* = *get-level M L*)
  ⟨*proof*⟩

fun *bt-cut* **where**
bt-cut i (*Propagated - -* # *Ls*) = *bt-cut i Ls* |
bt-cut i (*Decided K k* # *Ls*) = (*if k* = *Suc i then Some* (*Decided K k* # *Ls*) *else bt-cut i Ls*) |
bt-cut i [] = *None*

lemma *bt-cut-some-decomp*:
  bt-cut i M = *Some M'* ⟹ ∃ *K M2 M1*. *M* = *M2* @ *M'* ∧ *M'* = *Decided K* (*i*+*1*) # *M1*
  ⟨*proof*⟩

lemma *bt-cut-not-none*: *M* = *M2* @ *Decided K* (*Suc i*) # *M'* ⟹ *bt-cut i M* ≠ *None*
  ⟨*proof*⟩

lemma *get-all-ann-decomposition-ex*:
  ∃ *N*. (*Decided K* (*Suc i*) # *M'*, *N*) ∈ *set* (*get-all-ann-decomposition* (*M2*@*Decided K* (*Suc i*) # *M'*))
  ⟨*proof*⟩

lemma *bt-cut-in-get-all-ann-decomposition*:
  bt-cut i M = *Some M'* ⟹ ∃ *M2*. (*M'*, *M2*) ∈ *set* (*get-all-ann-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 ⇒ (*M*, *N*, *U*, *k*, *Some D*)
  | *Some* (*L*, *j*) ⇒
    (*case bt-cut j M of*
      Some (*Decided - -* # *Ls*) ⇒ (*Propagated L D* # *Ls*, *N*, *D* # *U*, *j*, *None*)
    | *-* ⇒ (*M*, *N*, *U*, *k*, *Some D*))
) |
do-backtrack-step S = *S*

lemma *get-all-ann-decomposition-map-convert*:
  (*get-all-ann-decomposition* (*map mmset-of-mlit' M*)) =
    map (λ(*a*, *b*). (*map mmset-of-mlit' a*, *map mmset-of-mlit' b*)) (*get-all-ann-decomposition M*)
  ⟨*proof*⟩

lemma *do-backtrack-step*:
  assumes
    db: $do\text{-}backtrack\text{-}step\ S \neq S$ **and**
    inv: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
  shows $backtrack\ S\ (do\text{-}backtrack\text{-}step\ S)$
  ⟨*proof*⟩

lemma *map-eq-list-length*:
  $map\ f\ L = L' \Longrightarrow length\ L = length\ L'$
  ⟨*proof*⟩

lemma *map-mmset-of-mlit-eq-cons*:
  assumes $map\ mmset\text{-}of\text{-}mlit'\ M = a\ @\ c$
  obtains $a'\ c'$ **where**
    $M = a'\ @\ c'$ **and**
    $a = map\ mmset\text{-}of\text{-}mlit'\ a'$ **and**
    $c = map\ mmset\text{-}of\text{-}mlit'\ c'$
  ⟨*proof*⟩

lemma *do-backtrack-step-no*:
  assumes
    db: $do\text{-}backtrack\text{-}step\ S = S$ **and**
    inv: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
  shows $no\text{-}step\ backtrack\ S$
⟨*proof*⟩

lemma *rough-state-of-state-of-backtrack*[*simp*]:
  assumes *inv*: $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$
  shows $rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}backtrack\text{-}step\ S)) = do\text{-}backtrack\text{-}step\ S$
⟨*proof*⟩

Decide

fun *do-decide-step* **where**
$do\text{-}decide\text{-}step\ (M,\ N,\ U,\ k,\ None) =$
  $(case\ find\text{-}first\text{-}unused\text{-}var\ N\ (lits\text{-}of\text{-}l\ M)\ of$
    $None \Rightarrow (M,\ N,\ U,\ k,\ None)$
  $|\ Some\ L \Rightarrow (Decided\ L\ (Suc\ k)\ \#\ M,\ N,\ U,\ k{+}1,\ None))\ |$
$do\text{-}decide\text{-}step\ S = S$

lemma *do-decide-step*:
  fixes $S :: 'v\ cdcl_W\text{-}state\text{-}inv\text{-}st$
  assumes $do\text{-}decide\text{-}step\ S \neq S$
  shows $decide\ S\ (do\text{-}decide\text{-}step\ S)$
  ⟨*proof*⟩

lemma *mmset-of-mlit'-eq-Decided*[*iff*]: $mmset\text{-}of\text{-}mlit'\ z = Decided\ x\ k \longleftrightarrow z = Decided\ x\ k$
  ⟨*proof*⟩

lemma *do-decide-step-no*:
  $do\text{-}decide\text{-}step\ S = S \Longrightarrow no\text{-}step\ decide\ S$
  ⟨*proof*⟩

lemma *rough-state-of-state-of-do-decide-step*[*simp*]:
  $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \Longrightarrow rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}decide\text{-}step\ S)) = do\text{-}decide\text{-}step\ S$
⟨*proof*⟩

lemma *rough-state-of-state-of-do-skip-step*[*simp*]:
$cdcl_W$*-all-struct-inv S* $\Longrightarrow$ *rough-state-of* (*state-of* (*do-skip-step S*)) = *do-skip-step S*
⟨*proof*⟩

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 = *state-of* (*if* $cdcl_W$*-all-struct-inv xs then xs else* ([], [], [], *0*, *None*))

lemma [*code abstype*]:
Con (*rough-state-of S*) = *S*
⟨*proof*⟩

definition *do-cp-step′* **where**
do-cp-step′ S = *state-of* (*do-cp-step* (*rough-state-of S*))

typedef *′v* $cdcl_W$*-state-inv-from-init-state* = {*S*::*′v* $cdcl_W$*-state-inv-st.* $cdcl_W$*-all-struct-inv S*
∧ $cdcl_W$*-stgy*** (*raw-S0-*$cdcl_W$ (*raw-init-clss S*)) *S*}
morphisms *rough-state-from-init-state-of state-from-init-state-of*
⟨*proof*⟩

instantiation $cdcl_W$*-state-inv-from-init-state* :: (*type*) *equal*
begin
definition *equal-*$cdcl_W$*-state-inv-from-init-state* :: *′v* $cdcl_W$*-state-inv-from-init-state* ⇒
′v $cdcl_W$*-state-inv-from-init-state* ⇒ *bool* **where**
equal-$cdcl_W$*-state-inv-from-init-state S S′* ⟷
(*rough-state-from-init-state-of S* = *rough-state-from-init-state-of S′*)
instance
⟨*proof*⟩
end

definition *ConI* **where**
ConI S = *state-from-init-state-of* (*if* $cdcl_W$*-all-struct-inv S*
∧ $cdcl_W$*-stgy*** (*raw-S0-*$cdcl_W$ (*raw-init-clss S*)) *S then S else* ([], [], [], *0*, *None*))

lemma [*code abstype*]:
ConI (*rough-state-from-init-state-of S*) = *S*
⟨*proof*⟩

definition *id-of-I-to* :: *′v* $cdcl_W$*-state-inv-from-init-state* ⇒ *′v* $cdcl_W$*-state-inv* **where**
id-of-I-to S = *state-of* (*rough-state-from-init-state-of S*)

lemma [*code abstract*]:
rough-state-of (*id-of-I-to S*) = *rough-state-from-init-state-of S*
⟨*proof*⟩

Conflict and Propagate **function** *do-full1-cp-step* :: *′v* $cdcl_W$*-state-inv* ⇒ *′v* $cdcl_W$*-state-inv*
where
do-full1-cp-step S =
(*let S′* = *do-cp-step′ S in*
if S = *S′ then S else do-full1-cp-step S′*)
⟨*proof*⟩

termination
⟨*proof*⟩

lemma *do-full1-cp-step-fix-point-of-do-full1-cp-step*:
do-cp-step(*rough-state-of* (*do-full1-cp-step S*)) = *rough-state-of* (*do-full1-cp-step S*)
⟨*proof*⟩

lemma *in-clauses-rough-state-of-is-distinct*:
c∈*set* (*raw-init-clss* (*rough-state-of S*) @ *raw-learned-clss* (*rough-state-of S*)) ⟹ *distinct c*
⟨*proof*⟩

lemma *do-full1-cp-step-full*:
full $cdcl_W$*-cp* (*rough-state-of S*)
(*rough-state-of* (*do-full1-cp-step S*))
⟨*proof*⟩

lemma [*code abstract*]:
rough-state-of (*do-cp-step′ S*) = *do-cp-step* (*rough-state-of S*)
⟨*proof*⟩

The other rules **fun** *do-other-step* **where**
do-other-step S =
(*let T* = *do-skip-step S in*
if T ≠ *S*
then T
else
(*let U* = *do-resolve-step T in*
if U ≠ *T*
then U else
(*let V* = *do-backtrack-step U in*
if V ≠ *U then V else do-decide-step V*)))

lemma *do-other-step*:
assumes *inv*: $cdcl_W$*-all-struct-inv S* **and**
st: *do-other-step S* ≠ *S*
shows $cdcl_W$*-o S* (*do-other-step S*)
⟨*proof*⟩

lemma *do-other-step-no*:
assumes *inv*: $cdcl_W$*-all-struct-inv S* **and**
st: *do-other-step S* = *S*
shows *no-step* $cdcl_W$*-o S*
⟨*proof*⟩

lemma *rough-state-of-state-of-do-other-step*[*simp*]:
rough-state-of (*state-of* (*do-other-step* (*rough-state-of S*))) = *do-other-step* (*rough-state-of S*)
⟨*proof*⟩

definition *do-other-step′* **where**
do-other-step′ S =
state-of (*do-other-step* (*rough-state-of S*))

lemma *rough-state-of-do-other-step′*[*code abstract*]:
rough-state-of (*do-other-step′ S*) = *do-other-step* (*rough-state-of S*)
⟨*proof*⟩

definition *do-cdcl*$_W$*-stgy-step* **where**
do-cdcl$_W$*-stgy-step S* =
  (*let T* = *do-full1-cp-step S in*
    if T $\neq$ *S*
    then T
    else
      (*let U* = (*do-other-step′ T*) *in*
       (*do-full1-cp-step U*)))

definition *do-cdcl*$_W$*-stgy-step′* **where**
do-cdcl$_W$*-stgy-step′ S* = *state-from-init-state-of* (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*id-of-I-to S*)))

lemma *toS-do-full1-cp-step-not-eq*: *do-full1-cp-step S* $\neq$ *S* $\Longrightarrow$
  rough-state-of S $\neq$ *rough-state-of* (*do-full1-cp-step S*)
⟨*proof*⟩

do-full1-cp-step should not be unfolded anymore:

declare *do-full1-cp-step.simps*[*simp del*]

Correction of the transformation

lemma *do-cdcl*$_W$*-stgy-step*:
  assumes *do-cdcl*$_W$*-stgy-step S* $\neq$ *S*
  shows *cdcl*$_W$*-stgy* (*rough-state-of S*) (*rough-state-of* (*do-cdcl*$_W$*-stgy-step S*))
⟨*proof*⟩

lemma *do-skip-step-trail-changed-or-conflict*:
  assumes *d*: *do-other-step S* $\neq$ *S*
  and *inv*: *cdcl*$_W$*-all-struct-inv S*
  shows *trail S* $\neq$ *trail* (*do-other-step S*)
⟨*proof*⟩

lemma *do-full1-cp-step-induct*:
  ($\bigwedge$*S*. (*S* $\neq$ *do-cp-step′ S* $\Longrightarrow$ *P* (*do-cp-step′ S*)) $\Longrightarrow$ *P S*) $\Longrightarrow$ *P a0*
  ⟨*proof*⟩

lemma *do-cp-step-neq-trail-increase*:
  $\exists$ *c. raw-trail* (*do-cp-step S*) = *c* @ *raw-trail S* $\wedge$($\forall$ *m* $\in$ *set c*. $\neg$ *is-decided m*)
  ⟨*proof*⟩

lemma *do-full1-cp-step-neq-trail-increase*:
  $\exists$ *c. raw-trail* (*rough-state-of* (*do-full1-cp-step S*)) = *c* @ *raw-trail* (*rough-state-of S*)
    $\wedge$ ($\forall$ *m* $\in$ *set c*. $\neg$ *is-decided m*)
  ⟨*proof*⟩

lemma *do-cp-step-conflicting*:
  conflicting (*rough-state-of S*) $\neq$ *None* $\Longrightarrow$ *do-cp-step′ S* = *S*
  ⟨*proof*⟩

lemma *do-full1-cp-step-conflicting*:
  conflicting (*rough-state-of S*) $\neq$ *None* $\Longrightarrow$ *do-full1-cp-step S* = *S*
  ⟨*proof*⟩

lemma *do-decide-step-not-conflicting-one-more-decide*:
  assumes
    conflicting S = *None* **and**

$do\text{-}decide\text{-}step\ S \neq S$
shows $Suc\ (length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ S)))$
$= length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
$\langle proof \rangle$

lemma *do-decide-step-not-conflicting-one-more-decide-bt*:
assumes $conflicting\ S \neq None$ **and**
$do\text{-}decide\text{-}step\ S \neq S$
shows $length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ S)) <$
$length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (do\text{-}decide\text{-}step\ S)))$
$\langle proof \rangle$

lemma *do-other-step-not-conflicting-one-more-decide-bt*:
assumes
$conflicting\ (rough\text{-}state\text{-}of\ S) \neq None$ **and**
$conflicting\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S)) = None$ **and**
$do\text{-}other\text{-}step'\ S \neq S$
shows $length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)))$
$> length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S))))$
$\langle proof \rangle$

lemma *do-other-step-not-conflicting-one-more-decide*:
assumes $conflicting\ (rough\text{-}state\text{-}of\ S) = None$ **and**
$do\text{-}other\text{-}step'\ S \neq S$
shows $1 + length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ S)))$
$= length\ (filter\ is\text{-}decided\ (raw\text{-}trail\ (rough\text{-}state\text{-}of\ (do\text{-}other\text{-}step'\ S))))$
$\langle proof \rangle$

lemma *rough-state-of-state-of-do-skip-step-rough-state-of*[*simp*]:
$rough\text{-}state\text{-}of\ (state\text{-}of\ (do\text{-}skip\text{-}step\ (rough\text{-}state\text{-}of\ S))) = do\text{-}skip\text{-}step\ (rough\text{-}state\text{-}of\ S)$
$\langle proof \rangle$

lemma *conflicting-do-resolve-step-iff*[*iff*]:
$conflicting\ (do\text{-}resolve\text{-}step\ S) = None \longleftrightarrow conflicting\ S = None$
$\langle proof \rangle$

lemma *conflicting-do-skip-step-iff*[*iff*]:
$conflicting\ (do\text{-}skip\text{-}step\ S) = None \longleftrightarrow conflicting\ S = None$
$\langle proof \rangle$

lemma *conflicting-do-decide-step-iff*[*iff*]:
$conflicting\ (do\text{-}decide\text{-}step\ S) = None \longleftrightarrow conflicting\ S = None$
$\langle proof \rangle$

lemma *conflicting-do-backtrack-step-imp*[*simp*]:
$do\text{-}backtrack\text{-}step\ S \neq S \Longrightarrow conflicting\ (do\text{-}backtrack\text{-}step\ S) = None$
$\langle proof \rangle$

lemma *do-skip-step-eq-iff-trail-eq*:
$do\text{-}skip\text{-}step\ S = S \longleftrightarrow trail\ (do\text{-}skip\text{-}step\ S) = trail\ S$
$\langle proof \rangle$

lemma *do-decide-step-eq-iff-trail-eq*:
$do\text{-}decide\text{-}step\ S = S \longleftrightarrow trail\ (do\text{-}decide\text{-}step\ S) = trail\ S$
$\langle proof \rangle$

lemma *do-backtrack-step-eq-iff-trail-eq*:
do-backtrack-step $S = S \longleftrightarrow$ *raw-trail* (*do-backtrack-step* S) = *raw-trail* S
⟨*proof*⟩

lemma *do-resolve-step-eq-iff-trail-eq*:
do-resolve-step $S = S \longleftrightarrow$ *trail* (*do-resolve-step* S) = *trail* S
⟨*proof*⟩

lemma *do-other-step-eq-iff-trail-eq*:
do-other-step $S = S \longleftrightarrow$ *raw-trail* (*do-other-step* S) = *raw-trail* S

