SAT Solver verification

By Filip Marić

August 28, 2014

Abstract

This document contains formal correctness proofs of modern SAT solvers. Two different approaches are used — state-transition systems and shallow embedding into HOL.

Formalization based on state-transition systems follows [1, 3]. Several different SAT solver descriptions are given and their partial correctness and termination is proved. These include:

1. a solver based on classical DPLL procedure (based on backtracking search with unit propagation),
2. a very general solver with backjumping and learning (similar to the description given in [3]), and
3. a solver with a specific conflict analysis algorithm (similar to the description given in [1]).

Formalization based on shallow embedding into HOL defines a SAT solver as a set or recursive HOL functions. Solver supports most state-of-the-art techniques including the two-watch literal propagation scheme.

Within the SAT solver correctness proofs, a large number of lemmas about propositional logic and CNF formulae are proved. This theory is self-contained and could be used for further exploring of properties of CNF based SAT algorithms.

Contents

1 MoreList 3
  1.1 last and butlast - last element of list and elements before it 3
  1.2 removeAll - element removal 4
  1.3 uniq - no duplicate elements 5
  1.4 firstPos - first position of an element 7
  1.5 precedes - ordering relation induced by firstPos 10
  1.6 isPrefix - prefixes of list 18
  1.7 list-diff - the set difference operation on two lists 20
  1.8 remdups - removing duplicates 22
  1.9 Levi’s lemma 29
  1.10 Single element lists 30
2 CNF

2.1 Syntax ............................................. 31
  2.1.1 Basic datatypes .............................. 31
  2.1.2 Membership ................................. 31
  2.1.3 Variables .................................. 32
  2.1.4 Opposite literals ........................... 35
  2.1.5 Tautological clauses ....................... 38

2.2 Semantics ....................................... 38
  2.2.1 Valuations ................................ 38
  2.2.2 True/False literals ....................... 39
  2.2.3 True/False clauses ....................... 41
  2.2.4 True/False formulae ...................... 43
  2.2.5 Valuation viewed as a formula ............ 46
  2.2.6 Consistency of valuations ................. 46
  2.2.7 Totality of valuations .................... 50
  2.2.8 Models and satisfiability ................. 53
  2.2.9 Tautological clauses ..................... 55
  2.2.10 Entailment ................................ 60
  2.2.11 Equivalency ................................ 72
  2.2.12 Remove false and duplicate literals of a clause .. 76
  2.2.13 Resolution ............................... 81
  2.2.14 Unit clauses ............................. 84
  2.2.15 Reason clauses ........................... 86
  2.2.16 Last asserted literal of a list .......... 89

3 Trail datatype definition and its properties 94

  3.1 Trail elements ............................... 94
  3.2 Marked trail elements ....................... 97
  3.3 Prefix before/upto a trail element .......... 99
  3.4 Marked elements upto a given trail element .. 102
  3.5 Last marked element in a trail ............... 103
  3.6 Level of a trail element ..................... 104
  3.7 Current trail level .......................... 109
  3.8 Prefix to a given trail level ............... 110
  3.9 Number of literals of every trail level .... 121
  3.10 Prefix before last marked element .......... 126

4 Verification of DPLL based SAT solvers. 128

  4.1 Literal Trail .................................. 129
  4.2 Invariants .................................... 129
    4.2.1 Auxiliary lemmas ......................... 130
    4.2.2 Transition rules preserve invariants .... 134
  4.3 Different characterizations of backjumping ... 157
  4.4 Termination .................................. 174
    4.4.1 Trail ordering ............................ 175
    4.4.2 Conflict clause ordering ................. 192
    4.4.3 ConflictFlag ordering .................... 196
    4.4.4 Formulae ordering ....................... 197
    4.4.5 Properties of well-founded relations .... 198
5 BasicDPLL

5.1 Specification ................................................. 200
5.2 Invariants ...................................................... 204
5.3 Soundness ...................................................... 209
5.4 Termination ..................................................... 211
5.5 Completeness .................................................. 219

6 Transition system of Nieuwenhuis, Oliveras and Tinelli. 223

6.1 Specification ................................................. 224
6.2 Invariants ...................................................... 229
6.3 Soundness ...................................................... 237
6.4 Termination ..................................................... 238
6.5 Completeness .................................................. 252

7 Transition system of Krsti´ c and Goel. 259

7.1 Specification ................................................. 259
7.2 Invariants ...................................................... 268
7.3 Soundness ...................................................... 282
7.4 Termination ..................................................... 283
7.5 Completeness .................................................. 303

8 Functional implementation of a SAT solver with Two Watch literal propagation. 309

8.1 Specification ................................................. 309
8.2 Total correctness theorem ................................. 730

1 MoreList

theory MoreList
imports Main ~~/src/HOL/Library/Multiset
begin

Theory contains some additional lemmas and functions for the List datatype. Warning: some of these notions are obsolete because they already exist in List.thy in similar form.

1.1 last and butlast - last element of list and elements before it

lemma listEqualsButlastAppendLast:
  assumes list ≠ []
  shows list = (butlast list) @ [last list]
using assms
by (induct list) auto

lemma lastListInList [simp]:
  assumes list ≠ []
  shows last list ∈ set list
using assms
by (induct list) auto

lemma butlastIsSubset:
  shows set (butlast list) ⊆ set list
by (induct list) (auto split: split-if-asm)

lemma setListIsSetButlastAndLast:
  shows set list ⊆ set (butlast list) ∪ {last list}
by (induct list) auto

lemma butlastAppend:
  shows butlast (list1 @ list2) = (if list2 = [] then butlast list1 else (list1 @ butlast list2))
by (induct list1) auto

1.2 removeAll - element removal

lemma removeAll-multiset:
  assumes distinct a x ∈ set a
  shows multiset-of a = {#x#} + multiset-of (removeAll x a)
using assms
proof (induct a)
  case (Cons y a')
  thus thesis
  proof (cases x = y)
    case True
    with '>'distinct (y # a') (x ∈ set (y # a'))
    have ¬ x ∈ set a'
      by auto
    hence removeAll x a' = a'
      by (rule removeAll-id)
    with (x = y) show thesis
      by (simp add: union-commute)
  next
    case False
    with (x ∈ set (y # a'))
    have x ∈ set a'
      by simp
    with '>'distinct (y # a')
    have x # y distinct a'
      by auto
    hence multiset-of a' = {#x#} + multiset-of (removeAll x a')
      using (x ∈ set a')
      using Cons(1)
      by simp
    thus thesis
      using (x # y)
      by (simp add: union-assoc)
  qed
qed simp

lemma removeAll-map:
  assumes \( \forall x \ y. x \neq y \rightarrow f x \neq f y \)
  shows \( \text{removeAll} (f x) (\text{map} \ f \ a) = \text{map} \ f \ (\text{removeAll} \ x \ a) \)
  using assms
  by (induct a arbitrary: \( x \)) auto

1.3 uniq - no duplicate elements.

uniq list holds iff there are no repeated elements in a list. Obso-
lete: same as distinct in List.thy.

primrec uniq :: 'a list => bool
  where
    uniq [] = True |
    uniq (h#t) = (h \notin set t \&\& uniq t)

lemma uniqDistinct:
  uniq l = distinct l
  by (induct l) auto

lemma uniqAppend:
  assumes uniq (l1 @ l2)
  shows uniq l1 uniq l2
  using assms
  by (induct l1) auto

lemma uniqAppendIff:
  uniq (l1 @ l2) = (uniq l1 \&\& uniq l2 \&\& set l1 \&\& set l2 = \\{\}) \( \text{is} \ \text{lhs} = \text{rhs} \)
  by (induct l1) auto

lemma uniqAppendElement:
  assumes uniq l
  shows e \notin set l = uniq (l @ [e])
  using assms
  by (induct l) (auto split: split-if-asm)

lemma uniqImpliesNotLastMemButlast:
  assumes uniq l
  shows last l \notin set (butlast l)
  proof (cases l = [])
    case True
    thus \( \text{thesis} \)
      using assms
      by simp
    next
    case False
    hence l = butlast l @ [last l]

by (rule listEqualsButlastAppendLast)
moreover
with (uniq l)
have uniq (butlast l)
  using uniqAppend[of butlast l [last l]]
  by simp
ultimately
show ?thesis
  using assms
  using uniqAppendElement[of butlast l last l]
  by simp
qed

lemma uniqButlastNotUniqListImpliesLastMemButlast:
  assumes uniq (butlast l) ¬ uniq l
  shows last l ∈ set (butlast l)
proof (cases l = [])
  case True
  thus ?thesis
  using assms
  by auto
next
  case False
  hence l = butlast l @ [(last l)]
  by (rule listEqualsButlastAppendLast)
  thus ?thesis
  using assms
  using uniqAppendElement[of butlast l last l]
  by auto
qed

lemma uniqRemdups:
  shows uniq (remdups x)
by (induct x) auto

lemma uniqHead TailSet:
  assumes uniq l
  shows set (tl l) = (set l) − {hd l}
using assms
by (induct l) auto

lemma uniqLengthEqCardSet:
  assumes uniq l
  shows length l = card (set l)
using assms
by (induct l) auto

lemma lengthGtOneTwoDistinctElements:
  assumes
uniq \ l \ length \ l \ > \ 1 \ l \ \neq \ []
shows
\exists \ a1 \ a2. \ a1 \in \set \ l \ \land \ a2 \in \set \ l \ \land \ a1 \ \neq \ a2
proof—
let \ ?a1 = \ l \ ! \ 0
let \ ?a2 = \ l \ ! \ 1
have \ ?a1 \in \set \ l
  using \ nth-mem[of \ 0 \ l]
  using \ assms
  by \ simp
moreover
have \ ?a2 \in \set \ l
  using \ nth-mem[of \ 1 \ l]
  using \ assms
  by \ simp
moreover
have \ ?a1 \neq \ ?a2
  using \ nth-eq-iff-index-eq[of \ l \ 0 \ 1]
  using \ assms
  by \ (auto \ simp \ add: \ uniqDistinct)
ultimately
show \ ?thesis
  by \ auto
qed