⟨*proof*⟩

lemma *do-full1-cp-step-do-other-step'-normal-form*[*dest*!]:
assumes H: *do-full1-cp-step* (*do-other-step'* S) = S
shows *do-other-step'* $S = S \wedge$ *do-full1-cp-step* $S = S$
⟨*proof*⟩

lemma *do-cdcl$_W$-stgy-step-no*:
assumes S: *do-cdcl$_W$-stgy-step* $S = S$
shows *no-step cdcl$_W$-stgy* (*rough-state-of* S)
⟨*proof*⟩

lemma *toS-rough-state-of-state-of-rough-state-from-init-state-of*[*simp*]:
rough-state-of (*state-of* (*rough-state-from-init-state-of* S))
= *rough-state-from-init-state-of* S
⟨*proof*⟩

lemma *cdcl$_W$-cp-is-rtranclp-cdcl$_W$*: *cdcl$_W$-cp* S $T \Longrightarrow$ *cdcl$_W$*** S T
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-cp-is-rtranclp-cdcl$_W$*: *cdcl$_W$-cp*** S $T \Longrightarrow$ *cdcl$_W$*** S T
⟨*proof*⟩

lemma *cdcl$_W$-stgy-is-rtranclp-cdcl$_W$*:
cdcl$_W$-stgy S $T \Longrightarrow$ *cdcl$_W$*** S T
⟨*proof*⟩

lemma *cdcl$_W$-stgy-init-clss*: *cdcl$_W$-stgy* S $T \Longrightarrow$ *cdcl$_W$-M-level-inv* $S \Longrightarrow$ *init-clss* S = *init-clss* T
⟨*proof*⟩

lemma *clauses-toS-rough-state-of-do-cdcl$_W$-stgy-step*[*simp*]:
init-clss (*rough-state-of* (*do-cdcl$_W$-stgy-step* (*state-of* (*rough-state-from-init-state-of* S))))
= *init-clss* (*rough-state-from-init-state-of* S) (**is** - = *init-clss* *?S*)
⟨*proof*⟩

lemma *raw-init-clss-do-cp-step*[*simp*]:
raw-init-clss (*do-cp-step* S) = *raw-init-clss* S
⟨*proof*⟩
lemma *raw-init-clss-do-cp-step'*[*simp*]:
raw-init-clss (*rough-state-of* (*do-cp-step'* S)) = *raw-init-clss* (*rough-state-of* S)
⟨*proof*⟩

lemma *raw-init-clss-rough-state-of-do-full1-cp-step*[*simp*]:

raw-init-clss (*rough-state-of* (*do-full1-cp-step* S))
= *raw-init-clss* (*rough-state-of* S)
⟨*proof*⟩

lemma *raw-init-clss-do-skip-def*[*simp*]:
raw-init-clss (*do-skip-step* S) = *raw-init-clss* S
⟨*proof*⟩

lemma *raw-init-clss-do-resolve-def*[*simp*]:
raw-init-clss (*do-resolve-step* S) = *raw-init-clss* S
⟨*proof*⟩

lemma *raw-init-clss-do-backtrack-def*[*simp*]:
raw-init-clss (*do-backtrack-step* S) = *raw-init-clss* S
⟨*proof*⟩

lemma *raw-init-clss-do-decide-def*[*simp*]:
raw-init-clss (*do-decide-step* S) = *raw-init-clss* S
⟨*proof*⟩

lemma *raw-init-clss-rough-state-of-do-other-step'*[*simp*]:
raw-init-clss (*rough-state-of* (*do-other-step'* S))
= *raw-init-clss* (*rough-state-of* S)
⟨*proof*⟩

lemma [*simp*]:
raw-init-clss (*rough-state-of* (*do-cdcl*$_W$*-stgy-step* (*state-of* (*rough-state-from-init-state-of* S))))
=
raw-init-clss (*rough-state-from-init-state-of* S)
⟨*proof*⟩

lemma *rough-state-from-init-state-of-do-cdcl*$_W$*-stgy-step'*[*code abstract*]:
rough-state-from-init-state-of (*do-cdcl*$_W$*-stgy-step'* S) =
rough-state-of (*do-cdcl*$_W$*-stgy-step* (*id-of-I-to* S))
⟨*proof*⟩

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'* $S \neq S \Longrightarrow P$ (*do-cdcl*$_W$*-stgy-step'* S)) $\Longrightarrow P\ S$) $\Longrightarrow P\ a0$
⟨*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-rtranclp-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 ($E \neq$ *Some* [] $\wedge$ *satisfiable* (*set* (*map mset* N)))
$\vee$ ($E =$ *Some* [] $\wedge$ *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 NOTs 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-learning*$_W$
begin
declare *cdcl*$_W$*.intros*[*intro*] *cdcl*$_W$*-bj.intros*[*intro*] *cdcl*$_W$*-o.intros*[*intro*]

lemma *backtrack-no-cdcl*$_W$*-bj*:
assumes *cdcl*: *cdcl*$_W$*-bj* T U **and** *inv*: *cdcl*$_W$*-M-level-inv* V

shows $\neg$*backtrack V T*
⟨*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 *rtranclp-cdcl*$_W$*-bj-skip-or-resolve-backtrack*:
assumes *cdcl*$_W$*-bj*** *S U* **and** *inv*: *cdcl*$_W$*-M-level-inv S*
shows *skip-or-resolve*** *S U* ∨ (∃ *T*. *skip-or-resolve*** *S T* ∧ *backtrack T U*)
⟨*proof*⟩

lemma *rtranclp-skip-or-resolve-rtranclp-cdcl*$_W$:
skip-or-resolve** *S T* ⟹ *cdcl*$_W$** *S T*
⟨*proof*⟩

definition *backjump-l-cond* :: *'v clause* ⇒ *'v clause* ⇒ *'v literal* ⇒ *'st* ⇒ *'st* ⇒ *bool* **where**
backjump-l-cond ≡ λ*C C' L' S T*. *True*

definition *inv*$_{NOT}$ :: *'st* ⇒ *bool* **where**
inv$_{NOT}$ ≡ λ*S*. *no-dup* (*trail S*)

declare *inv*$_{NOT}$*-def*[*simp*]
end

context *conflict-driven-clause-learning*$_W$
begin

21.2 More lemmas conflict–propagate and backjumping

21.2.1 Termination

lemma *cdcl*$_W$*-cp-normalized-element-all-inv*:
assumes *inv*: *cdcl*$_W$*-all-struct-inv S*
obtains *T* **where** *full cdcl*$_W$*-cp S T*
⟨*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*)
⟨*proof*⟩

lemma *wf-cdcl*$_W$*-bj*:
wf {(*b*,*a*). *cdcl*$_W$*-bj a b* ∧ *cdcl*$_W$*-M-level-inv a*}
⟨*proof*⟩

lemma *cdcl*$_W$*-bj-exists-normal-form*:
assumes *lev*: *cdcl*$_W$*-M-level-inv S*
shows ∃ *T*. *full cdcl*$_W$*-bj S T*
⟨*proof*⟩
lemma *rtranclp-skip-state-decomp*:
assumes *skip*** *S T* **and** *no-dup* (*trail S*)
shows

$\exists M.$ *trail* $S = M$ @ *trail* $T \wedge (\forall m\in$*set* $M. \neg$*is-decided* $m)$
init-clss S = *init-clss* T
learned-clss S = *learned-clss* T
backtrack-lvl S = *backtrack-lvl* T
conflicting S = *conflicting* T
⟨*proof*⟩

21.2.2 More backjumping

Backjumping after skipping or jump directly **lemma** *rtranclp-skip-backtrack-backtrack*:
assumes
skip** S T **and**
backtrack T W **and**
$cdcl_W$*-all-struct-inv* S
shows *backtrack* S W
⟨*proof*⟩

lemma *fst-get-all-ann-decomposition-prepend-not-decided*:
assumes $\forall m\in$*set* $MS. \neg$ *is-decided* m
shows *set* (*map fst* (*get-all-ann-decomposition* M))
= *set* (*map fst* (*get-all-ann-decomposition* (MS @ M)))
⟨*proof*⟩

See also ⟦*skip*** *?S ?T*; *backtrack ?T ?W*; $cdcl_W$*-all-struct-inv ?S*⟧ $\Longrightarrow$ *backtrack ?S ?W*

lemma *rtranclp-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
⟨*proof*⟩

lemma $cdcl_W$*-bj-decomp-resolve-skip-and-bj*:
assumes $cdcl_W$*-bj*** S T **and** *inv*: $cdcl_W$*-M-level-inv* S
shows (*skip-or-resolve*** S T
$\vee$ ($\exists U.$ *skip-or-resolve*** S U $\wedge$ *backtrack* U T))
⟨*proof*⟩

lemma *resolve-skip-deterministic*:
resolve S T $\Longrightarrow$ *skip* S U $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *backtrack-unique*:
assumes
bt-T: *backtrack* S T **and**
bt-U: *backtrack* S U **and**
inv: $cdcl_W$*-all-struct-inv* S
shows $T \sim U$
⟨*proof*⟩

lemma *if-can-apply-backtrack-no-more-resolve*:
assumes
skip: *skip*** S U **and**
bt: *backtrack* S T **and**
inv: $cdcl_W$*-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: $cdcl_W$*-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-or-resolve*** S U **and**
inv: $cdcl_W$*-all-struct-inv* S
shows *skip*** S U
$\langle proof \rangle$

lemma $cdcl_W$*-bj-bj-decomp*:
assumes $cdcl_W$*-bj*** S W **and** $cdcl_W$*-all-struct-inv* S
shows
($\exists$ T U V. (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** S T
$\wedge$ (λT U. *resolve* T U $\wedge$ *no-step backtrack* T) T U
$\wedge$ *skip*** U V $\wedge$ *backtrack* V W)
$\vee$ ($\exists$ T U. (λS T. *skip-or-resolve* S T $\wedge$ *no-step backtrack* S)** S T
$\wedge$ (λT U. *resolve* T U $\wedge$ *no-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
$cdcl_W$*-bj*** S V **and**
$cdcl_W$*-bj*** S T **and**
n-s: *no-step* $cdcl_W$*-bj* V **and**
inv: $cdcl_W$*-all-struct-inv* S
shows $T \sim V$ $\vee$ $cdcl_W$*-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 $T \sim U$
$\langle proof \rangle$

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 $T \sim U$
$\langle proof \rangle$

21.3 CDCL FW

inductive *cdcl*$_W$*-merge-restart* :: *'st* ⇒ *'st* ⇒ *bool* **where**
fw-r-propagate: *propagate S S'* ⟹ *cdcl*$_W$*-merge-restart S S'* |
fw-r-conflict: *conflict S T* ⟹ *full cdcl*$_W$*-bj T U* ⟹ *cdcl*$_W$*-merge-restart S U* |
fw-r-decide: *decide S S'* ⟹ *cdcl*$_W$*-merge-restart S S'*|
fw-r-rf: *cdcl*$_W$*-rf S S'* ⟹ *cdcl*$_W$*-merge-restart S S'*

lemma *rtranclp-cdcl*$_W$*-bj-rtranclp-cdcl*$_W$:
cdcl$_W$*-bj*** *S T* ⟹ *cdcl*$_W$** *S T*
⟨*proof*⟩

lemma *cdcl*$_W$*-merge-restart-cdcl*$_W$:
assumes *cdcl*$_W$*-merge-restart S T*
shows *cdcl*$_W$** *S T*
⟨*proof*⟩

lemma *cdcl*$_W$*-merge-restart-conflicting-true-or-no-step*:
assumes *cdcl*$_W$*-merge-restart S T*
shows *conflicting T* = *None* ∨ *no-step cdcl*$_W$ *T*
⟨*proof*⟩

inductive *cdcl*$_W$*-merge* :: *'st* ⇒ *'st* ⇒ *bool* **where**
fw-propagate: *propagate S S'* ⟹ *cdcl*$_W$*-merge S S'* |
fw-conflict: *conflict S T* ⟹ *full cdcl*$_W$*-bj T U* ⟹ *cdcl*$_W$*-merge S U* |
fw-decide: *decide S S'* ⟹ *cdcl*$_W$*-merge S S'*|
fw-forget: *forget S S'* ⟹ *cdcl*$_W$*-merge S S'*

lemma *cdcl*$_W$*-merge-cdcl*$_W$*-merge-restart*:
cdcl$_W$*-merge S T* ⟹ *cdcl*$_W$*-merge-restart S T*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-merge-tranclp-cdcl*$_W$*-merge-restart*:
cdcl$_W$*-merge*** *S T* ⟹ *cdcl*$_W$*-merge-restart*** *S T*
⟨*proof*⟩

lemma *cdcl*$_W$*-merge-rtranclp-cdcl*$_W$:
cdcl$_W$*-merge S T* ⟹ *cdcl*$_W$** *S T*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-merge-rtranclp-cdcl*$_W$:
cdcl$_W$*-merge*** *S T* ⟹ *cdcl*$_W$** *S T*
⟨*proof*⟩

lemmas *rulesE* =
skipE resolveE backtrackE propagateE conflictE decideE restartE forgetE

lemma *cdcl*$_W$*-all-struct-inv-tranclp-cdcl*$_W$*-merge-tranclp-cdcl*$_W$*-merge-cdcl*$_W$*-all-struct-inv*:
assumes
inv: *cdcl*$_W$*-all-struct-inv b*
cdcl$_W$*-merge*$^{++}$ *b a*
shows (λ*S T*. *cdcl*$_W$*-all-struct-inv S* ∧ *cdcl*$_W$*-merge S T*)$^{++}$ *b a*
⟨*proof*⟩

lemma *backtrack-is-full1-cdcl*$_W$*-bj*:
assumes *bt*: *backtrack S T* **and** *inv*: *cdcl*$_W$*-M-level-inv S*

shows $full1\ cdcl_W\text{-}bj\ S\ T$
⟨proof⟩

lemma *rtrancl-cdcl*$_W$*-conflicting-true-cdcl*$_W$*-merge-restart*:
assumes $cdcl_W^{**}\ S\ V$ **and** *inv*: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and** $conflicting\ S = None$
shows $(cdcl_W\text{-}merge\text{-}restart^{**}\ S\ V \wedge conflicting\ V = None)$
$\vee (\exists\ T\ U.\ cdcl_W\text{-}merge\text{-}restart^{**}\ S\ T \wedge conflicting\ V \neq None \wedge conflict\ T\ U \wedge cdcl_W\text{-}bj^{**}\ U\ V)$
⟨proof⟩

lemma *no-step-cdcl*$_W$*-no-step-cdcl*$_W$*-merge-restart*: $no\text{-}step\ cdcl_W\ S \Longrightarrow no\text{-}step\ cdcl_W\text{-}merge\text{-}restart\ S$
⟨proof⟩

lemma *no-step-cdcl*$_W$*-merge-restart-no-step-cdcl*$_W$:
assumes
$conflicting\ S = None$ **and**
$cdcl_W\text{-}M\text{-}level\text{-}inv\ S$ **and**
$no\text{-}step\ cdcl_W\text{-}merge\text{-}restart\ S$
shows $no\text{-}step\ cdcl_W\ S$
⟨proof⟩

lemma *cdcl*$_W$*-merge-restart-no-step-cdcl*$_W$*-bj*:
assumes
$cdcl_W\text{-}merge\text{-}restart\ S\ T$
shows $no\text{-}step\ cdcl_W\text{-}bj\ T$
⟨proof⟩

lemma *rtranclp-cdcl*$_W$*-merge-restart-no-step-cdcl*$_W$*-bj*:
assumes
$cdcl_W\text{-}merge\text{-}restart^{**}\ S\ T$ **and**
$conflicting\ S = None$
shows $no\text{-}step\ cdcl_W\text{-}bj\ T$
⟨proof⟩