1.4 \ firstPos - \ first \ position \ of \ an \ element

\firstPos \ returns \ the \ zero-based \ index \ of \ the \ first \ occurrence \ of \ an
\ element \ into \ a \ list, \ or \ the \ length \ of \ the \ list \ if \ the \ element \ does \ not
\ occur.

primrec \ firstPos :: \ 'a :=> 'a \ list \ => \ nat
where
firstPos \ a \ [] = 0 |
firstPos \ a \ (h \ # \ t) = \ (if \ a = h \ then \ 0 \ else \ 1 + (firstPos \ a \ t))

lemma \ firstPosEqualZero:
  shows \ (firstPos \ a \ (m \ # \ M') = 0) = (a = m)
by \ (induct \ M') \ auto

lemma \ firstPosLeLength:
  assumes \ a \in \set \ l
  shows \ firstPos \ a \ l \ < \ length \ l
using \ assms
by \ (induct \ l) \ auto

lemma \ firstPosAppend:
  assumes \ a \in \set \ l
  shows \ firstPos \ a \ l = \ firstPos \ a \ (l \ @ \ l')
using \ assms
by (induct l) auto

lemma firstPosAppendNonMemberFirstMemberSecond:
  assumes a \notin \text{set } l1 \text{ and } a \in \text{set } l2
  shows firstPos a (l1 \oplus l2) = \text{length } l1 + \text{firstPos } a \text{ l2}
  using assms
  by (induct l1) auto

lemma firstPosDomainForElements:
  shows \((0 \leq \text{firstPos } a \text{ l } \land \text{firstPos } a \text{ l } < \text{length } l) = \text{ (a } \in \text{ set } l)\) \text{ (is } ?lhs \text{ = } ?rhs)\n  by (induct l) auto

lemma firstPosEqual:
  assumes a \in \text{set } l \text{ and } b \in \text{set } l
  shows \((\text{firstPos } a \text{ l } = \text{firstPos } b \text{ l}) = \text{ (a = b)}\) \text{ (is } ?lhs = ?rhs)\n
proof–
{ }
  assume ?lhs
  hence ?rhs
    using assms
  proof (induct l)
    case (Cons m l')
    { }
      assume a = m
      have b = m
        proof–
          from \(a = m\)
          have firstPos a (m \# l') = 0
            by simp
          with Cons
          have firstPos b (m \# l') = 0
            by simp
          with \(b \in \text{set } (m \# l')\)
          have firstPos b (m \# l') = 0
            by simp
          thus ?thesis
            using firstPosEqualZero[of b m l']
            by simp
        qed
        with \(a = m\)
        have ?case
          by simp
    }
  note * = this
  moreover
  { }
  assume b = m
  have a = m

proof

from \( \langle b = m \rangle \)
have \( \text{firstPos } b \ (m \# l') = 0 \)
  by simp
with Cons
have \( \text{firstPos } a \ (m \# l') = 0 \)
  by simp
with \( \langle a \in \text{set } (m \# l') \rangle \)
have \( \text{firstPos } a \ (m \# l') = 0 \)
  by simp
thus \(?thesis\)
  using \( \text{firstPosEqualZero[of } a \ m \ l' \) \)
  by simp
qed
with \( \langle b = m \rangle \)
have \(?case\)
  by simp
}

note \(\ast\ast = \text{this}\)

moreover

\{  
assume \( Q : a \neq m \ b \neq m \)
from \( Q \ \langle a \in \text{set } (m \# l') \rangle \)
have \( a \in \text{set } l' \)
  by simp
from \( Q \ \langle b \in \text{set } (m \# l') \rangle \)
have \( b \in \text{set } l' \)
  by simp
from \( \langle a \in \text{set } l' \ \langle b \in \text{set } l' \ Cons \rangle \)
have \( \text{firstPos } a \ l' = \text{firstPos } b \ l' \)
  by (simp split: split-if-asm)
with Cons
have \(?case\)
  by (simp split: split-if-asm)
\}

note \(\ast\ast\ast = \text{this}\)

moreover

\{
  have \( a = m \lor b = m \lor a \neq m \land b \neq m \)
    by auto
\}

ultimately
show \(?thesis\)

proof (case a = m)
  case True
  thus \(?thesis\)
    by (rule \(\ast\))
next
  case False
thus \( ?\text{thesis} \)
proof (cases \( b = m \))
  case True
  thus \( ?\text{thesis} \)
    by (rule **) 
next 
  case False 
  with \( \langle a \neq m \rangle \) show \( ?\text{thesis} \)
    by (rule ***) 
qed
qed simp 
\} 
thus \( ?\text{thesis} \)
  by auto 
qed

lemma firstPosLast:
assumes \( l \neq [] \) uniq \( l \) 
shows \( (\text{firstPos} \ x \ l = \text{length} \ l - 1) = (x = \text{last} \ l) \)
using assms 
by (induct \( l \) ) auto

1.5 precedes - ordering relation induced by firstPos 
definition precedes :: \( 'a \Rightarrow \'a \ightarrow \'a \) list \Rightarrow \text{bool} 
where
precedes \( a \ b \ l \) \( \equiv \) (\( a \in \text{set} \ l \ \wedge \ b \in \text{set} \ l \ \wedge \ \text{firstPos} \ a \ l \ < = \ \text{firstPos} \ b \ l \) )

lemma noElementsPrecedesFirstElement:
assumes \( a \neq b \) 
shows \( \neg \text{precedes} \ a \ b \ (b \# \text{list}) \)
proof-
\{ 
  assume precedes \( a \ b \ (b \# \text{list}) \) 
  hence \( a \in \text{set} \ (b \# \text{list}) \) \( \text{firstPos} \ a \ (b \# \text{list}) \ < = \ 0 \) 
    unfolding precedes-def
    by (auto split: split-if-asm)
  hence \( \text{firstPos} \ a \ (b \# \text{list}) \ = \ 0 \) 
    by auto 
  with \( \langle a \neq b \rangle \) 
  have \( \text{False} \) 
    using \( \text{firstPosEqualZero}[@ \ a \ b \ \text{list}] \)
    by simp 
\} 
thus \( ?\text{thesis} \)
  by auto 
qed
lemma lastPrecedesNoElement:
assumes uniq l
shows ¬(∃ a. a ≠ last l ∧ precedes (last l) a l)
proof-
{
  assume ¬ ?thesis
  then obtain a
    where precedes (last l) a l a ≠ last l
    by auto
  hence a ∈ set l last l ∈ set l firstPos (last l) l ≤ firstPos a l
    unfolding precedes-def
    by auto
  hence length l − 1 ≤ firstPos a l
    using firstPosLast[of l last l]
    using ⟨uniq l⟩
    by force
  hence firstPos a l = length l − 1
    using firstPosAppend[of a l last l]
    using ⟨a ∈ set l⟩
    by auto
  hence a = last l
    using firstPosAppend[of a l last l]
    using ⟨a ∈ set l ⟩ ⟨last l ∈ set l ⟩
    using ⟨uniq l ⟩
    by force
  with ⟨a ≠ last l⟩
  have False
    by simp
}
thus ?thesis
  by auto
qed

lemma precedesAppend:
assumes precedes a b l
shows precedes a b (l @ l')
proof−
from ⟨precedes a b⟩
have a ∈ set l b ∈ set l firstPos a l ≤ firstPos b l
  unfolding precedes-def
  by (auto split: split-if-asm)
thus ?thesis
  using firstPosAppend[of a l l']
  using firstPosAppend[of b l l']
  unfolding precedes-def
  by simp
qed
lemma precedesMemberHeadMemberTail:
assumes \( a \in \text{set } l_1 \) and \( b \notin \text{set } l_1 \) and \( b \in \text{set } l_2 \)
shows \( \text{precedes } a \ b \ (l_1 @ l_2) \)
proof –
from \( \langle a \in \text{set } l_1 \rangle \)
have \( \text{firstPos } a \ \text{length } l_1 \)
using \( \text{firstPosLeLength } \langle \text{of } l_1 \rangle \)
by simp
moreover
from \( \langle a \in \text{set } l_1 \rangle \)
have \( \text{firstPos } a \ (l_1 @ l_2) = \text{firstPos } a \ l_1 \)
using \( \text{firstPosAppend } \langle \text{of } a \ l_1 \ l_2 \rangle \)
by simp
moreover
from \( \langle b \notin \text{set } l_1 \rangle \ \langle b \in \text{set } l_2 \rangle \)
have \( \text{firstPos } b \ (l_1 @ l_2) = \text{length } l_1 + \text{firstPos } b \ l_2 \)
by (rule firstPosAppendNonMemberFirstMemberSecond)
moreover
have \( \text{firstPos } b \ l_2 \geq 0 \)
by auto
ultimately
show ?thesis
unfolding \( \text{precedes-def} \)
using \( \langle a \in \text{set } l_1 \rangle \ \langle b \in \text{set } l_2 \rangle \)
by simp
qed

lemma precedesReflexivity:
assumes \( a \in \text{set } l \)
shows \( \text{precedes } a \ a \ l \)
using assms
unfolding \( \text{precedes-def} \)
by simp

lemma precedesTransitivity:
assumes \( \text{precedes } a \ b \ l \) and \( \text{precedes } b \ c \ l \)
shows \( \text{precedes } a \ c \ l \)
using assms
unfolding \( \text{precedes-def} \)
by auto

lemma precedesAntisymmetry:
assumes \( a \in \text{set } l \) and \( b \in \text{set } l \) and \( \text{precedes } a \ b \ l \) and \( \text{precedes } b \ a \ l \)
shows
\( a = b \)

proof

from \textit{assms}

have \( \text{firstPos } a \ l = \text{firstPos } b \ l \)

unfolding \texttt{precedes-def}

by \textit{auto}

thus \ \texttt{?thesis}

using \texttt{firstPosEqual[of \ a \ l \ \ b]}

using \textit{assms}

by \textit{simp}

qed

\textbf{lemma} \texttt{precedesTotalOrder}:\n
\textit{assumes} \( a \in \text{set } l \) \textit{and} \( b \in \text{set } l \)

\textit{shows} \( a = b \lor \text{precedes } a \ b \ l \lor \text{precedes } b \ a \ l \)

using \textit{assms}

unfolding \texttt{precedes-def}

by \textit{auto}

\textbf{lemma} \texttt{precedesMap}:\n
\textit{assumes} \( \text{precedes } a \ b \ \text{list} \) \textit{and} \( \forall \ x \ y. \ x \neq y \longrightarrow f \ x \neq f \ y \)

\textit{shows} \( \text{precedes } (f \ a) \ (f \ b) \ (\text{map } f \ \text{list}) \)

using \textit{assms}

\textbf{proof} (induct \textit{list})

\textbf{case} \( (\texttt{Cons } l \ \textit{list'}) \)

\{

\textbf{assume} \( a = l \)

\textbf{have} \ \texttt{?case}

\textbf{proof} –

\textbf{from} \( \langle a = l \rangle \)

\textbf{have} \( \text{firstPos } (f \ a) \ (\text{map } f \ (l \ \# \ \textit{list'})) = 0 \)

\textbf{using} \texttt{firstPosEqualZero[of \ a \ f \ l \ map \ f \ \textit{list}]}

\textbf{by} \textit{simp}

\textbf{moreover}

\textbf{from} \( \langle \text{precedes } a \ b \ (l \ \# \ \textit{list'}) \rangle \)

\textbf{have} \( b \in \text{set } (l \ \# \ \textit{list'}) \)

\textbf{unfolding} \texttt{precedes-def}

\textbf{by} \textit{simp}

\textbf{hence} \( f \ b \in \text{set } (\text{map } f \ (l \ \# \ \textit{list'})) \)

\textbf{by} \textit{auto}

\textbf{moreover}

\textbf{hence} \( \text{firstPos } (f \ b) \ (\text{map } f \ (l \ \# \ \textit{list'})) \geq 0 \)

\textbf{by} \textit{auto}

\textbf{ultimately}

\textbf{show} \ \texttt{?thesis}

\textbf{using} \( \langle a = l \rangle ; f b \in \text{set } (\text{map } f \ (l \ \# \ \textit{list'})) \)

\textbf{unfolding} \texttt{precedes-def}

\textbf{by} \textit{simp}

qed
moreover
{
  assume \(b = l\)
  with \(\text{precedes } a \ b \ (l \neq \text{list}')\);
  have \(a = l\)
    using \(\text{noElementsPrecedesFirstElement}\[\text{of } a \ \text{list}']\)
    by auto
  from \(a = l\) \(b = l\)
  have \(?\text{case}\)
    unfolding \(\text{precedes-def}\)
    by \(\text{simp}\)
}
moreover
{
  assume \(a \neq l \ b \neq l\)
  with \(\forall \ x \ y. \ x \neq y \rightarrow f x \neq f y\)
  have \(f \ a \neq f l \ f \ b \neq f l\)
    by auto
  from \(\text{precedes } a \ b \ (l \neq \text{list}')\)
  have \(b \in \text{set}(l \neq \text{list}') \ a \in \text{set}(l \neq \text{list}') \ \text{firstPos } a \ (l \neq \text{list}') \leq \text{firstPos } b \ (l \neq \text{list}')\)
    unfolding \(\text{precedes-def}\)
    by auto
  with \(\forall \ a \neq l \ b \neq l\)
  have \(a \in \text{set list'} \ b \in \text{set list'} \ \text{firstPos } a \ \text{list'} \leq \text{firstPos } b \ \text{list'}\)
    by auto
  hence \(\text{precedes } a \ b \ \text{list'}\)
    unfolding \(\text{precedes-def}\)
    by \(\text{simp}\)
  with \(\text{Cons}\)
  have \(\text{precedes } (f \ a) \ (f \ b) \ (\text{map } f \ \text{list}')\)
    by \(\text{simp}\)
  with \(\forall \ a \neq f \ b \ b \neq f \ b\)
  have \(?\text{case}\)
    unfolding \(\text{precedes-def}\)
    by auto
}
ultimately
show \(?\text{case}\)
  by auto
next
case \(\text{Nil}\)
thus \(?\text{case}\)
  unfolding \(\text{precedes-def}\)
  by \(\text{simp}\)
qed

lemma \(\text{precedesFilter}\):
assumes `precedes a b list` and `f a` and `f b`
shows `precedes a b (filter f list)`
using `assms`

proof (induct list)
case (Cons l list′)
show ?case
proof −
  from ⟨precedes a b (l # list′)⟩
  have `a ∈ set(l # list′)` `b ∈ set(l # list′)` \(\text{firstPos } a (l # list′) \leq \text{firstPos } b (l # list′)\)
  unfolding `precedes-def`
  by auto
  from ⟨f a⟩ ⟨a ∈ set(l # list′)⟩
  have `a ∈ set(filter f (l # list′))`
  by auto
moreover
  from ⟨f b⟩ ⟨b ∈ set(l # list′)⟩
  have `b ∈ set(filter f (l # list′))`
  by auto
moreover
  have `\text{firstPos } a (filter f (l # list′)) ≤ \text{firstPos } b (filter f (l # list′))`
proof −
  \{ 
  assume `a = l`
  with ⟨f a⟩
  have `\text{firstPos } a (filter f (l # list′)) = 0`
  by auto
  with ⟨b ∈ set (filter f (l # list′))⟩
  have ?thesis
  by auto
  \}
moreover
\{ 
  assume `b = l`
  with ⟨precedes a b (l # list′)⟩
  have `a = b`
  using `noElementsPrecedesFirstElement[of a b list′]`
  by auto
  hence ?thesis
  by (simp add: `precedesReflexivity`)
\}
moreover
\{ 
  assume `a ≠ l b ≠ l`
  with ⟨precedes a b (l # list′)⟩
  have `\text{firstPos } a \text{ list′} ≤ \text{firstPos } b \text{ list′)`
  unfolding `precedes-def`
  by auto
moreover
from \( a \neq b \, \langle \forall a \in \text{set} \,(l \# \text{list}') \rangle \)
have \( a \in \text{set list}' \)
by simp
moreover
from \( b \neq l \, \langle \forall b \in \text{set} \,(l \# \text{list}') \rangle \)
have \( b \in \text{set list}' \)
by simp
ultimately
have \( \text{precedes a b list}' \)
unfolding preceedes-def
by simp
with \( \langle f a \rangle \langle f b \rangle \text{Cons}(1) \)
have \( \text{precedes a b } (\text{filter f list}') \)
by simp
with \( \langle a \neq l \rangle \langle b \neq l \rangle \)
have \( \text{thesis} \)
unfolding preceedes-def
by auto
}
ultimately
show \( \text{thesis} \)
by blast
qed
ultimately
show \( \text{thesis} \)
unfolding preceedes-def
by simp
qed
definition
precedesOrder list == \{ (a, b). \text{precedes a b list} \land a \neq b \}\
lemma transPrecedesOrder:
trans \( (\text{precedesOrder list}) \)
proof-
{ 
  fix \( x \ y \ z \)
  assume \( \text{precedes x y list} \ x \neq y \text{precedes y z list} \ y \neq z \)
  hence \( \text{precedes x z list} \ x \neq z \)
  using \( \text{precedesTransitivity[of x y list z]} \)
  using \( \text{firstPosEqual[of y list z]} \)
  unfolding \( \text{precedes-def} \)
  by auto
}
thus \( \text{thesis} \)
unfolding trans-def
unfolding \( \text{precedesOrder-def} \)
by blast
lemma wellFoundedPrecedesOrder:
  shows \( \text{wf \ (precedesOrder \ list)} \)
unfolding \( \text{wf-eq-minimal} \)
proof –
  show \( \forall \ Q \ a. \ a:\ Q \rightarrow (\exists \ a\Min \in \ Q. \ \forall \ a'. \ (a', \ a\Min) \in \text{precedesOrder}\ list \rightarrow a' \notin \ Q) \)
  proof –
  \{
  fix a :: 'a and Q::'a set
  assume a \in Q
  let ?listQ = filter \( \lambda \ x. \ x \in Q)\ list
  have \( \exists \ a\Min \in Q. \ \forall \ a'. \ (a', \ a\Min) \in \text{precedesOrder}\ list \rightarrow a' \notin \ Q)\ 
  proof (cases ?listQ = [])
  case True
  let ?aMin = a
  have \( \forall \ a'. \ (a', \ ?aMin) \in \text{precedesOrder}\ list \rightarrow a' \notin \ Q)\ 
  proof –
  \{
  fix a'
  assume \( (a', \ ?aMin) \in \text{precedesOrder}\ list)\ 
  hence a \in set list
  unfolding \text{precedesOrder-def} \ 
  unfolding \text{precedes-def} \ 
  by simp
  with \( a \in Q)\ 
  have a \in set ?listQ
  by (induct list) auto
  with \( ?listQ = [])\ 
  have False
  by simp
  hence a' \notin Q
  by simp
  \}
  thus ?thesis
  by simp
  qed
  with \( a \in Q)\ obtain \ a\Min\ where \ a\Min \in Q \ \forall \ a'. \ (a', \ a\Min) \ 
  \in \text{precedesOrder}\ list \rightarrow a' \notin \ Q)\ 
  by auto
  thus ?thesis
  by auto
  next
  case False
  let ?aMin = hd ?listQ
  from False