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 *confl*: $conflicting\ S = None$ **and** *lev*: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $full\ cdcl_W\ S\ V \longleftrightarrow full\ cdcl_W\text{-}merge\text{-}restart\ S\ V$
⟨proof⟩

lemma *init-state-true-full-cdcl*$_W$*-iff-full-cdcl*$_W$*-merge*:
shows $full\ cdcl_W\ (init\text{-}state\ N)\ V \longleftrightarrow full\ cdcl_W\text{-}merge\text{-}restart\ (init\text{-}state\ N)\ V$
⟨proof⟩

21.4 FW with strategy

21.4.1 The intermediate step

inductive $cdcl_W\text{-}s' :: 'st \Rightarrow 'st \Rightarrow bool$ **where**
conflict': $full1\ cdcl_W\text{-}cp\ S\ S' \Longrightarrow cdcl_W\text{-}s'\ S\ S'$ |
decide': $decide\ S\ S' \Longrightarrow no\text{-}step\ cdcl_W\text{-}cp\ S \Longrightarrow full\ cdcl_W\text{-}cp\ S'\ S'' \Longrightarrow cdcl_W\text{-}s'\ S\ S''$ |
bj': $full1\ cdcl_W\text{-}bj\ S\ S' \Longrightarrow no\text{-}step\ cdcl_W\text{-}cp\ S \Longrightarrow full\ cdcl_W\text{-}cp\ S'\ S'' \Longrightarrow cdcl_W\text{-}s'\ S\ S''$

inductive-cases $cdcl_W$*-s′E*: $cdcl_W$*-s′ S T*

lemma *rtranclp-*$cdcl_W$*-bj-full1-cdclp-*$cdcl_W$*-stgy*:
$cdcl_W\text{-}bj^{**}\ S\ S' \Longrightarrow full\ cdcl_W\text{-}cp\ S'\ S'' \Longrightarrow cdcl_W\text{-}stgy^{**}\ S\ S''$
⟨*proof*⟩

lemma $cdcl_W$*-s′-is-rtranclp-*$cdcl_W$*-stgy*:
$cdcl_W\text{-}s'\ S\ T \Longrightarrow cdcl_W\text{-}stgy^{**}\ S\ T$
⟨*proof*⟩

lemma $cdcl_W$*-cp-*$cdcl_W$*-bj-bissimulation*:
assumes
full $cdcl_W$*-cp T U* **and**
$cdcl_W\text{-}bj^{**}\ T\ T'$ **and**
$cdcl_W$*-all-struct-inv T* **and**
no-step $cdcl_W$*-bj T′*
shows *full* $cdcl_W$*-cp T′ U*
$\vee (\exists U'\ U''.\ full\ cdcl_W\text{-}cp\ T'\ U'' \wedge full1\ cdcl_W\text{-}bj\ U\ U' \wedge full\ cdcl_W\text{-}cp\ U'\ U''$
$\wedge\ cdcl_W\text{-}s'^{**}\ U\ U'')$
⟨*proof*⟩

lemma $cdcl_W$*-cp-*$cdcl_W$*-bj-bissimulation′*:
assumes
full $cdcl_W$*-cp T U* **and**
$cdcl_W\text{-}bj^{**}\ T\ T'$ **and**
$cdcl_W$*-all-struct-inv T* **and**
no-step $cdcl_W$*-bj T′*
shows *full* $cdcl_W$*-cp T′ U*
$\vee (\exists U'.\ full1\ cdcl_W\text{-}bj\ U\ U' \wedge (\forall U''.\ full\ cdcl_W\text{-}cp\ U'\ U'' \longrightarrow full\ cdcl_W\text{-}cp\ T'\ U''$
$\wedge\ cdcl_W\text{-}s'^{**}\ U\ U''))$
⟨*proof*⟩

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*
$\vee (\exists U'.\ full1\ cdcl_W\text{-}bj\ U\ U' \wedge (\forall U''.\ full\ cdcl_W\text{-}cp\ U'\ U'' \longrightarrow cdcl_W\text{-}s'\ S\ U''))$
⟨*proof*⟩

lemma $cdcl_W$*-stgy-*$cdcl_W$*-s′-connected′*:
assumes $cdcl_W$*-stgy S U* **and** $cdcl_W$*-all-struct-inv S*
shows $cdcl_W$*-s′ S U*
$\vee (\exists U'\ U''.\ cdcl_W\text{-}s'\ S\ U'' \wedge full1\ cdcl_W\text{-}bj\ U\ U' \wedge full\ cdcl_W\text{-}cp\ U'\ U'')$
⟨*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 *rtranclp-*$cdcl_W$*-stgy-connected-to-rtranclp-*$cdcl_W$*-s′*:
assumes $cdcl_W\text{-}stgy^{**}\ S\ U$ **and** *inv*: $cdcl_W$*-M-level-inv S*
shows $cdcl_W\text{-}s'^{**}\ S\ U \vee (\exists T.\ cdcl_W\text{-}s'^{**}\ S\ T \wedge cdcl_W\text{-}bj^{++}\ T\ U \wedge conflicting\ U \neq None)$
⟨*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'-tranclp-*$cdcl_W$:
$cdcl_W$*-s'* $S\ S' \Longrightarrow cdcl_W^{++}\ S\ S'$
⟨*proof*⟩

lemma *tranclp-*$cdcl_W$*-s'-tranclp-*$cdcl_W$:
$cdcl_W\text{-}s'^{++}\ S\ S' \Longrightarrow cdcl_W^{++}\ S\ S'$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-s'-rtranclp-*$cdcl_W$:
$cdcl_W\text{-}s'^{**}\ S\ S' \Longrightarrow cdcl_W^{**}\ S\ S'$
⟨*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 $\exists\, T.\ cdcl_W\text{-}stgy\ S\ T$
⟨*proof*⟩

lemma *decide-step-*$cdcl_W$*-stgy-step*:
assumes
decide $S\ T$
$cdcl_W$*-all-struct-inv* S
shows $\exists\, T.\ cdcl_W\text{-}stgy\ S\ T$
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-cp-conflicting-Some*:
$cdcl_W\text{-}cp^{**}\ S\ T \Longrightarrow$ *conflicting* $S =$ *Some* $D \Longrightarrow S = T$
⟨*proof*⟩

inductive $cdcl_W$*-merge-cp* :: *'st* ⇒ *'st* ⇒ *bool* **where**
conflict': *conflict* $S\ T \Longrightarrow$ *full* $cdcl_W$*-bj* $T\ U \Longrightarrow cdcl_W$*-merge-cp* $S\ U$ |
propagate': $propagate^{++}\ S\ S' \Longrightarrow cdcl_W$*-merge-cp* $S\ S'$

lemma $cdcl_W$*-merge-restart-cases*[*consumes 1*, *case-names conflict propagate*]:
assumes
$cdcl_W$*-merge-cp* $S\ U$ **and**
$\bigwedge T.$ *conflict* $S\ T \Longrightarrow$ *full* $cdcl_W$*-bj* $T\ U \Longrightarrow P$ **and**
$propagate^{++}\ S\ U \Longrightarrow P$
shows P
⟨*proof*⟩

lemma $cdcl_W$*-merge-cp-tranclp-*$cdcl_W$*-merge*:
$cdcl_W$*-merge-cp* $S\ T \Longrightarrow cdcl_W\text{-}merge^{++}\ S\ T$
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-cp-rtranclp-cdcl$_W$*:
cdcl$_W$-merge-cp** S T $\Longrightarrow$ *cdcl$_W$** S T*
⟨*proof*⟩

lemma *full1-cdcl$_W$-bj-no-step-cdcl$_W$-bj*:
full1 cdcl$_W$-bj S T $\Longrightarrow$ *no-step cdcl$_W$-cp S*
⟨*proof*⟩

21.4.2 Full Transformation

inductive *cdcl$_W$-s′-without-decide* **where**
conflict′-without-decide[*intro*]: *full1 cdcl$_W$-cp S S′* $\Longrightarrow$ *cdcl$_W$-s′-without-decide S S′* |
bj′-without-decide[*intro*]: *full1 cdcl$_W$-bj S S′* $\Longrightarrow$ *no-step cdcl$_W$-cp S* $\Longrightarrow$ *full cdcl$_W$-cp S′ S″*
$\Longrightarrow$ *cdcl$_W$-s′-without-decide S S″*

lemma *rtranclp-cdcl$_W$-s′-without-decide-rtranclp-cdcl$_W$*:
cdcl$_W$-s′-without-decide** S T $\Longrightarrow$ *cdcl$_W$** S T*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s′-without-decide-rtranclp-cdcl$_W$-s′*:
cdcl$_W$-s′-without-decide** S T $\Longrightarrow$ *cdcl$_W$-s′** S T*
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-cp-is-rtranclp-cdcl$_W$-s′-without-decide*:
assumes
cdcl$_W$-merge-cp** S V
conflicting S = None
shows
(*cdcl$_W$-s′-without-decide** S V*)
$\vee$ ($\exists$ *T. cdcl$_W$-s′-without-decide** S T* $\wedge$ *propagate^{++} T V*)
$\vee$ ($\exists$ *T U. cdcl$_W$-s′-without-decide** S T* $\wedge$ *full1 cdcl$_W$-bj T U* $\wedge$ *propagate** U V*)
⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-s′-without-decide-is-rtranclp-cdcl$_W$-merge-cp*:
assumes
cdcl$_W$-s′-without-decide** S V **and**
confl: *conflicting S = None*
shows
(*cdcl$_W$-merge-cp** S V* $\wedge$ *conflicting V = None*)
$\vee$ (*cdcl$_W$-merge-cp** S V* $\wedge$ *conflicting V* $\neq$ *None* $\wedge$ *no-step cdcl$_W$-cp V* $\wedge$ *no-step cdcl$_W$-bj V*)
$\vee$ ($\exists$ *T. cdcl$_W$-merge-cp** S T* $\wedge$ *conflict T V*)
⟨*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
confl: *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* $S \Longrightarrow$ $cdcl_W$*-M-level-inv* $S \Longrightarrow$ *no-step* $cdcl_W$*-cp* S
⟨*proof*⟩

lemma *conflicting-not-true-rtranclp-*$cdcl_W$*-merge-cp-no-step-*$cdcl_W$*-bj*:
assumes
conflicting $S = None$ **and**
$cdcl_W$*-merge-cp*** S T
shows *no-step* $cdcl_W$*-bj* T
⟨*proof*⟩

lemma *conflicting-true-full-*$cdcl_W$*-merge-cp-iff-full-*$cdcl_W$*-s'-without-decode*:
assumes
confl: *conflicting* $S = None$ **and**
inv: $cdcl_W$*-all-struct-inv* S
shows
full $cdcl_W$*-merge-cp* S $V \longleftrightarrow$ *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
confl: *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-decide* S V
⟨*proof*⟩

inductive $cdcl_W$*-merge-stgy* **where**
fw-s-cp[*intro*]: *full1* $cdcl_W$*-merge-cp* S $T \Longrightarrow$ $cdcl_W$*-merge-stgy* S T |
fw-s-decide[*intro*]: *decide* S $T \Longrightarrow$ *no-step* $cdcl_W$*-merge-cp* $S \Longrightarrow$ *full* $cdcl_W$*-merge-cp* T U
$\Longrightarrow$ $cdcl_W$*-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 $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy\text{-}rtranclp\text{-}cdcl_W\text{-}merge$:
assumes fw: $cdcl_W\text{-}merge\text{-}stgy^{**}\ S\ T$
shows $cdcl_W\text{-}merge^{**}\ S\ T$
$\langle proof \rangle$

lemma $cdcl_W\text{-}merge\text{-}stgy\text{-}rtranclp\text{-}cdcl_W$:
$cdcl_W\text{-}merge\text{-}stgy\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
$\langle proof \rangle$

lemma $rtranclp\text{-}cdcl_W\text{-}merge\text{-}stgy\text{-}rtranclp\text{-}cdcl_W$:
$cdcl_W\text{-}merge\text{-}stgy^{**}\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
$\langle proof \rangle$

lemma $cdcl_W\text{-}merge\text{-}stgy\text{-}cases[consumes\ 1,\ case\text{-}names\ fw\text{-}s\text{-}cp\ fw\text{-}s\text{-}decide]$:
assumes
$cdcl_W\text{-}merge\text{-}stgy\ S\ U$
$full1\ cdcl_W\text{-}merge\text{-}cp\ S\ U \Longrightarrow P$
$\bigwedge T.\ decide\ S\ T \Longrightarrow no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S \Longrightarrow full\ cdcl_W\text{-}merge\text{-}cp\ T\ U \Longrightarrow P$
shows P
$\langle proof \rangle$

inductive $cdcl_W\text{-}s'\text{-}w :: 'st \Rightarrow 'st \Rightarrow bool$ **where**
$conflict'$: $full1\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S\ S' \Longrightarrow cdcl_W\text{-}s'\text{-}w\ S\ S'$ |
$decide'$: $decide\ S\ S' \Longrightarrow no\text{-}step\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S \Longrightarrow full\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S'\ S''$
$\Longrightarrow cdcl_W\text{-}s'\text{-}w\ S\ S''$

lemma $cdcl_W\text{-}s'\text{-}w\text{-}rtranclp\text{-}cdcl_W$:
$cdcl_W\text{-}s'\text{-}w\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
$\langle proof \rangle$

lemma $rtranclp\text{-}cdcl_W\text{-}s'\text{-}w\text{-}rtranclp\text{-}cdcl_W$:
$cdcl_W\text{-}s'\text{-}w^{**}\ S\ T \Longrightarrow cdcl_W^{**}\ S\ T$
$\langle proof \rangle$

lemma $no\text{-}step\text{-}cdcl_W\text{-}cp\text{-}no\text{-}step\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide$:
assumes $no\text{-}step\ cdcl_W\text{-}cp\ S$ **and** $conflicting\ S = None$ **and** inv: $cdcl_W\text{-}M\text{-}level\text{-}inv\ S$
shows $no\text{-}step\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S$
$\langle proof \rangle$

lemma $no\text{-}step\text{-}cdcl_W\text{-}cp\text{-}no\text{-}step\text{-}cdcl_W\text{-}merge\text{-}restart$:
assumes $no\text{-}step\ cdcl_W\text{-}cp\ S$ **and** $conflicting\ S = None$
shows $no\text{-}step\ cdcl_W\text{-}merge\text{-}cp\ S$
$\langle proof \rangle$
lemma $after\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp$:
assumes $cdcl_W\text{-}s'\text{-}without\text{-}decide\ S\ T$
shows $no\text{-}step\ cdcl_W\text{-}cp\ T$
$\langle proof \rangle$

lemma $no\text{-}step\text{-}cdcl_W\text{-}s'\text{-}without\text{-}decide\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp$:
$cdcl_W\text{-}all\text{-}struct\text{-}inv\ S \Longrightarrow no\text{-}step\ cdcl_W\text{-}s'\text{-}without\text{-}decide\ S \Longrightarrow no\text{-}step\ cdcl_W\text{-}cp\ S$
$\langle proof \rangle$

lemma $after\text{-}cdcl_W\text{-}s'\text{-}w\text{-}no\text{-}step\text{-}cdcl_W\text{-}cp$:
assumes $cdcl_W\text{-}s'\text{-}w\ S\ T$ **and** $cdcl_W\text{-}all\text{-}struct\text{-}inv\ S$

shows *no-step* $cdcl_W$*-cp* T
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-s'-w-no-step-*$cdcl_W$*-cp-or-eq*:
assumes $cdcl_W$*-s'-w*** S T **and** $cdcl_W$*-all-struct-inv* S
shows $S = T \lor$ *no-step* $cdcl_W$*-cp* T
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-stgy'-no-step-*$cdcl_W$*-cp-or-eq*:
assumes $cdcl_W$*-merge-stgy*** S T **and** *inv*: $cdcl_W$*-all-struct-inv* S
shows $S = T \lor$ *no-step* $cdcl_W$*-cp* T
⟨*proof*⟩

lemma *no-step-*$cdcl_W$*-s'-without-decide-no-step-*$cdcl_W$*-bj*:
assumes *no-step* $cdcl_W$*-s'-without-decide* S **and** *inv*: $cdcl_W$*-all-struct-inv* S
shows *no-step* $cdcl_W$*-bj* S
⟨*proof*⟩

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
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-s'-w-no-step-*$cdcl_W$*-bj-or-eq*:
assumes $cdcl_W$*-s'-w*** S T **and** $cdcl_W$*-all-struct-inv* S
shows $S = T \lor$ *no-step* $cdcl_W$*-bj* T
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-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 $\land$ *conflicting* $V = None$)
$\lor$ ($cdcl_W$*-merge-stgy*** R V $\land$ *conflicting* $V \neq None$ $\land$ *no-step* $cdcl_W$*-bj* V)
$\lor$ ($\exists S\ T\ U.$ $cdcl_W$*-merge-stgy*** R S $\land$ *no-step* $cdcl_W$*-merge-cp* S $\land$ *decide* S T
$\land$ $cdcl_W$*-merge-cp*** T U $\land$ *conflict* U V)
$\lor$ ($\exists S\ T.$ $cdcl_W$*-merge-stgy*** R S $\land$ *no-step* $cdcl_W$*-merge-cp* S $\land$ *decide* S T
$\land$ $cdcl_W$*-merge-cp*** T V
$\land$ *conflicting* $V = None$)
$\lor$ ($cdcl_W$*-merge-cp*** R V $\land$ *conflicting* $V = None$)
$\lor$ ($\exists U.$ $cdcl_W$*-merge-cp*** R U $\land$ *conflict* U V)
⟨*proof*⟩

lemma *decide-rtranclp-*$cdcl_W$*-s'-rtranclp-*$cdcl_W$*-s'*:
assumes
dec: *decide* S T **and**
$cdcl_W$*-s'*** T U **and**
n-s-S: *no-step* $cdcl_W$*-cp* S **and**
no-step $cdcl_W$*-cp* U
shows $cdcl_W$*-s'*** S U
⟨*proof*⟩

lemma *rtranclp-*$cdcl_W$*-merge-stgy-rtranclp-*$cdcl_W$*-s'*:
assumes

*$cdcl_W$-merge-stgy*** *R V* **and**
inv: *$cdcl_W$-all-struct-inv R*
shows *$cdcl_W$-s'*** *R V*
⟨*proof*⟩

lemma *rtranclp-$cdcl_W$-merge-stgy-distinct-mset-clauses*:
assumes *invR*: *$cdcl_W$-all-struct-inv R* **and**
st: *$cdcl_W$-merge-stgy*** *R S* **and**
dist: *distinct-mset* (*clauses R*) **and**
R: *trail R* = []
shows *distinct-mset* (*clauses S*)
⟨*proof*⟩

lemma *no-step-$cdcl_W$-s'-no-step-$cdcl_W$-merge-stgy*:
assumes
inv: *$cdcl_W$-all-struct-inv R* **and** *s'*: *no-step $cdcl_W$-s' R*
shows *no-step $cdcl_W$-merge-stgy R*
⟨*proof*⟩
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-tranclp-$cdcl_W$-merge*: *wf* $\{(T, S).$ *$cdcl_W$-all-struct-inv S* $\wedge$ *$cdcl_W$-merge^{++} S T*$\}$
⟨*proof*⟩

lemma *wf-$cdcl_W$-merge-cp*:
wf$\{(T, S).$ *$cdcl_W$-all-struct-inv S* $\wedge$ *$cdcl_W$-merge-cp S T*$\}$
⟨*proof*⟩

lemma *wf-$cdcl_W$-merge-stgy*:
wf$\{(T, S).$ *$cdcl_W$-all-struct-inv S* $\wedge$ *$cdcl_W$-merge-stgy S T*$\}$
⟨*proof*⟩

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 *rtranclp-$cdcl_W$-merge-cp-no-step-$cdcl_W$-bj*:
assumes *conflicting R = None* **and** *$cdcl_W$-merge-cp*** *R S*
shows *no-step $cdcl_W$-bj S*

⟨*proof*⟩

lemma *rtranclp-cdcl$_W$-merge-stgy-no-step-cdcl$_W$-bj*:
assumes *confl*: *conflicting R = None* **and** *cdcl$_W$-merge-stgy** R S*
shows *no-step cdcl$_W$-bj S*
⟨*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-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*) *ann-lits* **and**
hd-raw-trail :: *'st* ⇒ (*'v*, *nat*, *'cls*) *ann-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*) *ann-lit* ⇒ *'st* ⇒ *'st* **and**
tl-trail :: *'st* ⇒ *'st* **and**
add-init-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
add-learned-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
remove-cls :: *'cls* ⇒ *'st* ⇒ *'st* **and**
update-backtrack-lvl :: *nat* ⇒ *'st* ⇒ *'st* **and**
update-conflicting :: *'ccls option* ⇒ *'st* ⇒ *'st* **and**

init-state :: *'clss* ⇒ *'st* **and**
restart-state :: *'st* ⇒ *'st* +
fixes *f* :: *nat* ⇒ *nat*
assumes *f*: *unbounded f*
begin

The condition of the differences of cardinality has to be strict. Otherwise, you could be in a strange state, where nothing remains to do, but a restart is done. See the proof of well-foundedness.

inductive $cdcl_W$*-merge-with-restart* **where**
restart-step:
($cdcl_W$*-merge-stgy*^^(*card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)))) *S T*
⟹ *card* (*set-mset* (*learned-clss T*)) − *card* (*set-mset* (*learned-clss S*)) > *f n*
⟹ *restart T U* ⟹ $cdcl_W$*-merge-with-restart* (*S*, *n*) (*U*, *Suc n*) |
restart-full: *full1* $cdcl_W$*-merge-stgy S T* ⟹ $cdcl_W$*-merge-with-restart* (*S*, *n*) (*T*, *Suc n*)

lemma $cdcl_W$*-merge-with-restart S T* ⟹ $cdcl_W$*-merge-restart*** (*fst S*) (*fst T*)
⟨*proof*⟩

lemma $cdcl_W$*-merge-with-restart-rtranclp-*$cdcl_W$:
$cdcl_W$*-merge-with-restart S T* ⟹ $cdcl_W{}^{**}$ (*fst S*) (*fst T*)
⟨*proof*⟩

lemma $cdcl_W$*-merge-with-restart-increasing-number*:
$cdcl_W$*-merge-with-restart S T* ⟹ *snd T* = *1* + *snd S*
⟨*proof*⟩

lemma *full1* $cdcl_W$*-merge-stgy S T* ⟹ $cdcl_W$*-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*) ⊆ *simple-clss* (*atms-of-mm* (*init-clss S*))
⟨*proof*⟩

lemma $cdcl_W$*-merge-with-restart-init-clss*:
$cdcl_W$*-merge-with-restart S T* ⟹ $cdcl_W$*-M-level-inv* (*fst S*) ⟹
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*^^(*card* (*set-mset* (*learned-clss* T)) − *card* (*set-mset* (*learned-clss* S)))) S T $\Longrightarrow$
    card (*set-mset* (*learned-clss* T)) − *card* (*set-mset* (*learned-clss* S)) > f n $\Longrightarrow$
    restart T U $\Longrightarrow$
   $cdcl_W$*-with-restart* (S, n) (U, *Suc* n) |
restart-full: *full1* $cdcl_W$*-stgy* S T $\Longrightarrow$ $cdcl_W$*-with-restart* (S, n) (T, *Suc* n)

lemma $cdcl_W$*-with-restart-rtranclp-*$cdcl_W$:
  $cdcl_W$*-with-restart* S T $\Longrightarrow$ $cdcl_W{}^{**}$ (*fst* S) (*fst* T)
  ⟨*proof*⟩

lemma $cdcl_W$*-with-restart-increasing-number*:
  $cdcl_W$*-with-restart* S T $\Longrightarrow$ *snd* T = *1* + *snd* S
  ⟨*proof*⟩

lemma *full1* $cdcl_W$*-stgy* S T $\Longrightarrow$ $cdcl_W$*-with-restart* (S, n) (T, *Suc* n)
  ⟨*proof*⟩

lemma $cdcl_W$*-with-restart-init-clss*:
  $cdcl_W$*-with-restart* S T $\Longrightarrow$ $cdcl_W$*-M-level-inv* (*fst* S) $\Longrightarrow$ *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 $\exists k$::*nat*. $(2 \widehat{\ } (k - 1) \leq i \wedge i < 2 \widehat{\ } k - 1) \vee i = 2 \widehat{\ } k - 1$
⟨*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* ⇒ *nat* **where**
luby-sequence-core i =
(*if* $\exists k.\ i = 2\widehat{\ }k - 1$
then $2\widehat{\ }((SOME\ k.\ i = 2\widehat{\ }k - 1) - 1)$
else *luby-sequence-core* $(i - 2\widehat{\ }((SOME\ k.\ 2\widehat{\ }(k-1) \leq i \wedge i < 2\widehat{\ }k - 1) - 1) + 1)$)
⟨*proof*⟩
termination
⟨*proof*⟩

function *natlog2* :: *nat* ⇒ *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: $(2::nat) \widehat{\ } (k::nat) - 1 = 2 \widehat{\ } k' - 1$
shows $k' = k$
⟨*proof*⟩

lemma *luby-sequence-core-two-power-minus-one*:
luby-sequence-core $(2\widehat{\ }k - 1) = 2\widehat{\ }(k-1)$ (**is** $?L = ?K$)
⟨*proof*⟩

lemma *different-luby-decomposition-false*:
assumes
H: $2 \widehat{\ } (k - Suc\ 0) \leq i$ **and**
k': $i < 2 \widehat{\ } k' - Suc\ 0$ **and**
$k\text{-}k'$: $k > k'$
shows *False*
⟨*proof*⟩

lemma *luby-sequence-core-not-two-power-minus-one*:
assumes
$k\text{-}i$: $2 \widehat{\ } (k - 1) \leq i$ **and**
$i\text{-}k$: $i < 2\widehat{\ }k - 1$
shows *luby-sequence-core* i = *luby-sequence-core* $(i - 2 \widehat{\ } (k - 1) + 1)$
⟨*proof*⟩

lemma *unbounded-luby-sequence-core*: *unbounded luby-sequence-core*
⟨*proof*⟩

abbreviation *luby-sequence* :: *nat* ⇒ *nat* **where**
luby-sequence n ≡ *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* ≥ *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*) *ann-lits* **and**
hd-raw-trail :: *'st* ⇒ (*'v*, *nat*, *'cls*) *ann-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*) *ann-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 $cdcl_W$*-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-ann-lit-from-W* **where**
convert-ann-lit-from-W (*Propagated L -*) = *Propagated L* () |
convert-ann-lit-from-W (*Decided L -*) = *Decided L* ()

abbreviation *convert-trail-from-W* ::
(*'v*, *'lvl*, *'a*) *ann-lit list*
⇒ (*'v*, *unit*, *unit*) *ann-lit list* **where**
convert-trail-from-W ≡ *map convert-ann-lit-from-W*

lemma *lits-of-l-convert-trail-from-W*[*simp*]:
lits-of-l (*convert-trail-from-W M*) = *lits-of-l M*
⟨*proof*⟩

lemma *lit-of-convert-trail-from-W*[*simp*]:
lit-of (*convert-ann-lit-from-W L*) = *lit-of L*
⟨*proof*⟩

lemma *no-dup-convert-from-W*[*simp*]:
no-dup (*convert-trail-from-W M*) ⟷ *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-ann-lit-from-NOT*
:: (*'a*, *'e*, *'b*) *ann-lit* ⇒ (*'a*, *nat*, *'cls*) *ann-lit* **where**
convert-ann-lit-from-NOT (*Propagated L -*) = *Propagated L dummy-cls* |
convert-ann-lit-from-NOT (*Decided L -*) = *Decided L 0*

abbreviation *convert-trail-from-NOT* **where**
convert-trail-from-NOT ≡ *map convert-ann-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-ann-lit-from-W (*convert-ann-lit-from-NOT L*) = *L*
⟨*proof*⟩

abbreviation $trail_{NOT}$ **where**
$trail_{NOT}$ *S* ≡ *convert-trail-from-W* (*fst S*)

lemma *undefined-lit-convert-trail-from-W*[*iff*]:
undefined-lit (*convert-trail-from-W M*) *L* $\longleftrightarrow$ *undefined-lit M L*
⟨*proof*⟩

lemma *lit-of-convert-ann-lit-from-NOT*[*iff*]:
lit-of (*convert-ann-lit-from-NOT L*) = *lit-of L*
⟨*proof*⟩

sublocale $state_W$ ⊆ *dpll-state-ops*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
λ*S*. *convert-trail-from-W* (*trail S*)
raw-clauses
λ*L S*. *cons-trail* (*convert-ann-lit-from-NOT L*) *S*
λ*S*. *tl-trail S*
λ*C S*. *add-learned-cls C S*
λ*C S*. *remove-cls C S*

$\langle proof \rangle$

context $state_W$
begin
lemma *convert-ann-lit-from-W-convert-ann-lit-from-NOT*[*simp*]:
convert-ann-lit-from-W (*mmset-of-mlit* (*convert-ann-lit-from-NOT* L)) = L
$\langle proof \rangle$
end

sublocale $state_W \subseteq$ *dpll-state*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$
$\langle proof \rangle$

context $state_W$
begin
declare $state\text{-}simp_{NOT}$[*simp del*]

end

sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}$*-merge-bj-learn-ops*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$
λ- -. *True*
λ- $S.$ *raw-conflicting* S = *None*
$\lambda C\ C'\ L'\ S\ T.$ *backjump-l-cond* $C\ C'\ L'\ S\ T$
$\wedge$ *distinct-mset* (C' + {#L'#}) $\wedge$ $\neg$*tautology* (C' + {#L'#})
$\langle proof \rangle$

thm $cdcl_{NOT}$*-merge-bj-learn-proxy.axioms*
sublocale $conflict\text{-}driven\text{-}clause\text{-}learning_W \subseteq cdcl_{NOT}$*-merge-bj-learn-proxy*
mset-cls insert-cls remove-lit
mset-clss union-clss in-clss insert-clss remove-from-clss
$\lambda S.$ *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S.$ *cons-trail* (*convert-ann-lit-from-NOT* L) S
$\lambda S.$ *tl-trail* S
$\lambda C\ S.$ *add-learned-cls* $C\ S$
$\lambda C\ S.$ *remove-cls* $C\ S$

λ- -. *True*

λ- S. *raw-conflicting* S = *None*
backjump-l-cond
inv_{NOT}
⟨*proof*⟩

sublocale *conflict-driven-clause-learning*$_W$ ⊆ $cdcl_{NOT}$*-merge-bj-learn-proxy2* - - - - - - - -
λS. *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S$. *cons-trail* (*convert-ann-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 = *None* *backjump-l-cond* inv_{NOT}
⟨*proof*⟩

sublocale *conflict-driven-clause-learning*$_W$ ⊆ $cdcl_{NOT}$*-merge-bj-learn* - - - - - - - -
λS. *convert-trail-from-W* (*trail* S)
raw-clauses
$\lambda L\ S$. *cons-trail* (*convert-ann-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 = *None* inv_{NOT}
⟨*proof*⟩

context *conflict-driven-clause-learning*$_W$
begin

Notations are lost while proving locale inclusion:

notation *state-eq*$_{NOT}$ (**infix** $\sim_{NOT}$ *50*)