qed
have \( a_{\text{Min}} \in Q \)
by (induct list) auto
have \( \forall a'. \ (a', a_{\text{Min}}) \in \text{precedesOrder list} \rightarrow a' \notin Q \)
proof
fix \( a' \)
{
assume \( (a', a_{\text{Min}}) \in \text{precedesOrder list} \)
hence \( a' \in \text{set list precedes a'} \ ?a_{\text{Min}} \ \text{list} \ a' \neq \ ?a_{\text{Min}} \)
unfolding \text{precedesOrder-def}
unfolding \text{precedes-def}
by auto
have \( a' \notin Q \)
proof
{
assume \( a' \in Q \)
with \( :a_{\text{Min}} \in Q \) \( (\text{precedes a'} \ ?a_{\text{Min}} \ \text{list}) \)
have \( \text{precedes a'} \ ?a_{\text{Min}} \ ?\text{list} \)
using \text{precedesFilter}[\text{of a'} \ ?a_{\text{Min}} \ \text{list} \ \lambda \ x. \ x \in Q]
by blast
from \( a' \neq a_{\text{Min}} \)
have \( \neg \text{precedes a'} \ (\text{hd} \ ?\text{list}Q) \ (\text{hd} \ ?\text{list}Q \neq \ \text{tl} \ ?\text{list}Q) \)
by (rule \text{noElementsPrecedesFirstElement})
with \( \text{False} \ \text{precedes a'} \ ?a_{\text{Min}} \ ?\text{list}Q \)
have \( \text{False} \)
by auto
}
thus \( \text{thesis} \)
by auto
qed
} thus \( (a', a_{\text{Min}}) \in \text{precedesOrder list} \rightarrow a' \notin Q \)
by simp
qed
with \( \langle a_{\text{Min}} \in Q \rangle \)
show \( \text{thesis} \)
..
qed
qed

1.6 \textit{isPrefix} - prefixes of list.

Check if a list is a prefix of another list. Obsolete: similiar notion
is defined in \textit{List_prefixes.thy}.

definition \textit{isPrefix} :: 'a list \\Rightarrow 'a list \\Rightarrow bool
where \textit{isPrefix} \ p \ t = (\exists \ s. \ p @ s = t)
lemma prefixIsSubset:
  assumes isPrefix p l
  shows set p ⊆ set l
  using assms
  unfolding isPrefix-def
  by auto

lemma uniqListImpliesUniqPrefix:
  assumes isPrefix p l and uniq l
  shows uniq p
  proof
    from ⟨isPrefix p l⟩ obtain s
    where p @ s = l
    unfolding isPrefix-def
    by auto
    with ⟨uniq l⟩
    show ?thesis
    using uniqAppend[of p s]
    by simp
  qed

lemma firstPosPrefixElement:
  assumes isPrefix p l and a ∈ set p
  shows firstPos a p = firstPos a l
  proof
    from ⟨isPrefix p l⟩ obtain s
    where p @ s = l
    unfolding isPrefix-def
    by auto
    from ⟨precedes a b l⟩
    have a ∈ set l b ∈ set l firstPos a l ≤ firstPos b l
    unfolding precedes-def
    by (auto simp add: firstPos_append)
  qed

lemma laterInPrefixRetainsPrecedes:
  assumes isPrefix p l and precedes a b l and b ∈ set p
  shows precedes a b p
  proof
    from ⟨isPrefix p l⟩ obtain s
    where p @ s = l
    unfolding isPrefix-def
    by auto
    from ⟨precedes a b l⟩
    have a ∈ set l b ∈ set l firstPos a l ≤ firstPos b l
    unfolding precedes-def
    by (auto simp add: firstPos_append)
by (auto split: split-if-asm)

from \( p \otimes s = l \) \( b \in \text{set } p \)
have \( \text{firstPos } b \ l = \text{firstPos } b \ p \)
  using \( \text{firstPosAppend } [of \ b \ p \ s] \)
by simp

show \( {?}\text{thesis} \)
proof (cases \( a \in \text{set } p \) )
case \( \text{True} \)
from \( p \otimes s = l \) \( a \in \text{set } p \)
have \( \text{firstPos } a \ l = \text{firstPos } a \ p \)
  using \( \text{firstPosAppend } [of \ a \ p \ s] \)
by simp

from \( \text{firstPos } a \ l = \text{firstPos } a \ p \) \( \text{firstPos } b \ l = \text{firstPos } b \ p \)
\( \text{firstPos } a \ l \leq \text{firstPos } b \ b \)
\( a \in \text{set } p \) \( b \in \text{set } p \)
show \( {?}\text{thesis} \)
  unfolding \( \text{precedes-def} \)
by simp

next
case \( \text{False} \)
from \( a \notin \text{set } p \) \( a \in \text{set } s \)
  \( p \otimes s = l \)
have \( a \in \text{set } s \)
  by auto
with \( a \notin \text{set } p \) \( p \otimes s = l \)
have \( \text{firstPos } a \ l = \text{length } p + \text{firstPos } a \ s \)
  using \( \text{firstPosAppendNonMemberFirstMemberSecond}[of \ a \ p \ s] \)
by simp
moreover
from \( b \in \text{set } p \)
have \( \text{firstPos } b \ p < \text{length } p \)
  by (rule \( \text{firstPosLeLength} \))
ultimately
show \( {?}\text{thesis} \)
  using \( \text{firstPos } b \ l = \text{firstPos } b \ p \) \( \text{firstPos } a \ l \leq \text{firstPos } b \ b \)
by simp
qed
qed

1.7 \( \text{list-diff} \) - the set difference operation on two lists.

primrec \( \text{list-diff} :: 'a \ \text{list} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{list} \)
where
\( \text{list-diff } x \ \text{[]} = x \ |
\text{list-diff } x \ (y#ys) = \text{list-diff}(\text{removeAll } y \ x) \ ys \)
lemma [simp]:
  shows  list-diff  []  y  =  []
by (induct y) auto

lemma [simp]:
  shows  list-diff  (x  #  xs)  y  =  (if  x  \in  set  y  then  list-diff  xs  y  else  x
  #  list-diff  xs  y)
proof (induct y arbitrary: xs)
  case (Cons y ys)
  thus  ?case
  proof (cases x = y)
    case True
    thus  ?thesis
    by  simp
  next
  case False
  thus  ?thesis
  proof (cases x \in  set  ys)
    case True
    thus  ?thesis
    using  Cons
    by  simp
  next
  case False
  thus  ?thesis
  using  Cons
  by  simp
  qed
  qed simp

lemma listDiffIff:
  shows  (x  \in  set  a  \&  x  \notin  set  b)  =  (x  \in  set  (list-diff  a  b))
by (induct a) auto

lemma listDiffDoubleRemoveAll:
  assumes  x  \in  set  a
  shows  list-diff  b  a  =  list-diff  b  (x  #  a)
using  assms
by (induct b) auto

lemma removeAllListDiff[simp]:
  shows  removeAll  x  (list-diff  a  b)  =  list-diff  (removeAll  x  a)  b
by (induct a) auto

lemma listDiffRemoveAllNonMember:
  assumes  x  \notin  set  a
  shows  list-diff  a  b  =  list-diff  a  (removeAll  x  b)
using  assms

21
proof (induct b arbitrary: a)
case (Cons y b')
  from \( \{ x \notin \text{set } a \} \)
  have \( x \notin \text{set } (\text{removeAll } y \ a) \)
    by auto
  thus \(?\)case
proof (cases \( x = y \))
case False
  thus \(?\)thesis
    using Cons(2)
    using Cons(1)[of removeAll y a]
    using \( \{ x \notin \text{set } (\text{removeAll } y \ a) \} \)
    by auto
next
  case True
  thus \(?\)thesis
    using Cons(1)[of removeAll y a]
    using \( \{ x \notin \text{set } a \} \)
    using \( \{ x \notin \text{set } (\text{removeAll } y \ a) \} \)
    by auto
qed simp

lemma listDiffMap:
  assumes \( \forall \ x \ y. \ x \neq y \rightarrow f \ x \neq f \ y \)
  shows \( \text{map } f \ (\text{list-diff } a \ b) = \text{list-diff } (\text{map } f \ a) \ (\text{map } f \ b) \)
using assms
by (induct b arbitrary: a) (auto simp add: removeAll-map)

1.8 remdups - removing duplicates

lemma remdupsRemoveAllCommute[simp]:
  shows \( \text{remdups } (\text{removeAll } a \ \text{list}) = \text{removeAll } a \ (\text{remdups } \text{list}) \)
by (induct \text{list}) auto

lemma remdupsAppend:
  shows \( \text{remdups } (a @ b) = \text{remdups } (\text{list-diff } a \ b) @ \text{remdups } b \)
proof (induct \text{a})
  case (Cons \( x \ a' \))
  thus \(?\)case
    using listDiffIff[of \( x \ a' \ b \)]
    by auto
qed simp

lemma remdupsAppendSet:
  shows \( \text{set } (\text{remdups } (a @ b)) = \text{set } (\text{remdups } a @ \text{remdups } (\text{list-diff } b \ a)) \)
proof (induct \text{a})
  case Nil
thus \(?\text{case}\)
by \text{auto}

next

\text{case} (\text{Cons} \, x \, a')
thus \(?\text{case}\)

\text{proof} (\text{cases} \, x \in \text{set} \, a')
\text{case} \, \text{True}
thus \(?\text{thesis}\)
using \text{Cons}
using \text{listDiffDoubleRemoveAll}[\text{of} \, x \, a' \, b]
by \text{simp}

next

\text{case} \, \text{False}
thus \(?\text{thesis}\)

\text{proof} (\text{cases} \, x \in \text{set} \, b)
\text{case} \, \text{True}
\text{show} \(?\text{thesis}\)

\text{proof–}

\begin{align*}
\text{have} \ & \text{set} \ (\text{remdups} \ (x \ # \ a') \ @ \ \text{remdups} \ (\text{list-diff} \ b \ (x \ # \ a'))) \\
= \ & \text{set} \ (x \ # \ \text{remdups} \ a' @ \ \text{remdups} \ (\text{list-diff} \ b \ (x \ # \ a'))) \\
\text{using} \ & \ (x \notin \ \text{set} \, a') \\
\text{by} \ & \text{auto} \\
\text{also} \ & \text{have} \ \ldots = \text{set} \ (x \ # \ \text{remdups} \ a' @ \ \text{remdups} \ (\text{list-diff} \ (\text{removeAll} \ x \ b) \ a')) \\
\text{by} \ & \text{auto} \\
\text{also} \ & \text{have} \ \ldots = \text{set} \ (x \ # \ \text{remdups} \ a' @ \ \text{remdups} \ (\text{removeAll} \ x (\text{list-diff} \ b \ a'))) \\
\text{by} \ & \text{simp} \\
\text{also} \ & \text{have} \ \ldots = \text{set} \ (\text{remdups} \ a' @ x \ # \ \text{removeAll} \ x (\text{remdups} (\text{list-diff} \ b \ a'))) \\
\text{by} \ & \text{(simp only: remdupsRemoveAllCommute)} \\
\text{also} \ & \text{have} \ \ldots = \text{set} \ (\text{remdups} \ a') \cup \text{set} \ (x \ # \ \text{removeAll} \ x (\text{remdups} (\text{list-diff} \ b \ a'))) \\
\text{by} \ & \text{simp} \\
\text{also} \ & \text{have} \ \ldots = \text{set} \ (\text{remdups} \ a') \cup \{x\} \cup \text{set} \ (\text{removeAll} \ x (\text{remdups} (\text{list-diff} \ b \ a'))) \\
\text{by} \ & \text{auto} \\
\text{also} \ & \text{have} \ \ldots = \text{set} \ (\text{remdups} \ a') \cup \text{set} \ (\text{remdups} (\text{list-diff} \ b \ a')) \\
\end{align*}

\text{proof–}

\text{from} \ (x \notin \ \text{set} \, a') \ (x \in \ \text{set} \, b)
\text{have} \ x \in \ \text{set} \ (\text{list-diff} \ b \ a')
\text{using} \ \text{listDiffIff}[\text{of} \, x \, b \ a']
\text{by} \ \text{simp}
\text{hence} \ x \in \ \text{set} \ (\text{remdups} \ (\text{list-diff} \ b \ a'))

23
by auto
thus \(?thesis
by auto
qed
also have \dots = set (remdups (a' @ b))
  using Cons(1)
  by simp
also have \dots = set (remdups ((x # a') @ b))
  using (x \in set b)
  by simp
finally show \(?thesis
  by simp
qed
next
case False
thus \(?thesis
proof
  have set (remdups (x # a') @ remdups (list-diff b (x # a')))
  =
    set (x # (remdups a') @ remdups (list-diff b (x # a')))
    using \(x \notin set a'
    by auto
    also have \dots = set (x # remdups a' @ remdups (list-diff (removeAll x b) a'))
    by auto
    also have \dots = set (x # remdups a' @ remdups (list-diff b a'))
    using (x \notin set b)
    by auto
    also have \dots = \{x\} \cup set (remdups (a' @ b))
    using Cons(1)
    by simp
    also have \dots = set (remdups ((x # a') @ b))
    by auto
    finally show \(?thesis
      by simp
    qed
qed

lemma remdupsAppendMultiSet:
  shows multiset-of (remdups (a @ b)) = multiset-of (remdups a @ remdups (list-diff b a))
proof (induct a)
case Nil
  thus \(?case
    by auto
next
case (Cons x a')
thus \( \forall \) case
  proof (\cases x \in \text{set } a')
  \quad \text{case } True
  \quad \text{thus } \forall \text{thesis}
  \quad \text{using } Cons
  \quad \text{using } \text{listDiffDoubleRemoveAll[of } x \text{ a' b]}
  \quad \text{by } \text{simp}
next
\quad \text{case } False
\quad \text{thus } \forall \text{thesis}
  \text{proof (\cases x \in \text{set } b) }
  \quad \text{case } True
  \quad \text{show } \forall \text{thesis}
  \quad \text{proof}
  \quad \quad \text{have } \text{multiset-of } (\text{remdups } (x \# a') @ \text{remdups } (\text{list-diff } b (x \# a')) )
  \quad \quad \quad = \text{multiset-of } (x \# \text{remdups } a' @ \text{remdups } (\text{list-diff } b (x \# a')) )
  \quad \quad \text{proof --}
  \quad \quad \quad \text{have } \text{remdups } (x \# a') = x \# \text{remdups } a'
  \quad \quad \quad \quad \text{using } x \notin \text{set } a'
  \quad \quad \quad \quad \text{by } \text{auto}
  \quad \quad \text{thus } \forall \text{thesis}
  \quad \quad \quad \text{by } \text{simp}
  \quad \quad \text{qed}
  \quad \text{also have } \ldots = \text{multiset-of } (x \# \text{remdups } a' @ \text{remdups } (\text{list-diff } (\text{removeAll } x \text{ b'} a') ) )
  \quad \quad \text{by } \text{auto}
  \quad \text{also have } \ldots = \text{multiset-of } (\text{remdups } a' @ x \# \text{remdups } (\text{removeAll } x \text{ (list-diff } b \text{ a')}) )
  \quad \quad \text{by } \text{simp}
  \quad \text{also have } \ldots = \text{multiset-of } (\text{remdups } a' @ x \# \text{remdups } (\text{list-diff } b \text{ a')})
  \quad \quad \text{by } (\text{simp add: union-assoc})
  \quad \text{also have } \ldots = \text{multiset-of } (\text{remdups } a' @ x \# \text{removeAll } x \text{ (remdups } (\text{list-diff } b \text{ a')}) )
  \quad \quad \text{by } (\text{simp only: remdupsRemoveAllCommutate})
  \quad \text{also have } \ldots = \text{multiset-of } (\text{remdups } a' + \text{multiset-of } (x \# \text{removeAll } x \text{ (remdups } (\text{list-diff } b \text{ a')}) )
  \quad \quad \text{by } \text{simp}
  \quad \text{also have } \ldots = \text{multiset-of } (\text{remdups } a') + \#x\# + \text{multiset-of } (\text{removeAll } x \text{ (remdups } (\text{list-diff } b \text{ a')}) )
  \quad \quad \text{by } (\text{simp add: union-assoc} \text{ simp add: union-commute})
  \quad \text{also have } \ldots = \text{multiset-of } (\text{remdups } a') + \text{multiset-of } (\text{remdups } (\text{list-diff } b \text{ a')})
  \quad \quad \text{proof --}
  \quad \quad \text{from } x \notin \text{ set } a' \forall x \in \text{ set } b
  \quad \quad \text{have } x \in \text{ set } (\text{list-diff } b \text{ a')}
  \quad \quad \text{using } \text{listDiffIff[of } x \text{ b a']}
  \quad \quad \text{by } \text{simp}
hence $x \in \text{set}\ (\text{remdups}\ (\text{list-diff}\ b\ a'))$
  by auto
thus $\theta$thesis
  using $\text{removeAll-multiset}[\text{of}\ \text{remdups}\ (\text{list-diff}\ b\ a')\ x]$
  by (simp\ add: union-assoc)
qed
also have $\ldots = \text{multiset-of}\ (\text{remdups}\ (a' \odot b))$
  using $\text{Cons}(1)$
  by simp
also have $\ldots = \text{multiset-of}\ (\text{remdups}\ ((x \# a') \odot b))$
  using $(x \in \text{set}\ b)$
  by simp
finally show $\theta$thesis
  by simp
qed
next
case False
thus $\theta$thesis
proof −
  have $\text{multiset-of}\ (\text{remdups}\ (x \# a') \odot \text{remdups}\ (\text{list-diff}\ b\ (x \# a'))) =$
    $\text{multiset-of}\ (x \# \text{remdups}\ a' \odot \text{remdups}\ (\text{list-diff}\ b\ (x \# a')))$
    proof −
    have $\text{remdups}\ (x \# a') = x \# \text{remdups}\ a'$
      using $(x \notin \text{set}\ a')$
      by auto
    thus $\theta$thesis
      by simp
    qed
  also have $\ldots = \text{multiset-of}\ (\text{remdups}\ (x \# a') \odot \text{remdups}\ (\text{list-diff}\ (\text{removeAll}\ x\ b)\ a'))$
    by auto
  also have $\ldots = \text{multiset-of}\ (\text{remdups}\ (x \# a') \odot \text{remdups}\ (\text{list-diff}\ b\ a'))$
    using $(x \notin \text{set}\ b)$
    using $\text{removeAll-id}[\text{of}\ x\ b]$
    by simp
  also have $\ldots = \{\#x\}\ + \text{multiset-of}\ (\text{remdups}\ (a' \odot b))$
    using $\text{Cons}(1)$
    by (simp\ add: union-commute)
  also have $\ldots = \text{multiset-of}\ (\text{remdups}\ ((x \# a') \odot b))$
    using $(x \notin \text{set}\ a'\ \&\ x \notin \text{set}\ b)$
    by (auto simp\ add: union-commute)
finally show $\theta$thesis
  by simp
qed
qed
qed