22.2 Additional Lemmas between NOT and W states

lemma *trail*$_W$*-eq-reduce-trail-to*$_{NOT}$*-eq*:
trail S = *trail* T ⟹ *trail* (*reduce-trail-to*$_{NOT}$ $F\ S$) = *trail* (*reduce-trail-to*$_{NOT}$ $F\ T$)
⟨*proof*⟩

lemma *trail-reduce-trail-to*$_{NOT}$*-add-learned-cls*:
no-dup (*trail* S) ⟹
trail (*reduce-trail-to*$_{NOT}$ M (*add-learned-cls* $D\ S$)) = *trail* (*reduce-trail-to*$_{NOT}$ $M\ S$)
⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$*-reduce-trail-convert*:
reduce-trail-to$_{NOT}$ $C\ S$ = *reduce-trail-to* (*convert-trail-from-NOT* C) S
⟨*proof*⟩

lemma *reduce-trail-to-map*[*simp*]:
reduce-trail-to (*map f M*) S = *reduce-trail-to* $M\ S$
⟨*proof*⟩

lemma *reduce-trail-to*$_{NOT}$*-map*[*simp*]:
reduce-trail-to$_{NOT}$ (*map f M*) S = *reduce-trail-to*$_{NOT}$ $M\ S$
⟨*proof*⟩

lemma *skip-or-resolve-state-change*:
assumes *skip-or-resolve*** *S T*
shows
$\exists M.$ *trail* $S = M$ @ *trail* $T \wedge (\forall m \in$ *set* $M. \neg$*is-decided* $m)$
clauses $S =$ *clauses* T
backtrack-lvl $S =$ *backtrack-lvl* T
⟨*proof*⟩

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*
$\vee$ (*no-step cdcl$_W$-merge T* $\wedge$ *conflicting* $T \neq$ *None*)
⟨*proof*⟩

abbreviation *cdcl$_{NOT}$-restart* **where**
cdcl$_{NOT}$-restart $\equiv$ *restart-ops.cdcl$_{NOT}$-raw-restart cdcl$_{NOT}$ restart*

lemma *cdcl$_W$-merge-restart-is-cdcl$_{NOT}$-merged-bj-learn-restart-no-step*:
assumes
inv: *cdcl$_W$-all-struct-inv S* **and**
cdcl$_W$:*cdcl$_W$-merge-restart S T*
shows *cdcl$_{NOT}$-restart*** *S T* $\vee$ (*no-step cdcl$_W$-merge T* $\wedge$ *conflicting* $T \neq$ *None*)
⟨*proof*⟩

abbreviation μ_{FW} :: *'st* $\Rightarrow$ *nat* **where**
μ_{FW} $S \equiv$ (*if no-step cdcl$_W$-merge S then 0 else 1+μ_{CDCL}'-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 μ_{FW} $T < \mu_{FW}$ S
⟨*proof*⟩

lemma *wf-cdcl$_W$-merge*: *wf* $\{(T, S)$. *cdcl$_W$-all-struct-inv S* $\wedge$ *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* $\longleftrightarrow$ *full cdcl$_W$-merge-stgy R V* (**is** *?s'* $\longleftrightarrow$ *?fw*)
⟨*proof*⟩

lemma *full-cdcl$_W$-stgy-full-cdcl$_W$-merge*:
assumes

$conflicting\ R = None$ **and**
inv: $cdcl_W$-all-$struct$-$inv\ R$
shows $full\ cdcl_W$-$stgy\ R\ V \longleftrightarrow full\ cdcl_W$-$merge$-$stgy\ R\ V$
⟨*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-clss* N)
shows (*conflicting* $S' = Some$ {#} ∧ *unsatisfiable* (*set-mset* (*mset-clss* N)))
∨ (*conflicting* $S' = None$ ∧ *trail* S' ⊨*asm mset-clss* N ∧ *satisfiable* (*set-mset* (*mset-clss* N)))
⟨*proof*⟩
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* $S \longleftrightarrow$
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
⟨*proof*⟩

lemma *rtranclp-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
⟨*proof*⟩

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 tho first literal included in C.

Then we can mark the conflict.

fun *cut-trail-wrt-clause* **where**
cut-trail-wrt-clause C [] $S = S$ |
cut-trail-wrt-clause C (*Decided* L - # M) S =
(*if* $-L \in\#$ C *then* S
else cut-trail-wrt-clause C M (*decr-bt-lvl* (*tl-trail* S))) |
cut-trail-wrt-clause C (*Propagated* L - # M) S =
(*if* $-L \in\#$ C *then* S
else cut-trail-wrt-clause C M (*tl-trail* S))

definition *add-new-clause-and-update* :: $'ccls \Rightarrow 'st \Rightarrow 'st$ **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
⟨*proof*⟩

lemma *learned-clss-cut-trail-wrt-clause*[*simp*]:
learned-clss (*cut-trail-wrt-clause* C M S) = *learned-clss* S
⟨*proof*⟩

lemma *conflicting-clss-cut-trail-wrt-clause*[*simp*]:
conflicting (*cut-trail-wrt-clause* C M S) = *conflicting* S
⟨*proof*⟩

lemma *trail-cut-trail-wrt-clause*:
$\exists M$. *trail* S = M @ *trail* (*cut-trail-wrt-clause* C (*trail* S) S)
⟨*proof*⟩

lemma *n-dup-no-dup-trail-cut-trail-wrt-clause*[*simp*]:
assumes *n-d*: *no-dup* (*trail* T)
shows *no-dup* (*trail* (*cut-trail-wrt-clause* C (*trail* T) T))
⟨*proof*⟩

lemma *cut-trail-wrt-clause-backtrack-lvl-length-decided*:
assumes
backtrack-lvl T = *length* (*get-all-levels-of-ann* (*trail* T))
shows
backtrack-lvl (*cut-trail-wrt-clause* C (*trail* T) T) =
length (*get-all-levels-of-ann* (*trail* (*cut-trail-wrt-clause* C (*trail* T) T)))
⟨*proof*⟩

lemma *cut-trail-wrt-clause-get-all-levels-of-ann*:
assumes *get-all-levels-of-ann* (*trail* T) = *rev* [*Suc 0*..<
Suc (*length* (*get-all-levels-of-ann* (*trail* T)))]
shows
get-all-levels-of-ann (*trail* ((*cut-trail-wrt-clause* C (*trail* T) T))) = *rev* [*Suc 0*..<
Suc (*length* (*get-all-levels-of-ann* (*trail* ((*cut-trail-wrt-clause* C (*trail* T) T)))))]
⟨*proof*⟩

lemma *cut-trail-wrt-clause-CNot-trail*:
  assumes *trail T* $\models$*as CNot C*
  shows
    (*trail* ((*cut-trail-wrt-clause C* (*trail T*) *T*))) $\models$*as CNot C*
  ⟨*proof*⟩

lemma *cut-trail-wrt-clause-hd-trail-in-or-empty-trail*:
  (($\forall$ *L* $\in$#*C*. $-L \notin$ *lits-of-l* (*trail T*)) $\wedge$ *trail* (*cut-trail-wrt-clause C* (*trail T*) *T*) = [])
    $\vee$ ($-$*lit-of* (*hd* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*))) $\in$# *C*
      $\wedge$ *length* (*trail* (*cut-trail-wrt-clause C* (*trail T*) *T*)) $\geq$ *1*)
  ⟨*proof*⟩

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-confl:
  trail S $\models$*asm init-clss S* $\Longrightarrow$ *distinct-mset* (*mset-ccls C*) $\Longrightarrow$ *conflicting S* = *None* $\Longrightarrow$
   trail S $\models$*as CNot* (*mset-ccls C*) $\Longrightarrow$
   full cdcl$_W$*-stgy*
     (*update-conflicting* (*Some C*)
       (*add-init-cls* (*cls-of-ccls C*) (*cut-trail-wrt-clause* (*mset-ccls C*) (*trail S*) *S*))) *T* $\Longrightarrow$
   incremental-cdcl$_W$ *S T* |
add-no-confl:
  trail S $\models$*asm init-clss S* $\Longrightarrow$ *distinct-mset* (*mset-ccls C*) $\Longrightarrow$ *conflicting S* = *None* $\Longrightarrow$
   $\neg$*trail S* $\models$*as CNot* (*mset-ccls C*) $\Longrightarrow$
   full cdcl$_W$*-stgy* (*add-init-cls* (*cls-of-ccls C*) *S*) *T* $\Longrightarrow$
   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*]: *trail T* $\models$*asm N* **and**
    tr-C[*simp*]: *trail T* $\models$*as CNot* (*mset-ccls C*) **and**
    [*simp*]: *distinct-mset* (*mset-ccls C*)
  shows *cdcl*$_W$*-all-struct-inv* (*add-new-clause-and-update C T*) (**is** *cdcl*$_W$*-all-struct-inv ?T'*)
⟨*proof*⟩

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 *rtranclp-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 $conflicting\ T = Some\ \{\#\} \wedge unsatisfiable\ (set\text{-}mset\ (init\text{-}clss\ T))$
$\vee\ conflicting\ T = None \wedge trail\ T \models asm\ init\text{-}clss\ T \wedge satisfiable\ (set\text{-}mset\ (init\text{-}clss\ T))$
⟨*proof*⟩

lemma *tranclp-incremental-correct*:
assumes
inc: *incremental-cdcl$_W$$^{++}$ S T* **and**
inv: *cdcl$_W$-all-struct-inv S* **and**
s-inv: *cdcl$_W$-stgy-invariant S*
shows $conflicting\ T = Some\ \{\#\} \wedge unsatisfiable\ (set\text{-}mset\ (init\text{-}clss\ T))$
$\vee\ conflicting\ T = None \wedge trail\ T \models asm\ init\text{-}clss\ T \wedge satisfiable\ (set\text{-}mset\ (init\text{-}clss\ T))$
⟨*proof*⟩

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 it the principle behind the two-watched literals, an implementation have to remember the candidates that have been found so far while updating the datstructure.

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 algoritm.

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*) *ann-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 *mmset-of-mlit'* :: (*'v*, *nat*, *'v twl-clause*) *ann-lit* ⇒ (*'v*, *nat*, *'v clause*) *ann-lit*
  where
mmset-of-mlit' (*Propagated L C*) = *Propagated L* (*mset* (*watched C* @ *unwatched C*)) |
mmset-of-mlit' (*Decided L i*) = *Decided L i*

lemma *lit-of-mmset-of-mlit'*[*simp*]: *lit-of* (*mmset-of-mlit' x*) = *lit-of x*
  ⟨*proof*⟩

lemma *lits-of-mmset-of-mlit'*[*simp*]: *lits-of* (*mmset-of-mlit'* ‘ *S*) = *lits-of S*
  ⟨*proof*⟩

abbreviation *trail* **where**
trail S ≡ *map mmset-of-mlit'* (*raw-trail S*)

abbreviation *clauses-of-l* **where**
  clauses-of-l ≡ λ*L*. *mset* (*map mset L*)

definition *raw-clause* :: *'v twl-clause* ⇒ *'v literal list* **where**
  raw-clause C ≡ *watched C* @ *unwatched C*

definition *clause* :: *'v twl-clause* ⇒ *'v clause* **where**
  clause C ≡ *mset* (*raw-clause C*)

lemma *clause-def-lambda*:
  clause = (λ*C*. *mset* (*raw-clause C*))
  ⟨*proof*⟩

abbreviation *raw-clss* :: *'v twl-state* ⇒ *'v clauses* **where**
  raw-clss S ≡ *mset* (*map clause* (*raw-init-clss S* @ *raw-learned-clss S*))

abbreviation *raw-clss-l* :: *'a twl-clause list* ⇒ *'a literal multiset multiset* **where**
  raw-clss-l C ≡ *mset* (*map clause C*)

interpretation *raw-cls*
clause
$\lambda L\ C.\ TWL\text{-}Clause\ (watched\ C)\ (L\ \#\ unwatched\ C)$
$\lambda L\ C.\ TWL\text{-}Clause\ []\ (remove1\ L\ (raw\text{-}clause\ C))$
⟨*proof*⟩

lemma *mset-map-clause-remove1-cond*:
$mset\ (map\ (\lambda x.\ mset\ (unwatched\ x) + mset\ (watched\ x))$
$(remove1\text{-}cond\ (\lambda D.\ clause\ D = clause\ a)\ Cs)) =$
$remove1\text{-}mset\ (clause\ a)\ (mset\ (map\ clause\ Cs))$
⟨*proof*⟩

interpretation *raw-clss*
clause

$\lambda L\ C.\ TWL\text{-}Clause\ (watched\ C)\ (L\ \#\ unwatched\ C)$
$\lambda L\ C.\ TWL\text{-}Clause\ []\ (remove1\ L\ (raw\text{-}clause\ C))$
raw-clss-l op @
$\lambda L\ C.\ L \in set\ C\ op\ \#\ \lambda C.\ remove1\text{-}cond\ (\lambda D.\ clause\ D = clause\ C)$
⟨*proof*⟩

lemma *ex-mset-unwatched-watched*:
$\exists\, a.\ mset\ (unwatched\ a) + mset\ (watched\ a) = E$
⟨*proof*⟩

thm *CDCL-Two-Watched-Literals.raw-cls-axioms*
interpretation *twl*: $state_W$*-ops*
clause

$\lambda L\ C.\ TWL\text{-}Clause\ (watched\ C)\ (L\ \#\ unwatched\ C)$
$\lambda L\ C.\ TWL\text{-}Clause\ []\ (remove1\ L\ (raw\text{-}clause\ C))$
raw-clss-l op @
$\lambda L\ C.\ L \in set\ C\ op\ \#\ \lambda C.\ remove1\text{-}cond\ (\lambda D.\ clause\ D = clause\ C)$

$mset\ \lambda xs\ ys.\ case\text{-}prod\ append\ (fold\ (\lambda x\ (ys,\ zs).\ (remove1\ x\ ys,\ x\ \#\ zs))\ xs\ (ys,\ []))$
op # remove1

$raw\text{-}clause\ \lambda C.\ TWL\text{-}Clause\ []\ C$
$trail\ \lambda S.\ hd\ (raw\text{-}trail\ S)$
raw-init-clss raw-learned-clss backtrack-lvl raw-conflicting

⟨*proof*⟩
declare *CDCL-Two-Watched-Literals.twl.mset-ccls-ccls-of-cls*[*simp del*]

lemma *mmset-of-mlit′-mmset-of-mlit*[*simp*]:
$twl.mmset\text{-}of\text{-}mlit\ L = mmset\text{-}of\text{-}mlit'\ L$
⟨*proof*⟩

definition
$candidates\text{-}propagate :: 'v\ twl\text{-}state \Rightarrow ('v\ literal \times 'v\ twl\text{-}clause)\ set$
where
$candidates\text{-}propagate\ S =$
$\{(L,\ C) \mid L\ C.$
$C \in set\ (twl.raw\text{-}clauses\ S)\ \wedge$
$set\ (watched\ C) - (uminus\ `\ lits\text{-}of\text{-}l\ (trail\ S)) = \{L\}\ \wedge$

undefined-lit (*raw-trail* S) L}

definition *candidates-conflict* :: *'v twl-state* $\Rightarrow$ *'v twl-clause set* **where**
candidates-conflict S =
{C. $C \in$ *set* (*twl.raw-clauses* S) $\wedge$
set (*watched* C) $\subseteq$ *uminus* ‘ *lits-of-l* (*raw-trail* S)}

primrec (*nonexhaustive*) *index* :: *'a list* $\Rightarrow$*'a* $\Rightarrow$ *nat* **where**
index (a # l) c = (*if* $a = c$ *then* 0 *else* 1+*index* l c)

lemma *index-nth*:
$a \in$ *set* $l \Longrightarrow l$! (*index* l a) = a
⟨*proof*⟩

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*, L does not get swapped with a watched literal L' such that $-L'$ is in the trail. This corresponds to the laziness of the data structure.

primrec *watched-decided-most-recently* :: (*'v*, *'lvl*, *'mark*) *ann-lit list* $\Rightarrow$
'v twl-clause $\Rightarrow$ *bool*
where
watched-decided-most-recently M (*TWL-Clause* W UW) $\longleftrightarrow$
($\forall L' \in$ *set* W. $\forall L \in$ *set* UW.
$-L' \in$ *lits-of-l* $M \longrightarrow -L \in$ *lits-of-l* $M \longrightarrow L \notin\#$ *mset* $W \longrightarrow$
index (*map lit-of* M) ($-L'$) $\leq$ *index* (*map lit-of* M) ($-L$))