26
lemma remdupsListDiff: 
remdups (list-diff a b) = list-diff (remdups a) (remdups b) 
proof (induct a) 
case Nil 
  thus ?case 
    by simp 
next 
case (Cons x a') 
  thus ?case 
    using listDiffIff [of x a' b] 
    by auto 
qed 

definition 
multiset-le a b r == a = b ∨ (a, b) ∈ mult r 

lemma multisetEmptyLeI: 
assumes 
trans r 
shows 
multiset-le {#} a r 
unfolding multiset-le-def 
using assms 
using one-step-implies-mult[of r a {#} {#}] 
by auto 

lemma multisetUnionLessMono2: 
shows 
trans r ⊢ (b1, b2) ∈ mult r ⊢ (a + b1, a + b2) ∈ mult r 
unfolding mult-def 
apply (erule trancl-induct) 
apply (blast intro: mult1-union transI) 
apply (blast intro: mult1-union transI trancl-trans) 
done 

lemma multisetUnionLessMono1: 
shows 
trans r ⊢ (a1, a2) ∈ mult r ⊢ (a1 + b, a2 + b) ∈ mult r 
using union-commute[of a1 b] 
using union-commute[of a2 b] 
using multisetUnionLessMono2[of r a1 a2 b]
by simp

lemma multisetUnionLeMono2:
  assumes
    trans r
    multiset-le b1 b2 r
  shows
    multiset-le (a + b1) (a + b2) r
  using assms
  unfolding multiset-le-def
  using multisetUnionLessMono2[of r b1 b2 a]
  by auto

lemma multisetUnionLeMono1:
  assumes
    trans r
    multiset-le a1 a2 r
  shows
    multiset-le (a1 + b) (a2 + b) r
  using assms
  unfolding multiset-le-def
  using multisetUnionLessMono1[of r a1 a2 b]
  by auto

lemma multisetLeTrans:
  assumes
    trans r
    multiset-le x y r
    multiset-le y z r
  shows
    multiset-le x z r
  using assms
  unfolding multiset-le-def
  unfolding mult-def
  by (blast intro: trancl-trans)

lemma multisetUnionLeMono:
  assumes
    trans r
    multiset-le a1 a2 r
    multiset-le b1 b2 r
  shows
    multiset-le (a1 + b1) (a2 + b2) r
  using assms
  using multisetUnionLeMono1[of r a1 a2 b1]
  using multisetUnionLeMono2[of r b1 b2 a2]
  using multisetLeTrans[of r a1 + b1 a2 + b1 a2 + b2]
lemma multisetLeListDiff:
assumes
  trans r
shows
  multiset-le (multiset-of (list-diff a b)) (multiset-of a) r
proof (induct a)
case Nil
  thus ?case
    unfolding multiset-le-def
    by simp
next
case (Cons x a')
  thus ?case
    using assms
    using multisetEmptyLeI[of r \{#x#\}]
    using multisetUnionLeMono[of r multiset-of (list-diff a' b) multiset-of a' \{##\}]
    using multisetUnionLeMonoI[of r multiset-of (list-diff a' b) multiset-of a' \{##\}]
    by auto
qed

1.9 Levi’s lemma

Obsolete: these two lemmas are already proved as append-eq-append-conv2
and append-eq-Cons-conv.

lemma FullLevi:
  shows (x @ y = z @ w) =
  (x = z ∧ y = w ∨
   (∃ t. z @ t = x ∧ t @ y = w) ∨
   (∃ t. x @ t = z ∧ t @ w = y)) (is ?lhs = ?rhs)
proof
  assume ?rhs
  thus ?lhs
    by auto
next
  assume ?lhs
  thus ?rhs
  proof (induct x arbitrary: z)
    case (Cons a x')
    show ?case
      proof
        cases z = []
        case True
          with (a # x') @ y = z @ w
          obtain t where z @ t = a # x' t @ y = w
            by auto
          thus ?thesis
  qed

29
by auto
next
case False
then obtain b and z’ where z = b # z’
  by (auto simp add: neq-Nil-conv)
with (a # x’) @ y = z @ w;
have x’ @ y = z’ @ w a = b
  by auto
with Cons(1)[of z’]
have x’ = z’ ∧ y = w ∨ (∃t. z’ @ t = x’ ∧ t @ y = w) ∨ (∃t. x’ @ t = z’ ∧ t @ w = y)
  by simp
with (a = b); z = b # z’
show ?thesis
by auto
qed
qed simp

lemma SimpleLevi:
  shows (p @ s = a # list) =
    (p = [] ∧ s = a # list ∨
    (∃t. p = a # t ∧ t @ s = list))
by (induct p) auto

1.10 Single element lists

lemma lengthOneCharacterisation:
  shows (length l = 1) = (l = [hd l])
by (induct l) auto

lemma lengthOneImpliesOnlyElement:
  assumes length l = 1 and a : set l
  shows ∀a’, a’: set l → a’ = a
proof (cases l)
case (Cons literal’ clause’)
  with assms
  show ?thesis
  by auto
qed simp

end

2 CNF

theory CNF
imports MoreList
begin
Theory describing formulae in Conjunctive Normal Form.

2.1 Syntax

2.1.1 Basic datatypes

type-synonym Variable = nat
datatype Literal = Pos Variable | Neg Variable
type-synonym Clause = Literal list
type-synonym Formula = Clause list

Notice that instead of set or multisets, lists are used in definitions of clauses and formulae. This is done because SAT solver implementation usually use list-like data structures for representing these datatypes.

2.1.2 Membership

Check if the literal is member of a clause, clause is a member of a formula or the literal is a member of a formula

corns member :: 'a ⇒ 'b ⇒ bool (infixl el 55)
defs (overloaded)
literalElClause-def [simp]: ((literal::Literal) el (clause::Clause)) == literal ∈ set clause
defs (overloaded)
clauseElFormula-def [simp]: ((clause::Clause) el (formula::Formula)) == clause ∈ set formula

overloading
el-literal ≡ op el :: Literal ⇒ Formula ⇒ bool
begin
primrec el-literal where
(literal::Literal) el ([])::Formula) = False |
((literal::Literal) el ((clause # formula)::Formula)) = ((literal el clause) ∨ (literal el formula))
end

lemma literalElFormulaCharacterization:
  fixes literal :: Literal and formula :: Formula
  shows (literal el formula) = (∃ (clause::Clause). clause el formula ∧ literal el clause)
by (induct formula) auto
2.1.3 Variables

The variable of a given literal

```
primrec
var :: Literal ⇒ Variable
where
  var (Pos v) = v
  | var (Neg v) = v
```

Set of variables of a given clause, formula or valuation

```
primrec
varsClause :: (Literal list) ⇒ (Variable set)
where
  varsClause [] = {}
  | varsClause (literal # list) = {var literal} ∪ (varsClause list)
```

```
primrec
varsFormula :: Formula ⇒ (Variable set)
where
  varsFormula [] = {}
  | varsFormula (clause # formula) = (varsClause clause) ∪ (varsFormula formula)
```

```
consts vars :: 'a ⇒ Variable set
defs (overloaded)
  vars-def-clause [simp]: vars (clause::Clause) == varsClause clause
  vars-def-formula [simp]: vars (formula::Formula) == varsFormula formula
  vars-def-set [simp]: vars (s::Literal set) == {vbl. ∃ l. l ∈ s ∧ var l = vbl}
```

```
lemma clauseContainsItsLiteralsVariable:
  fixes literal :: Literal and clause :: Clause
  assumes literal el clause
  shows var literal ∈ vars clause
using assms
by (induct clause) auto
```

```
lemma formulaContainsItsLiteralsVariable:
  fixes literal :: Literal and formula::Formula
  assumes literal el formula
  shows var literal ∈ vars formula
using assms
proof (induct formula)
  case Nil
  thus ?case
    by simp
next
  case (Cons clause formula)
```
thus \(?case\)
proof (cases literal el clause)
  case True
  with clauseContainsItsLiteralsVariable
  have \(\operatorname{var}\ \operatorname{literal} \in \operatorname{vars}\ \operatorname{clause}\)
    by simp
  thus \(?thesis\)
    by simp
next
  case False
  with Cons
  show \(?thesis\)
    by simp
qed
qed

lemma formulaContainsItsClausesVariables:
fixes \(\operatorname{clause} :: \operatorname{Clause}\) and \(\operatorname{formula} :: \operatorname{Formula}\)
assumes \(\operatorname{clause} \in \operatorname{formula}\)
shows \(\operatorname{vars}\ \operatorname{clause} \subseteq \operatorname{vars}\ \operatorname{formula}\)
using assms
by (induct \operatorname{formula}) auto

lemma varsAppendFormulae:
fixes \(\operatorname{formula1} :: \operatorname{Formula}\) and \(\operatorname{formula2} :: \operatorname{Formula}\)
shows \(\operatorname{vars}\ (\operatorname{formula1} @ \operatorname{formula2}) = \operatorname{vars}\ \operatorname{formula1} \cup \operatorname{vars}\ \operatorname{formula2}\)
by (induct \operatorname{formula1}) auto

lemma varsAppendClauses:
fixes \(\operatorname{clause1} :: \operatorname{Clause}\) and \(\operatorname{clause2} :: \operatorname{Clause}\)
shows \(\operatorname{vars}\ (\operatorname{clause1} @ \operatorname{clause2}) = \operatorname{vars}\ \operatorname{clause1} \cup \operatorname{vars}\ \operatorname{clause2}\)
by (induct \operatorname{clause1}) auto

lemma varsRemoveLiteral:
fixes \(\operatorname{literal} :: \operatorname{Literal}\) and \(\operatorname{clause} :: \operatorname{Clause}\)
shows \(\operatorname{vars}\ \operatorname{removeAll}\ \operatorname{literal}\ \operatorname{clause} \subseteq \operatorname{vars}\ \operatorname{clause}\)
by (induct \operatorname{clause}) auto

lemma varsRemoveLiteralSuperset:
fixes \(\operatorname{literal} :: \operatorname{Literal}\) and \(\operatorname{clause} :: \operatorname{Clause}\)
shows \(\operatorname{vars}\ \operatorname{clause} - \{\operatorname{var}\ \operatorname{literal}\} \subseteq \operatorname{vars}\ \operatorname{removeAll}\ \operatorname{literal}\ \operatorname{clause}\)
by (induct \operatorname{clause}) auto

lemma varsRemoveAllClause:
fixes \(\operatorname{clause} :: \operatorname{Clause}\) and \(\operatorname{formula} :: \operatorname{Formula}\)
shows \(\operatorname{vars}\ \operatorname{removeAll}\ \operatorname{clause}\ \operatorname{formula} \subseteq \operatorname{vars}\ \operatorname{formula}\)
by (induct \operatorname{formula}) auto

lemma varsRemoveAllClauseSuperset:
fixes clause :: Clause and formula :: Formula
shows \(\text{vars formula} - \text{vars clause} \subseteq \text{vars} (\text{removeAll} \text{ clause formula})\)
by (induct formula) auto

lemma varInClauseVars:
fixes variable :: Variable and clause :: Clause
shows \(\text{variable} \in \text{vars clause} = (\exists \text{ literal}. \text{ literal el clause} \land \text{ var literal} = \text{ variable})\)
by (induct clause) auto

lemma varInFormulaVars:
fixes variable :: Variable and formula :: Formula
shows \(\text{variable} \in \text{vars formula} = (\exists \text{ literal}. \text{ literal el formula} \land \text{ var literal} = \text{ variable})\)
(proof (induct formula)
next
show ?thesis
proof
assume \(P \: ?\text{lhs} \: (\text{clause} \# \text{ formula})\)
thus \(\text{rhs} \: (\text{clause} \# \text{ formula})\)
(proof (cases \text{variable} \in \text{vars clause})
next
assume \(\text{variable} \in \text{vars formula}\)
by (simp)
with \(\text{Cons}\)
show \(?\text{thesis}\)
by (auto)
qed

next
assume \(?\text{rhs} \: (\text{clause} \# \text{ formula})\)
then obtain \(l\)
where \(\text{IEl: l el clause} \# \text{ formula} \: \text{and} \: \text{varL: var l} = \text{ variable}\)
by (auto)
from \(\text{IEl formulaContainsItsLiteralsVariable} \: |\: \text{of l clause} \# \text{ formula}|\)
have \(\text{var l} \in \text{vars} (\text{clause} \# \text{ formula})\)
by (auto)

34
lemma varsSubsetFormula:
  fixes F :: Formula and F' :: Formula
  assumes ∀ c::Clause. c ∈ F → c ∈ F'
  shows vars F ⊆ vars F'
using assms
proof (induct F)
case Nil
  thus ?case by simp
next
case (Cons c' F'')
  thus ?case using formulaContainsItsClausesVariables[of c' F']
    by simp
qed

lemma varsClauseVarsSet:
  fixes clause :: Clause
  shows vars clause = vars (set clause)
by (induct clause) auto

2.1.4 Opposite literals

primrec
opposite :: Literal ⇒ Literal
where
  opposite (Pos v) = (Neg v)
| opposite (Neg v) = (Pos v)

lemma oppositeIdempotency [simp]:
  fixes literal::Literal
  shows opposite (opposite literal) = literal
by (induct literal) auto

lemma oppositeSymmetry [simp]:
  fixes literal1::Literal and literal2::Literal
  shows (opposite literal1 = literal2) = (opposite literal2 = literal1)
by auto

lemma oppositeUniqueness [simp]:
  fixes literal1::Literal and literal2::Literal
\begin{verbatim}
shows \((\text{opposite literal1} = \text{opposite literal2}) = (\text{literal1} = \text{literal2})\)
proof
  assume \(\text{opposite literal1} = \text{opposite literal2}\)
  hence \(\text{opposite (opposite literal1)} = \text{opposite (opposite literal2)}\)
    by simp
  thus \(\text{literal1} = \text{literal2}\)
    by simp
qed simp

lemma \text{oppositeIsDifferentFromLiteral} [simp]:
  fixes \text{literal} :: Literal
  shows \(\text{opposite literal} \neq \text{literal}\)
by (induct literal) auto

lemma \text{oppositeLiteralsHaveSameVariable} [simp]:
  fixes \text{literal} :: Literal
  shows \(\text{var (opposite literal)} = \text{var literal}\)
by (induct literal) auto

lemma \text{literalsWithSameVariableAreEqualOrOpposite}:
  fixes \text{literal1} :: Literal and \text{literal2} :: Literal
  shows \((\text{var literal1} = \text{var literal2}) = (\text{literal1} = \text{literal2} \lor \text{opposite literal1} = \text{literal2})\)
(is \(?\text{lhs} = ?\text{rhs}\))
proof
  assume \(?\text{lhs}\)
  show \(?\text{rhs}\)
  proof
    (cases \text{literal1})
    case \text{Pos}
    note \text{Pos1} = this
    show \(?\text{thesis}\)
    proof
      (cases \text{literal2})
      case \text{Pos}
      with \((?\text{lhs}) \text{Pos1}\) show \(?\text{thesis}\)
        by simp
      next
      case \text{Neg}
      with \((?\text{lhs}) \text{Pos1}\) show \(?\text{thesis}\)
        by simp
    qed
  next
  case \text{Neg}
  note \text{Neg1} = this
  show \(?\text{thesis}\)
  proof
    (cases \text{literal2})
    case \text{Pos}
    with \((?\text{lhs}) \text{Neg1}\) show \(?\text{thesis}\)
      by simp
    next
    case \text{Neg}
  qed
\end{verbatim}
with (?lhs) \texttt{Neg1} show \texttt{?thesis}
by simp
qed
qed
next
assume ?rhs
thus \texttt{?lhs}
by auto
qed

The list of literals obtained by negating all literals of a literal list (clause, valuation). Notice that this is not a negation of a clause, because the negation of a clause is a conjunction and not a disjunction.

definition
\texttt{oppositeLiteralList} :: \texttt{Literal list} \Rightarrow \texttt{Literal list}
where
\texttt{oppositeLiteralList} clause =\texttt{map \texttt{opposite}} clause

lemma \texttt{literalElListIffOppositeLiteralElOppositeLiteralList}:
fixes \texttt{literal} :: \texttt{Literal} and \texttt{literalList} :: \texttt{Literal list}
shows \texttt{literal el literalList} = \texttt{(opposite literal) el (oppositeLiteralList literalList)}
unfolding \texttt{oppositeLiteralList-def}
proof (induct \texttt{literalList})
case \texttt{Nil}
thus \texttt{?case}
by simp
next
case (\texttt{Cons \texttt{l} \texttt{literalList}})
show \texttt{?case}
proof (cases \texttt{l = literal})
case \texttt{True}
thus \texttt{?thesis}
by simp
next
case \texttt{False}
thus \texttt{?thesis}
by auto
qed
qed

lemma \texttt{oppositeLiteralListIdempotency} [simp]:
fixes \texttt{literalList :: Literal list}
shows \texttt{oppositeLiteralList (oppositeLiteralList literalList) = literalList}
unfolding \texttt{oppositeLiteralList-def}
by (induct \texttt{literalList}) auto

lemma \texttt{oppositeLiteralListRemove}:
fixes literal :: Literal and literalList :: Literal list
shows oppositeLiteralList (removeAll literal literalList) = removeAll
(opposite literal) (oppositeLiteralList literalList)
unfolding oppositeLiteralList-def
by (induct literalList) auto

lemma oppositeLiteralListNonempty:
fixes literalList :: Literal list
shows (literalList ≠ []) = ((oppositeLiteralList literalList) ≠ [])
unfolding oppositeLiteralList-def
by (induct literalList) auto

lemma varsOppositeLiteralList:
shows vars (oppositeLiteralList clause) = vars clause
unfolding oppositeLiteralList-def
by (induct clause) auto

2.1.5 Tautological clauses

Check if the clause contains both a literal and its opposite

primrec
clauseTautology :: Clause ⇒ bool
where
clauseTautology [] = False
| clauseTautology (literal # clause) = (opposite literal el clause ∨
clauseTautology clause)

lemma clauseTautologyCharacterization:
fixes clause :: Clause
shows clauseTautology clause = (∃ literal. literal el clause ∧ (opposite
literal) el clause)
by (induct clause) auto