The structural invariants states that there are at most two watched elements, that the watched literals are distinct, and that there are 2 watched literals if there are at least than two different literals in the full clauses.

primrec *struct-wf-twl-cls* :: *'v twl-clause* $\Rightarrow$ *bool* **where**
struct-wf-twl-cls (*TWL-Clause* W UW) $\longleftrightarrow$
distinct $W \wedge$ *length* $W \leq 2 \wedge$ (*length* $W < 2 \longrightarrow$ *set* $UW \subseteq$ *set* W)

Here are the invariant strictly related to the 2-WL data structure.

primrec *wf-twl-cls* :: (*'v*, *'lvl*, *'mark*) *ann-lit list* $\Rightarrow$ *'v twl-clause* $\Rightarrow$ *bool* **where**
wf-twl-cls M (*TWL-Clause* W UW) $\longleftrightarrow$
struct-wf-twl-cls (*TWL-Clause* W UW) $\wedge$
($\forall L \in$ *set* W. $-L \in$ *lits-of-l* $M \longrightarrow$ ($\forall L' \in$ *set* UW. $L' \notin$ *set* $W \longrightarrow -L' \in$ *lits-of-l* M)) $\wedge$
watched-decided-most-recently M (*TWL-Clause* W UW)

lemma *size-mset-2*: *size* $x1$ = 2 $\longleftrightarrow$ ($\exists$ a b. $x1$ = {#a, b#})
⟨*proof*⟩

lemma *distinct-mset-size-2*: *distinct-mset* {#a, b#} $\longleftrightarrow a \neq b$
⟨*proof*⟩

lemma *wf-twl-cls-annotation-independant*:
assumes M: *map lit-of* M = *map lit-of* M'
shows *wf-twl-cls* M (*TWL-Clause* W UW) $\longleftrightarrow$ *wf-twl-cls* M' (*TWL-Clause* W UW)
⟨*proof*⟩

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* ⇒ *bool* **where**
  wf-twl-state S ⟷
    (∀ *C* ∈ *set* (*twl.raw-clauses S*). *wf-twl-cls* (*raw-trail S*) *C*) ∧ *no-dup* (*raw-trail S*)

lemma *wf-candidates-propagate-sound*:
  assumes *wf*: *wf-twl-state S* **and**
    cand: (*L*, *C*) ∈ *candidates-propagate S*
  shows *raw-trail S* ⊨*as CNot* (*mset* (*removeAll L* (*raw-clause C*))) ∧ *undefined-lit* (*raw-trail S*) *L*
    (**is** *?Not* ∧ *?undef*)
⟨*proof*⟩

lemma *wf-candidates-propagate-complete*:
  assumes *wf*: *wf-twl-state S* **and**
    c-mem: *C* ∈ *set* (*twl.raw-clauses S*) **and**
    l-mem: *L* ∈ *set* (*raw-clause C*) **and**
    unsat: *trail S* ⊨*as CNot* (*mset-set* (*set* (*raw-clause C*) − {*L*})) **and**
    undef: *undefined-lit* (*raw-trail S*) *L*
  shows (*L*, *C*) ∈ *candidates-propagate S*
⟨*proof*⟩

lemma *wf-candidates-conflict-sound*:
  assumes *wf*: *wf-twl-state S* **and**
    cand: *C* ∈ *candidates-conflict S*
  shows *trail S* ⊨*as CNot* (*clause C*) ∧ *C* ∈ *set* (*twl.raw-clauses S*)
⟨*proof*⟩

lemma *wf-candidates-conflict-complete*:
  assumes *wf*: *wf-twl-state S* **and**
    c-mem: *C* ∈ *set* (*twl.raw-clauses S*) **and**
    unsat: *trail S* ⊨*as CNot* (*clause C*)
  shows *C* ∈ *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* ⇒ *'v twl-clause set* **where**
candidates-conflict-twl S ≡ *candidates-conflict* (*rough-state-of-twl S*)

abbreviation *candidates-propagate-twl* :: *'v wf-twl* ⇒ (*'v literal* × *'v twl-clause*) *set* **where**
candidates-propagate-twl S ≡ *candidates-propagate* (*rough-state-of-twl S*)

abbreviation *raw-trail-twl* :: *'a wf-twl* ⇒ (*'a*, *nat*, *'a twl-clause*) *ann-lit list* **where**
raw-trail-twl S ≡ *raw-trail* (*rough-state-of-twl S*)

abbreviation *trail-twl* :: *'a wf-twl* ⇒ (*'a*, *nat*, *'a literal multiset*) *ann-lit list* **where**
trail-twl S ≡ *trail* (*rough-state-of-twl S*)

abbreviation *raw-clauses-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-clauses-twl S ≡ *twl.raw-clauses* (*rough-state-of-twl S*)

abbreviation *raw-init-clss-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-init-clss-twl S ≡ *raw-init-clss* (*rough-state-of-twl S*)

abbreviation *raw-learned-clss-twl* :: *'a wf-twl* ⇒ *'a twl-clause list* **where**
raw-learned-clss-twl S ≡ *raw-learned-clss* (*rough-state-of-twl S*)

abbreviation *backtrack-lvl-twl* **where**
backtrack-lvl-twl S ≡ *backtrack-lvl* (*rough-state-of-twl S*)

abbreviation *raw-conflicting-twl* **where**
raw-conflicting-twl S ≡ *raw-conflicting* (*rough-state-of-twl S*)

lemma *wf-candidates-twl-conflict-complete*:
  assumes
    c-mem: *C* ∈ *set* (*raw-clauses-twl S*) **and**
    unsat: *trail-twl S* ⊨*as CNot* (*clause C*)
  shows *C* ∈ *candidates-conflict-twl S*
  ⟨*proof*⟩

abbreviation *update-backtrack-lvl* **where**
  update-backtrack-lvl k S ≡
   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 ≡
    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* ⇒ *'v literal list* ⇒ *'v twl-clause* **and**
    rewatch :: *'v literal* ⇒ *'v twl-state* ⇒
      'v twl-clause ⇒ *'v twl-clause* **and**

restart-learned :: *'v twl-state* ⇒ *'v twl-clause list*
assumes
clause-watch: *no-dup* (*raw-trail* *S*) ⟹ *clause* (*watch* *S* *C*) = *mset* *C* **and**
wf-watch: *no-dup* (*raw-trail* *S*) ⟹ *wf-twl-cls* (*raw-trail* *S*) (*watch* *S* *C*) **and**
clause-rewatch: *clause* (*rewatch* *L'* *S* *C'*) = *clause* *C'* **and**
wf-rewatch:
no-dup (*raw-trail* *S*) ⟹ *undefined-lit* (*raw-trail* *S*) (*lit-of* *L*) ⟹
wf-twl-cls (*raw-trail* *S*) *C'* ⟹
wf-twl-cls (*L* # *raw-trail* *S*) (*rewatch* (*lit-of* *L*) *S* *C'*)
and
restart-learned: *mset* (*restart-learned* *S*) ⊆# *mset* (*raw-learned-clss* *S*) — We need *mset* and not *set* to take care of duplicates.
begin

definition
cons-trail :: (*'v*, *nat*, *'v twl-clause*) *ann-lit* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
cons-trail *L* *S* =
TWL-State (*L* # *raw-trail* *S*) (*map* (*rewatch* (*lit-of* *L*) *S*) (*raw-init-clss* *S*))
(*map* (*rewatch* (*lit-of* *L*) *S*) (*raw-learned-clss* *S*)) (*backtrack-lvl* *S*) (*raw-conflicting* *S*)

definition
add-init-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
add-init-cls *C* *S* =
TWL-State (*raw-trail* *S*) (*watch* *S* *C* # *raw-init-clss* *S*) (*raw-learned-clss* *S*) (*backtrack-lvl* *S*)
(*raw-conflicting* *S*)

definition
add-learned-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
add-learned-cls *C* *S* =
TWL-State (*raw-trail* *S*) (*raw-init-clss* *S*) (*watch* *S* *C* # *raw-learned-clss* *S*) (*backtrack-lvl* *S*)
(*raw-conflicting* *S*)

definition
remove-cls :: *'v literal list* ⇒ *'v twl-state* ⇒ *'v twl-state*
where
remove-cls *C* *S* =
TWL-State (*raw-trail* *S*)
(*removeAll-cond* (λ*D*. *clause* *D* = *mset* *C*) (*raw-init-clss* *S*))
(*removeAll-cond* (λ*D*. *clause* *D* = *mset* *C*) (*raw-learned-clss* *S*))
(*backtrack-lvl* *S*)
(*raw-conflicting* *S*)

definition *init-state* :: *'v literal list list* ⇒ *'v twl-state* **where**
init-state *N* = *fold* *add-init-cls* *N* (*TWL-State* [] [] [] *0* *None*)

lemma *unchanged-fold-add-init-cls*:
raw-trail (*fold* *add-init-cls* *Cs* (*TWL-State* *M* *N* *U* *k* *C*)) = *M*
raw-learned-clss (*fold* *add-init-cls* *Cs* (*TWL-State* *M* *N* *U* *k* *C*)) = *U*
backtrack-lvl (*fold* *add-init-cls* *Cs* (*TWL-State* *M* *N* *U* *k* *C*)) = *k*
raw-conflicting (*fold* *add-init-cls* *Cs* (*TWL-State* *M* *N* *U* *k* *C*)) = *C*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *clauses-init-fold-add-init*:
  no-dup $M \Longrightarrow$
   twl.init-clss (*fold add-init-cls Cs* (*TWL-State M N U k C*)) =
   clauses-of-l Cs + *raw-clss-l N*
  ⟨*proof*⟩

lemma *init-clss-init-state*[*simp*]: *twl.init-clss* (*init-state* N) = *clauses-of-l N*
  ⟨*proof*⟩

definition *restart′* **where**
  restart′ S = *TWL-State* [] (*raw-init-clss S*) (*restart-learned S*) *0 None*

end

24.1.4 Instanciation of the previous locale

definition *watch-nat* :: *′v twl-state* ⇒ *′v literal list* ⇒ *′v twl-clause* **where**
  watch-nat S C =
   (*let*
      C′ = *remdups C*;
      neg-not-assigned = *filter* (λ*L*. −*L* ∉ *lits-of-l* (*raw-trail S*)) *C′*;
      neg-assigned-sorted-by-trail = *filter* (λ*L*. *L* ∈ *set C*) (*map* (λ*L*. −*lit-of L*) (*raw-trail S*));
      W = *take 2* (*neg-not-assigned* @ *neg-assigned-sorted-by-trail*);
      UW = *foldr remove1 W C*
    in TWL-Clause W UW)

lemma *list-cases2*:
  fixes *l* :: *′a list*
  assumes
    $l = [] \Longrightarrow P$ **and**
    $\bigwedge x.\ l = [x] \Longrightarrow P$ **and**
    $\bigwedge x\ y\ xs.\ l = x \# y \# xs \Longrightarrow P$
  shows P
  ⟨*proof*⟩

lemma *filter-in-list-prop-verifiedD*:
  assumes [*L*←*P* . *Q L*] = *l*
  shows ∀ *x* ∈ *set l*. *x* ∈ *set P* ∧ *Q x*
  ⟨*proof*⟩

lemma *no-dup-filter-diff*:
  assumes *n-d*: *no-dup M* **and** *H*: [*L*←*map* (λ*L*. − *lit-of L*) *M*. *L* ∈ *set C*] = *l*
  shows *distinct l*
  ⟨*proof*⟩

lemma *watch-nat-lists-disjointD*:
  assumes
    l: [*L*←*remdups C*. − *L* ∉ *lits-of-l* (*raw-trail S*)] = *l* **and**
    l′: [*L*←*map* (λ*L*. − *lit-of L*) (*raw-trail S*) . *L* ∈ *set C*] = *l′*

shows $\forall x \in set\ l.\ \forall y \in set\ l'.\ x \neq y$
$\langle proof \rangle$

lemma *watch-nat-list-cases-witness*[*consumes 2*, *case-names nil-nil nil-single nil-other single-nil single-other other*]:
fixes
C :: $'v$ *literal list* **and**
S :: $'v$ *twl-state*
defines
$xs \equiv [L \leftarrow remdups\ C.\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
assumes
n-d: $no\text{-}dup\ (raw\text{-}trail\ S)$ **and**
nil-nil: $xs = [] \Longrightarrow ys = [] \Longrightarrow P$ **and**
nil-single:
$\bigwedge a.\ xs = [] \Longrightarrow ys = [a] \Longrightarrow a \in set\ C \Longrightarrow P$ **and**
nil-other: $\bigwedge a\ b\ ys'.\ xs = [] \Longrightarrow ys = a\ \#\ b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
single-nil: $\bigwedge a.\ xs = [a] \Longrightarrow ys = [] \Longrightarrow P$ **and**
single-other: $\bigwedge a\ b\ ys'.\ xs = [a] \Longrightarrow ys = b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
other: $\bigwedge a\ b\ xs'.\ xs = a\ \#\ b\ \#\ xs' \Longrightarrow a \neq b \Longrightarrow P$
shows P
$\langle proof \rangle$

lemma *watch-nat-list-cases* [*consumes 1*, *case-names nil-nil nil-single nil-other single-nil single-other other*]:
fixes
C :: $'v$ *literal list* **and**
S :: $'v$ *twl-state*
defines
$xs \equiv [L \leftarrow remdups\ C\ .\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
assumes
n-d: $no\text{-}dup\ (raw\text{-}trail\ S)$ **and**
nil-nil: $xs = [] \Longrightarrow ys = [] \Longrightarrow P$ **and**
nil-single:
$\bigwedge a.\ xs = [] \Longrightarrow ys = [a] \Longrightarrow a \in set\ C \Longrightarrow P$ **and**
nil-other: $\bigwedge a\ b\ ys'.\ xs = [] \Longrightarrow ys = a\ \#\ b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
single-nil: $\bigwedge a.\ xs = [a] \Longrightarrow ys = [] \Longrightarrow P$ **and**
single-other: $\bigwedge a\ b\ ys'.\ xs = [a] \Longrightarrow ys = b\ \#\ ys' \Longrightarrow a \neq b \Longrightarrow P$ **and**
other: $\bigwedge a\ b\ xs'.\ xs = a\ \#\ b\ \#\ xs' \Longrightarrow a \neq b \Longrightarrow P$
shows P
$\langle proof \rangle$

lemma *watch-nat-lists-set-union-witness*:
fixes
C :: $'v$ *literal list* **and**
S :: $'v$ *twl-state*
defines
$xs \equiv [L \leftarrow remdups\ C.\ -\ L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S)]$ **and**
$ys \equiv [L \leftarrow map\ (\lambda L.\ -\ lit\text{-}of\ L)\ (raw\text{-}trail\ S)\ .\ L \in set\ C]$
assumes *n-d*: $no\text{-}dup\ (raw\text{-}trail\ S)$
shows $set\ C = set\ xs \cup set\ ys$
$\langle proof \rangle$

lemma *mset-intersection-inclusion*: $A + (B - A) = B \longleftrightarrow A \subseteq\#\ B$

$\langle proof \rangle$

lemma *clause-watch-nat*:
assumes *no-dup* (*raw-trail* S)
shows *clause* (*watch-nat* S C) = *mset* C
$\langle proof \rangle$

lemma *index-uminus-index-map-uminus*:
$-a \in set\ L \Longrightarrow index\ L\ (-a) = index\ (map\ uminus\ L)\ (a::'a\ literal)$
$\langle proof \rangle$

lemma *index-filter*:
$a \in set\ L \Longrightarrow b \in set\ L \Longrightarrow P\ a \Longrightarrow P\ b \Longrightarrow$
$index\ L\ a \leq index\ L\ b \longleftrightarrow index\ (filter\ P\ L)\ a \leq index\ (filter\ P\ L)\ b$
$\langle proof \rangle$

lemma *foldr-remove1-W-Nil*[*simp*]: *foldr remove1* W [] = []
$\langle proof \rangle$

lemma *image-lit-of-mmset-of-mlit'*[*simp*]:
lit-of ` *mmset-of-mlit'* ` A = *lit-of* ` A
$\langle proof \rangle$

lemma *distinct-filter-eq*:
assumes *distinct xs*
shows $[L\leftarrow xs.\ L = a] = (if\ a \in set\ xs\ then\ [a]\ else\ [])$
$\langle proof \rangle$

lemma *no-dup-distinct-map-uminus-lit-of*:
$no\text{-}dup\ xs \Longrightarrow distinct\ (map\ (\lambda L.\ -\ lit\text{-}of\ L)\ xs)$
$\langle proof \rangle$

lemma *wf-watch-witness*:
fixes C :: *'v literal list* **and**
S :: *'v twl-state*
defines
ass: $neg\text{-}not\text{-}assigned \equiv filter\ (\lambda L.\ -L \notin lits\text{-}of\text{-}l\ (raw\text{-}trail\ S))\ (remdups\ C)$ **and**
tr: $neg\text{-}assigned\text{-}sorted\text{-}by\text{-}trail \equiv filter\ (\lambda L.\ L \in set\ C)\ (map\ (\lambda L.\ -lit\text{-}of\ L)\ (raw\text{-}trail\ S))$
defines
W: $W \equiv take\ 2\ (neg\text{-}not\text{-}assigned\ @\ neg\text{-}assigned\text{-}sorted\text{-}by\text{-}trail)$
assumes
n-d[*simp*]: *no-dup* (*raw-trail* S)
shows *wf-twl-cls* (*raw-trail* S) (*TWL-Clause* W (*foldr remove1* W C))
$\langle proof \rangle$

lemma *wf-watch-nat*: $no\text{-}dup\ (raw\text{-}trail\ S) \Longrightarrow wf\text{-}twl\text{-}cls\ (raw\text{-}trail\ S)\ (watch\text{-}nat\ S\ C)$
$\langle proof \rangle$

definition
rewatch-nat ::
$'v\ literal \Rightarrow 'v\ twl\text{-}state \Rightarrow 'v\ twl\text{-}clause \Rightarrow 'v\ twl\text{-}clause$
where
rewatch-nat L S C =
$(if\ -\ L \in set\ (watched\ C)\ then$
$case\ filter\ (\lambda L'.\ L' \notin set\ (watched\ C) \wedge -\ L' \notin insert\ L\ (lits\text{-}of\text{-}l\ (trail\ S)))$

      $(unwatched\ C)$ *of*
     $[] \Rightarrow C$
    $|\ L' \ \#\ \text{-} \Rightarrow$
      $TWL\text{-}Clause\ (L' \ \#\ remove1\ (-L)\ (watched\ C))\ (-L\ \#\ remove1\ L'\ (unwatched\ C))$
  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 $clause\ (rewatch\text{-}nat\ L\ S\ C) = clause\ C$
  ⟨*proof*⟩

lemma *filter-sorted-list-of-multiset-Nil*:
  $[x \leftarrow sorted\text{-}list\text{-}of\text{-}multiset\ M.\ p\ x] = [] \longleftrightarrow (\forall x \in\# M.\ \neg\ p\ x)$
  ⟨*proof*⟩

lemma *filter-sorted-list-of-multiset-ConsD*:
  $[x \leftarrow sorted\text{-}list\text{-}of\text{-}multiset\ M.\ p\ x] = x\ \#\ xs \Longrightarrow p\ x$
  ⟨*proof*⟩

lemma *mset-minus-single-eq-mempty*:
  $a - \{\#b\#\} = \{\#\} \longleftrightarrow a = \{\#b\#\} \lor a = \{\#\}$
  ⟨*proof*⟩

lemma *size-mset-le-2-cases*:
  assumes $size\ W \le 2$
  shows $W = \{\#\} \lor (\exists a.\ W = \{\#a\#\}) \lor (\exists a\ b.\ W = \{\#a,b\#\})$
  ⟨*proof*⟩

lemma *filter-sorted-list-of-multiset-eqD*:
  assumes $[x \leftarrow sorted\text{-}list\text{-}of\text{-}multiset\ A.\ p\ x] = x\ \#\ xs$ (**is** *?comp* = -)
  shows $x \in\# A$
⟨*proof*⟩

lemma *clause-rewatch-witness'*:
  assumes
    wf: *wf-twl-cls* (*raw-trail* S) C **and**
    undef: *undefined-lit* (*raw-trail* S) (*lit-of* L)
  shows *wf-twl-cls* (L # *raw-trail* S) (*rewatch-nat* (*lit-of* L) S C)
⟨*proof*⟩

interpretation *twl*: *abstract-twl watch-nat rewatch-nat raw-learned-clss*
  ⟨*proof*⟩

interpretation *twl2*: *abstract-twl watch-nat rewatch-nat* λ-. []
  ⟨*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 datastructure.

context *abstract-twl*
begin

Direct Interpretation **lemma** *mset-map-removeAll-cond*:
  mset (*map clause*
    (*removeAll-cond* (λD. *clause* D = *clause* C) N))
  = *mset* (*removeAll* (*clause* C) (*map clause* N))
  ⟨*proof*⟩

lemma *mset-raw-init-clss-init-state*:
  mset (*map clause* (*raw-init-clss* (*init-state* (*map raw-clause* N))))
  = *mset* (*map clause* N)
  ⟨*proof*⟩

interpretation *rough-cdcl*: *state*$_W$
  clause

  λL C. *TWL-Clause* (*watched* C) (L # *unwatched* C)
  λL C. *TWL-Clause* [] (*remove1* L (*raw-clause* C))
  raw-clss-l op @
  λL C. $L \in$ *set* C *op* # λC. *remove1-cond* (λD. *clause* D = *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*$_W$
  clause

  λL C. *TWL-Clause* (*watched* C) (L # *unwatched* C)
  λL C. *TWL-Clause* [] (*remove1* L (*raw-clause* C))
  raw-clss-l op @
  λL C. $L \in$ *set* C *op* # λC. *remove1-cond* (λD. *clause* D = *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 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*⟩

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*) *ann-lit* ⇒ *'v wf-twl* ⇒ *'v wf-twl*
where
cons-trail-twl L S ≡ *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)
⟨*proof*⟩

lemma *rough-state-of-twl-cons-trail*:
undefined-lit (*raw-trail-twl* S) (*lit-of* L) ⟹
rough-state-of-twl (*cons-trail-twl* L S) = *cons-trail* L (*rough-state-of-twl* S)
⟨*proof*⟩

abbreviation *add-init-cls-twl* **where**
add-init-cls-twl C S ≡ *twl-of-rough-state* (*add-init-cls* C (*rough-state-of-twl* S))

lemma *wf-twl-add-init-cls*: *wf-twl-state* S ⟹ *wf-twl-state* (*add-init-cls* L S)
⟨*proof*⟩

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)
⟨*proof*⟩

abbreviation *add-learned-cls-twl* **where**
add-learned-cls-twl C S ≡ *twl-of-rough-state* (*add-learned-cls* C (*rough-state-of-twl* S))

lemma *wf-twl-add-learned-cls*: *wf-twl-state* S ⟹ *wf-twl-state* (*add-learned-cls* L S)
⟨*proof*⟩

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)
⟨*proof*⟩

abbreviation *remove-cls-twl* **where**
remove-cls-twl C S ≡ *twl-of-rough-state* (*remove-cls* C (*rough-state-of-twl* S))

lemma *set-removeAll-condD*: $x \in$ *set* (*removeAll-cond* f *xs*) ⟹ $x \in$ *set xs*
⟨*proof*⟩

lemma *wf-twl-remove-cls*: *wf-twl-state* S ⟹ *wf-twl-state* (*remove-cls* L S)
⟨*proof*⟩

lemma *rough-state-of-twl-remove-cls*:
rough-state-of-twl (*remove-cls-twl L S*) = *remove-cls L* (*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *init-state-twl* **where**
init-state-twl N ≡ *twl-of-rough-state* (*init-state N*)

lemma *wf-twl-state-wf-twl-state-fold-add-init-cls*:
assumes *wf-twl-state S*
shows *wf-twl-state* (*fold add-init-cls N S*)
⟨*proof*⟩

lemma *wf-twl-state-epsilon-state*[*simp*]:
wf-twl-state (*TWL-State* [] [] [] *0 None*)
⟨*proof*⟩

lemma *wf-twl-init-state*: *wf-twl-state* (*init-state N*)
⟨*proof*⟩

lemma *rough-state-of-twl-init-state*:
rough-state-of-twl (*init-state-twl N*) = *init-state N*
⟨*proof*⟩

abbreviation *tl-trail-twl* **where**
tl-trail-twl S ≡ *twl-of-rough-state* (*tl-trail* (*rough-state-of-twl S*))

lemma *wf-twl-state-tl-trail*: *wf-twl-state S* ⟹ *wf-twl-state* (*tl-trail S*)
⟨*proof*⟩

lemma *rough-state-of-twl-tl-trail*:
rough-state-of-twl (*tl-trail-twl S*) = *tl-trail* (*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *update-backtrack-lvl-twl* **where**
update-backtrack-lvl-twl k S ≡ *twl-of-rough-state* (*update-backtrack-lvl k* (*rough-state-of-twl S*))

lemma *wf-twl-state-update-backtrack-lvl*:
wf-twl-state S ⟹ *wf-twl-state* (*update-backtrack-lvl k S*)
⟨*proof*⟩

lemma *rough-state-of-twl-update-backtrack-lvl*:
rough-state-of-twl (*update-backtrack-lvl-twl k S*) = *update-backtrack-lvl k*
(*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *update-conflicting-twl* **where**
update-conflicting-twl k S ≡ *twl-of-rough-state* (*update-conflicting k* (*rough-state-of-twl S*))

lemma *wf-twl-state-update-conflicting*:
wf-twl-state S ⟹ *wf-twl-state* (*update-conflicting k S*)
⟨*proof*⟩

lemma *rough-state-of-twl-update-conflicting*:
rough-state-of-twl (*update-conflicting-twl k S*) = *update-conflicting k*

(*rough-state-of-twl S*)
⟨*proof*⟩

abbreviation *raw-clauses-twl* **where**
raw-clauses-twl S ≡ *twl.raw-clauses* (*rough-state-of-twl S*)

abbreviation *restart-twl* **where**
restart-twl S ≡ *twl-of-rough-state* (*restart′* (*rough-state-of-twl S*))
lemma *mset-union-mset-setD*:
mset A ⊆# *mset B* ⟹ *set A* ⊆ *set B*
⟨*proof*⟩

lemma *wf-wf-restart′*: *wf-twl-state S* ⟹ *wf-twl-state* (*restart′ S*)
⟨*proof*⟩

lemma *rough-state-of-twl-restart-twl*:
rough-state-of-twl (*restart-twl S*) = *restart′* (*rough-state-of-twl S*)
⟨*proof*⟩

lemma *undefined-lit-trail-twl-raw-trail*[*iff*]:
undefined-lit (*trail-twl S*) *L* ⟷ *undefined-lit* (*raw-trail-twl S*) *L*
⟨*proof*⟩

sublocale *wf-twl*: *conflict-driven-clause-learning*$_W$
clause

λ*L C*. *TWL-Clause* (*watched C*) (*L* # *unwatched C*)
λ*L C*. *TWL-Clause* [] (*remove1 L* (*raw-clause C*))
raw-clss-l op @
λ*L C*. *L* ∈ *set C op* # λ*C*. *remove1-cond* (λ*D*. *clause D* = *clause C*)

mset λ*xs ys*. *case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, []))
op # *remove1*

λ*C*. *raw-clause C* λ*C*. *TWL-Clause* [] *C*
trail-twl λ*S*. *hd* (*raw-trail-twl S*)
raw-init-clss-twl
raw-learned-clss-twl
backtrack-lvl-twl
raw-conflicting-twl
cons-trail-twl
tl-trail-twl
λ*C*. *add-init-cls-twl* (*raw-clause C*)
λ*C*. *add-learned-cls-twl* (*raw-clause C*)
λ*C*. *remove-cls-twl* (*raw-clause C*)
update-backtrack-lvl-twl
update-conflicting-twl
λ*N*. *init-state-twl* (*map raw-clause N*)
restart-twl
⟨*proof*⟩

declare *local.rough-cdcl.mset-ccls-ccls-of-cls*[*simp del*]
abbreviation *state-eq-twl* (**infix** ∼*TWL 51*) **where**
state-eq-twl S S′ ≡ *rough-cdcl.state-eq* (*rough-state-of-twl S*) (*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-twl* (**infix** $\sim$ *51*)

Alternative Definition of CDCL using the candidates of 2-WL

inductive *propagate-twl* :: *'v wf-twl* $\Rightarrow$ *'v wf-twl* $\Rightarrow$ *bool* **where**
propagate-twl-rule: $(L, C) \in$ *candidates-propagate-twl S* $\Longrightarrow$
  $S' \sim$ *cons-trail-twl* (*Propagated L C*) *S* $\Longrightarrow$
  raw-conflicting-twl S = None $\Longrightarrow$
  propagate-twl S S'

inductive-cases *propagate-twlE*: *propagate-twl S T*

lemma *distinct-filter-eq-if*:
  distinct C $\Longrightarrow$ *length* (*filter* (*op = L*) *C*) = (*if L* $\in$ *set C then 1 else 0*)
  ⟨*proof*⟩

lemma *distinct-mset-remove1-All*:
  distinct-mset C $\Longrightarrow$ *remove1-mset L C = removeAll-mset L C*
  ⟨*proof*⟩

lemma *propagate-twl-iff-propagate*:
  assumes *inv*: *wf-twl.cdcl*$_W$*-all-struct-inv S*
  shows *wf-twl.propagate S T* $\longleftrightarrow$ *propagate-twl S T* (**is** *?P* $\longleftrightarrow$ *?T*)
⟨*proof*⟩

no-notation *twl.state-eq-twl* (**infix** $\sim$*TWL 51*)

inductive *conflict-twl* **where**
conflict-twl-rule:
$C \in$ *candidates-conflict-twl S* $\Longrightarrow$
  $S' \sim$ *update-conflicting-twl* (*Some* (*raw-clause C*)) *S* $\Longrightarrow$
  raw-conflicting-twl S = None $\Longrightarrow$
  conflict-twl S S'

inductive-cases *conflict-twlE*: *conflict-twl S T*

lemma *conflict-twl-iff-conflict*:
  shows *wf-twl.conflict S T* $\longleftrightarrow$ *conflict-twl S T* (**is** *?C* $\longleftrightarrow$ *?T*)
⟨*proof*⟩

inductive *cdcl*$_W$*-twl* :: *'v wf-twl* $\Rightarrow$ *'v wf-twl* $\Rightarrow$ *bool* **for** *S* :: *'v wf-twl* **where**
propagate: *propagate-twl S S'* $\Longrightarrow$ *cdcl*$_W$*-twl S S'* |
conflict: *conflict-twl S S'* $\Longrightarrow$ *cdcl*$_W$*-twl S S'* |
other: *wf-twl.cdcl*$_W$*-o S S'* $\Longrightarrow$ *cdcl*$_W$*-twl S S'*|
rf: *wf-twl.cdcl*$_W$*-rf S S'* $\Longrightarrow$ *cdcl*$_W$*-twl S S'*

lemma *cdcl*$_W$*-twl-iff-cdcl*$_W$:
  assumes *wf-twl.cdcl*$_W$*-all-struct-inv S*
  shows *cdcl*$_W$*-twl S T* $\longleftrightarrow$ *wf-twl.cdcl*$_W$ *S T*
  ⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-twl-all-struct-inv-inv*:
  assumes *cdcl*$_W$*-twl*** *S T* **and** *wf-twl.cdcl*$_W$*-all-struct-inv S*

shows *wf-twl.cdcl*$_W$*-all-struct-inv T*
⟨*proof*⟩

lemma *rtranclp-cdcl*$_W$*-twl-iff-rtranclp-cdcl*$_W$:
assumes *wf-twl.cdcl*$_W$*-all-struct-inv S*
shows *cdcl*$_W$*-twl*** *S T* $\longleftrightarrow$ *wf-twl.cdcl*$_W$** *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* ⇒ *'a clause* ⇒ *'a clauses* **where**
clss-lt N C = $\{D \in N.\ D\ \#{\subset}\#\ C\}$

notation (*latex* **output**)
clss-lt (-<^bsup>-<^esup>)

locale *selection* =
fixes *S* :: *'a clause* ⇒ *'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* ⇒ *'a clause*
begin

context
  fixes *N* :: *'a clause set*
begin

We do not create an equivalent of δ, but we directly defined N_C by inlining the definition.

function
  production :: *'a clause* ⇒ *'a interp*
where
  production C =
   {*A*. *C* ∈ *N* ∧ *C* ≠ {#} ∧ *Max* (*set-mset C*) = *Pos A* ∧ *count C* (*Pos A*) ≤ *1*
     ∧ ¬ (⋃*D* ∈ {*D*. *D* #⊂# *C*}. *production D*) ⊨*h C* ∧ *S C* = {#}}
  ⟨*proof*⟩
termination ⟨*proof*⟩

declare *production.simps*[*simp del*]

definition *interp* :: *'a clause* ⇒ *'a interp* **where**
  interp C = (⋃*D* ∈ {*D*. *D* #⊂# *C*}. *production D*)

lemma *production-unfold*:
  production C = {*A*. *C* ∈ *N* ∧ *C* ≠ {#} ∧ *Max* (*set-mset C*) = *Pos A*∧ *count C* (*Pos A*) ≤ *1* ∧ ¬
interp C ⊨*h C* ∧ *S C* = {#}}
  ⟨*proof*⟩

abbreviation *productive A* ≡ (*production A* ≠ {})

abbreviation *produces* :: *'a clause* ⇒ *'a* ⇒ *bool* **where**
  produces C A ≡ *production C* = {*A*}

lemma *producesD*:
  produces C A ⟹ *C* ∈ *N* ∧ *C* ≠ {#} ∧ *Pos A* = *Max* (*set-mset C*) ∧ *count C* (*Pos A*) ≤ *1* ∧
    ¬ *interp C* ⊨*h C* ∧ *S C* = {#}
  ⟨*proof*⟩

lemma *produces C A* ⟹ *Pos A* ∈# *C*
  ⟨*proof*⟩

lemma *interp'-def-in-set*:
  interp C = (⋃*D* ∈ {*D* ∈ *N*. *D* #⊂# *C*}. *production D*)
  ⟨*proof*⟩

lemma *production-iff-produces*:
  produces D A ⟷ *A* ∈ *production D*
  ⟨*proof*⟩

definition *Interp* :: *'a clause* ⇒ *'a interp* **where**
  Interp C = *interp C* ∪ *production C*

lemma
  assumes *produces C P*
  shows *Interp C* ⊨*h C*

$\langle proof \rangle$

definition *INTERP* :: *'a interp* **where**
$INTERP = (\bigcup D \in N.\ production\ D)$

lemma *interp-subseteq-Interp*[*simp*]: $interp\ C \subseteq Interp\ C$
$\langle proof \rangle$

lemma *Interp-as-UNION*: $Interp\ C = (\bigcup D \in \{D.\ D \ \#\subseteq\#\ C\}.\ production\ D)$
$\langle proof \rangle$

lemma *productive-not-empty*: $productive\ C \Longrightarrow C \neq \{\#\}$
$\langle proof \rangle$

lemma *productive-imp-produces-Max-literal*: $productive\ C \Longrightarrow produces\ C\ (atm\text{-}of\ (Max\ (set\text{-}mset\ C)))$
$\langle proof \rangle$

lemma *productive-imp-produces-Max-atom*: $productive\ C \Longrightarrow produces\ C\ (Max\ (atms\text{-}of\ C))$
$\langle proof \rangle$

lemma *produces-imp-Max-literal*: $produces\ C\ A \Longrightarrow A = atm\text{-}of\ (Max\ (set\text{-}mset\ C))$
$\langle proof \rangle$

lemma *produces-imp-Max-atom*: $produces\ C\ A \Longrightarrow A = Max\ (atms\text{-}of\ C)$
$\langle proof \rangle$

lemma *produces-imp-Pos-in-lits*: $produces\ C\ A \Longrightarrow Pos\ A \in\#\ C$
$\langle proof \rangle$

lemma *productive-in-N*: $productive\ C \Longrightarrow C \in N$
$\langle proof \rangle$

lemma *produces-imp-atms-leq*: $produces\ C\ A \Longrightarrow B \in atms\text{-}of\ C \Longrightarrow B \leq A$
$\langle proof \rangle$

lemma *produces-imp-neg-notin-lits*: $produces\ C\ A \Longrightarrow \neg\ Neg\ A \in\#\ C$
$\langle proof \rangle$

lemma *less-eq-imp-interp-subseteq-interp*: $C\ \#\subseteq\#\ D \Longrightarrow interp\ C \subseteq interp\ D$
$\langle proof \rangle$

lemma *less-eq-imp-interp-subseteq-Interp*: $C\ \#\subseteq\#\ D \Longrightarrow interp\ C \subseteq Interp\ D$
$\langle proof \rangle$

lemma *less-imp-production-subseteq-interp*: $C\ \#\subset\#\ D \Longrightarrow production\ C \subseteq interp\ D$
$\langle proof \rangle$

lemma *less-eq-imp-production-subseteq-Interp*: $C\ \#\subseteq\#\ D \Longrightarrow production\ C \subseteq Interp\ D$
$\langle proof \rangle$

lemma *less-imp-Interp-subseteq-interp*: $C\ \#\subset\#\ D \Longrightarrow Interp\ C \subseteq interp\ D$
$\langle proof \rangle$

lemma *less-eq-imp-Interp-subseteq-Interp*: $C\ \#\subseteq\#\ D \Longrightarrow Interp\ C \subseteq Interp\ D$

⟨*proof*⟩

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin$ *Interp* $C \Longrightarrow A \in$ *interp* $D \Longrightarrow C$ #⊂# D
⟨*proof*⟩

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin$ *interp* $C \Longrightarrow A \in$ *interp* $D \Longrightarrow C$ #⊂# D
⟨*proof*⟩

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin$ *Interp* $C \Longrightarrow A \in$ *Interp* $D \Longrightarrow C$ #⊂# D
⟨*proof*⟩

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin$ *interp* $C \Longrightarrow A \in$ *Interp* $D \Longrightarrow C$ #⊆# D
⟨*proof*⟩

lemma *interp-subseteq-INTERP*: *interp* $C \subseteq$ *INTERP*
⟨*proof*⟩

lemma *production-subseteq-INTERP*: *production* $C \subseteq$ *INTERP*
⟨*proof*⟩

lemma *Interp-subseteq-INTERP*: *Interp* $C \subseteq$ *INTERP*
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: *Neg* A ∈# C **and** *d*: *produces* D A
shows $A \in$ *interp* C
⟨*proof*⟩

lemma *neg-notin-Interp-not-produce*: *Neg* A ∈# $C \Longrightarrow A \notin$ *Interp* $D \Longrightarrow C$ #⊆# $D \Longrightarrow \neg$ *produces* D'' A
⟨*proof*⟩

lemma *in-production-imp-produces*: $A \in$ *production* $C \Longrightarrow$ *produces* C A
⟨*proof*⟩

lemma *not-produces-imp-notin-production*: $\neg$ *produces* C $A \Longrightarrow A \notin$ *production* C
⟨*proof*⟩

lemma *not-produces-imp-notin-interp*: $(\bigwedge D.\ \neg$ *produces* D $A) \Longrightarrow A \notin$ *interp* C
⟨*proof*⟩

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 #⊆# D **and**
d-lt-d': D #⊂# D' **and**
c-at-d: *Interp* $D \models h$ C **and**
subs: *interp* $D' \subseteq (\bigcup C \in CC.$ *production* $C)$
shows $(\bigcup C \in CC.$ *production* $C) \models h$ C
⟨*proof*⟩

lemma *true-Interp-imp-interp*: C #⊆# $D \Longrightarrow D$ #⊂# $D' \Longrightarrow$ *Interp* $D \models h$ $C \Longrightarrow$ *interp* $D' \models h$ C
⟨*proof*⟩

lemma *true-Interp-imp-Interp*: $C \ \#\subseteq\# \ D \Longrightarrow D \ \#\subset\# \ D' \Longrightarrow \mathit{Interp}\ D \models h\ C \Longrightarrow \mathit{Interp}\ D' \models h\ C$
⟨*proof*⟩

lemma *true-Interp-imp-INTERP*: $C \ \#\subseteq\# \ D \Longrightarrow \mathit{Interp}\ D \models h\ C \Longrightarrow \mathit{INTERP} \models h\ C$
⟨*proof*⟩

lemma *true-interp-imp-general*:
assumes
c-le-d: $C \ \#\subseteq\# \ D$ **and**
d-lt-d': $D \ \#\subset\# \ D'$ **and**
c-at-d: $\mathit{interp}\ D \models h\ C$ **and**
subs: $\mathit{interp}\ D' \subseteq (\bigcup C \in CC.\ \mathit{production}\ C)$
shows $(\bigcup C \in CC.\ \mathit{production}\ C) \models h\ C$
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *true-interp-imp-interp*: $C \ \#\subseteq\# \ D \Longrightarrow D \ \#\subset\# \ D' \Longrightarrow \mathit{interp}\ D \models h\ C \Longrightarrow \mathit{interp}\ D' \models h\ C$
⟨*proof*⟩

lemma *true-interp-imp-Interp*: $C \ \#\subseteq\# \ D \Longrightarrow D \ \#\subset\# \ D' \Longrightarrow \mathit{interp}\ D \models h\ C \Longrightarrow \mathit{Interp}\ D' \models h\ C$
⟨*proof*⟩

lemma *true-interp-imp-INTERP*: $C \ \#\subseteq\# \ D \Longrightarrow \mathit{interp}\ D \models h\ C \Longrightarrow \mathit{INTERP} \models h\ C$
⟨*proof*⟩

lemma *productive-imp-false-interp*: $\mathit{productive}\ C \Longrightarrow \neg\ \mathit{interp}\ C \models h\ C$
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *cls-gt-double-pos-no-production*:
assumes D: $\{\#\mathit{Pos}\ P,\ \mathit{Pos}\ P\#\} \ \#\subset\# \ C$
shows $\neg \mathit{produces}\ C\ P$
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma
assumes D: $C+\{\#\mathit{Neg}\ P\#\} \ \#\subset\# \ D$
shows $\mathit{production}\ D \neq \{P\}$
⟨*proof*⟩

lemma *in-interp-is-produced*:
assumes $P \in \mathit{INTERP}$
shows $\exists\, D.\ D + \{\#\mathit{Pos}\ P\#\} \in N \wedge \mathit{produces}\ (D + \{\#\mathit{Pos}\ P\#\})\ P$
⟨*proof*⟩

end
end

abbreviation $\mathit{MMax}\ M \equiv \mathit{Max}\ (\mathit{set\text{-}mset}\ M)$

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* $(C + \{\#Pos\ P\#\} + \{\#Pos\ P\#\})\ B\ (C + \{\#Pos\ P\#\})$ |
superposition-l: *superposition-rules* $(C_1 + \{\#Pos\ P\#\})\ (C_2 + \{\#Neg\ P\#\})\ (C_1 + C_2)$

inductive *superposition* :: $'a$ *clauses* $\Rightarrow$ $'a$ *clauses* $\Rightarrow$ *bool* **where**
superposition: $A \in N \Longrightarrow B \in N \Longrightarrow$ *superposition-rules* $A\ B\ C$
$\Longrightarrow$ *superposition* $N\ (N \cup \{C\})$

definition *abstract-red* :: $'a$::*wellorder clause* $\Rightarrow$ $'a$ *clauses* $\Rightarrow$ *bool* **where**
abstract-red $C\ N = ($*clss-lt* $N\ C \models p\ C)$

lemma *less-multiset*[*iff*]: $M < N \longleftrightarrow M\ \#{\subset}\#\ N$
⟨*proof*⟩

lemma *less-eq-multiset*[*iff*]: $M \leq N \longleftrightarrow M\ \#{\subseteq}\#\ N$
⟨*proof*⟩

lemma *herbrand-true-clss-true-clss-cls-herbrand-true-clss*:
assumes
AB: $A \models hs\ B$ **and**
BC: $B \models p\ C$
shows $A \models h\ C$
⟨*proof*⟩

lemma *abstract-red-subset-mset-abstract-red*:
assumes
abstr: *abstract-red* $C\ N$ **and**
c-lt-d: $C \subseteq\#\ D$
shows *abstract-red* $D\ N$
⟨*proof*⟩

lemma *true-clss-cls-extended*:
assumes
$A \models p\ B$ **and**
tot: *total-over-m* $I\ (A)$ **and**
cons: *consistent-interp* I **and**
I-A: $I \models s\ A$
shows $I \models B$
⟨*proof*⟩
lemma
assumes
CP: $\neg$ *clss-lt* $N\ (\{\#C\#\} + \{\#E\#\}) \models p\ \{\#C\#\} + \{\#Neg\ P\#\}$ **and**
clss-lt $N\ (\{\#C\#\} + \{\#E\#\}) \models p\ \{\#E\#\} + \{\#Pos\ P\#\} \lor$ *clss-lt* $N\ (\{\#C\#\} + \{\#E\#\}) \models p$
$\{\#C\#\} + \{\#Neg\ P\#\}$
shows *clss-lt* $N\ (\{\#C\#\} + \{\#E\#\}) \models p\ \{\#E\#\} + \{\#Pos\ P\#\}$

⟨*proof*⟩

locale *ground-ordered-resolution-with-redundancy* =
ground-resolution-with-selection +
fixes *redundant* :: $'a$::*wellorder clause* $\Rightarrow$ $'a$ *clauses* $\Rightarrow$ *bool*
assumes
redundant-iff-abstract: *redundant* $A\ N \longleftrightarrow$ *abstract-red* $A\ N$

begin
definition *saturated* :: $'a$ *clauses* $\Rightarrow$ *bool* **where**
saturated $N \longleftrightarrow (\forall A\ B\ C.\ A \in N \longrightarrow B \in N \longrightarrow \neg redundant\ A\ N \longrightarrow \neg redundant\ B\ N$
$\longrightarrow superposition\text{-}rules\ A\ B\ C \longrightarrow redundant\ C\ N \vee C \in N)$

lemma
assumes
saturated: *saturated* N **and**
finite: *finite* N **and**
empty: $\{\#\} \notin N$
shows $INTERP\ N \models hs\ N$
⟨*proof*⟩

end

lemma *tautology-is-redundant*:
assumes *tautology* C
shows *abstract-red* $C\ N$
⟨*proof*⟩

lemma *subsumed-is-redundant*:
assumes AB: $A \subset\# B$
and AN: $A \in N$
shows *abstract-red* $B\ N$
⟨*proof*⟩

inductive *redundant* :: $'a$ *clause* $\Rightarrow$ $'a$ *clauses* $\Rightarrow$ *bool* **where**
subsumption: $A \in N \Longrightarrow A \subset\# B \Longrightarrow redundant\ B\ N$

lemma *redundant-is-redundancy-criterion*:
fixes A :: $'a$ :: *wellorder clause* **and** N :: $'a$ :: *wellorder clauses*
assumes *redundant* $A\ N$
shows *abstract-red* $A\ N$
⟨*proof*⟩

lemma *redundant-mono*:
$redundant\ A\ N \Longrightarrow A \subseteq\# B \Longrightarrow redundant\ B\ N$
⟨*proof*⟩

locale *truc* =
selection S **for** S :: *nat clause* $\Rightarrow$ *nat clause*
begin

end

end