2.2 Semantics

2.2.1 Valuations

type-synonym Valuation = Literal list

lemma valuationContainsItsLiteralsVariable:
fixes literal :: Literal and valuation :: Valuation
assumes literal el valuation
shows var literal ∈ vars valuation
using assms
by (induct valuation) auto

lemma varsSubsetValuation:
fixes valuation1 :: Valuation and valuation2 :: Valuation
assumes set valuation1 ⊆ set valuation2

38
shows \( \text{vars valuation1} \subseteq \text{vars valuation2} \)
using \( \text{assms} \)
proof (induct valuation1)
case Nil
show ?case
by simp
next
case (Cons literal valuation)
ote caseCons = this
hence literal el valuation2
by auto
with valuationContainsItsLiteralsVariable [of literal valuation2]
have var literal \( \in \) vars valuation2 .
with caseCons
show ?case
by simp
qed

lemma varsAppendValuation:
fixes valuation1 :: Valuation and valuation2 :: Valuation
shows \( \text{vars } (\text{valuation1 @ valuation2}) = \text{vars valuation1} \cup \text{vars valuation2} \)
by (induct valuation1) auto

lemma varsPrefixValuation:
fixes valuation1 :: Valuation and valuation2 :: Valuation
assumes isPrefix valuation1 valuation2
shows \( \text{vars valuation1} \subseteq \text{vars valuation2} \)
proof
from \( \text{assms} \)
have set valuation1 \( \subseteq \) set valuation2
by (auto simp add:isPrefix-def)
thus ?thesis
by (rule varsSubsetValuation)
qed

2.2.2 True/False literals

Check if the literal is contained in the given valuation

definition literalTrue :: Literal \( \Rightarrow \) Valuation \( \Rightarrow \) bool
where
literalTrue-def [simp]: literalTrue literal valuation \( = \) literal el valuation

Check if the opposite literal is contained in the given valuation

definition literalFalse :: Literal \( \Rightarrow \) Valuation \( \Rightarrow \) bool
where
literalFalse-def [simp]: literalFalse literal valuation \( = \) opposite literal el valuation
lemma variableDefinedImpliesLiteralDefined:
  fixes literal :: Literal and valuation :: Valuation
  shows var literal ∈ vars valuation = (literalTrue literal valuation ∨
  literalFalse literal valuation)
  (is (?lhs valuation) = (?rhs valuation))
proof
assume ?rhs valuation
thus ?lhs valuation
proof
  assume literalTrue literal valuation
  hence literal ∈ valuation
  by simp
  thus ?thesis
  using valuationContainsItsLiteralsVariable[of literal valuation]
  by simp
next
  assume literalFalse literal valuation
  hence opposite literal ∈ valuation
  by simp
  thus ?thesis
  using valuationContainsItsLiteralsVariable[of opposite literal valuation]
  by simp
qed
next
assume ?lhs valuation
thus ?rhs valuation
proof (induct valuation)
case Nil
  thus ?case
  by simp
next
case (Cons literal′ valuation′)
  note ih=this
  show ?case
  proof (cases var literal ∈ vars valuation′)
    case True
    with ih
    show ?rhs (literal′ ≠ valuation′)
      by auto
  next
    case False
    with ih
    have var literal′ = var literal
      by simp
    hence literal′ = literal ∨ opposite literal′ = literal
      by (simp add:literalsWithSameVariableAreEqualOrOpposite)
    thus ?rhs (literal′ ≠ valuation′)
2.2.3 True/False clauses

Check if there is a literal from the clause which is true in the given valuation

primrec clauseTrue :: Clause ⇒ Valuation ⇒ bool
where
  clauseTrue [] valuation = False
  | clauseTrue (literal # clause) valuation = (literalTrue literal valuation
    ∨ clauseTrue clause valuation)

Check if all the literals from the clause are false in the given valuation

primrec clauseFalse :: Clause ⇒ Valuation ⇒ bool
where
  clauseFalse [] valuation = True
  | clauseFalse (literal # clause) valuation = (literalFalse literal valuation
    ∧ clauseFalse clause valuation)

lemma clauseTrueIffContainsTrueLiteral:
  fixes clause :: Clause and valuation :: Valuation
  shows clauseTrue clause valuation = (∃ literal. literal el clause ∧
    literalTrue literal valuation)
by (induct clause) auto

lemma clauseFalseIffAllLiteralsAreFalse:
  fixes clause :: Clause and valuation :: Valuation
  shows clauseFalse clause valuation = (∀ literal. literal el clause →
    literalFalse literal valuation)
by (induct clause) auto

lemma clauseFalseRemove:
  assumes clauseFalse clause valuation
  shows clauseFalse (removeAll literal clause) valuation
proof–
  { fix l::Literal
    assume l el removeAll literal clause
    hence l el clause
    by simp
    with (clauseFalse clause valuation)
have \( \text{literalFalse } l \text{ valuation} \)
    by (simp add:clauseFalseIffAllLiteralsAreFalse)
}
thus \(?thesis\)
    by (simp add:clauseFalseIffAllLiteralsAreFalse)
qed

lemma clauseFalseAppendValuation:
  fixes clause :: Clause and valuation :: Valuation and valuation' :: Valuation
  assumes clauseFalse clause valuation
  shows clauseFalse clause (valuation @ valuation')
  using assms
  by (induct clause) auto

lemma clauseTrueAppendValuation:
  fixes clause :: Clause and valuation :: Valuation and valuation' :: Valuation
  assumes clauseTrue clause valuation
  shows clauseTrue clause (valuation @ valuation')
  using assms
  by (induct clause) auto

lemma emptyClauseIsFalse:
  fixes valuation :: Valuation
  shows clauseFalse [] valuation
  by auto

lemma emptyValuationFalsifiesOnlyEmptyClause:
  fixes clause :: Clause
  assumes clause \(\neq [\] \)
  shows \(\neg\) clauseFalse clause []
  using assms
  by (induct clause) auto

lemma valuationContainsItsFalseClausesVariables:
  fixes clause::Clause and valuation::Valuation
  assumes clauseFalse clause valuation
  shows vars clause \(\subseteq\) vars valuation
proof
  fix v::Variable
  assume v \(\in\) vars clause
  hence \(\exists\) l. var l = v \(\land\) l el clause
    by (induct clause) auto
  then obtain l
    where var l = v l el clause
    by auto
  from \(l\ el\ clause\) (clauseFalse clause valuation)

42
have literalFalse l valuation
by (simp add: clauseFalseIffAllLiteralsAreFalse)
with (var l = v)
show v ∈ vars valuation
using valuationContainsItsLiteralsVariable[of opposite l]
by simp
qed

2.2.4 True/False formulae

Check if all the clauses from the formula are false in the given valuation

primrec
formulaTrue :: Formula ⇒ Valuation ⇒ bool
where
  formulaTrue [] valuation = True
| formulaTrue (clause # formula) valuation = (clauseTrue clause valuation ∧ formulaTrue formula valuation)

Check if there is a clause from the formula which is false in the given valuation

primrec
formulaFalse :: Formula ⇒ Valuation ⇒ bool
where
  formulaFalse [] valuation = False
| formulaFalse (clause # formula) valuation = (clauseFalse clause valuation ∨ formulaFalse formula valuation)

lemma formulaTrueIffAllClausesAreTrue:
  fixes formula :: Formula and valuation :: Valuation
  shows formulaTrue formula valuation = (∀ clause. clause el formula → clauseTrue clause valuation)
  by (induct formula) auto

lemma formulaFalseIffContainsFalseClause:
  fixes formula :: Formula and valuation :: Valuation
  shows formulaFalse formula valuation = (∃ clause. clause el formula ∧ clauseFalse clause valuation)
  by (induct formula) auto

lemma formulaTrueAssociativity:
  fixes f1 :: Formula and f2 :: Formula and f3 :: Formula and valuation :: Valuation
  shows formulaTrue ((f1 @ f2) @ f3) valuation = formulaTrue (f1 @ (f2 @ f3)) valuation
  by (auto simp add: formulaTrueIffAllClausesAreTrue)
lemma formulaTrueCommutativity:
  fixes f1 :: Formula and f2 :: Formula and valuation :: Valuation
  shows formulaTrue (f1 @ f2) valuation = formulaTrue (f2 @ f1) valuation
by (auto simp add:formulaTrueIffAllClausesAreTrue)

lemma formulaTrueSubset:
  fixes formula :: Formula and formula' :: Formula and valuation :: Valuation
  assumes formulaTrue: formulaTrue formula valuation and
  subset: \( \forall \langle \text{clause} :: \text{Clause} \rangle. \text{clause el formula'} \rightarrow \text{clause el formula} \)
  shows formulaTrue formula' valuation
proof -
  { fix clause :: Clause
    assume clause el formula'
    with formulaTrue subset
    have clauseTrue clause valuation
      by (simp add:formulaTrueIffAllClausesAreTrue)
  }
  thus \(?thesis\)
  by (simp add:formulaTrueIffAllClausesAreTrue)
qed

lemma formulaTrueAppend:
  fixes formula1 :: Formula and formula2 :: Formula and valuation :: Valuation
  shows formulaTrue (formula1 @ formula2) valuation = (formulaTrue formula1 valuation \&\& formulaTrue formula2 valuation)
by (induct formula1) auto

lemma formulaTrueRemoveAll:
  fixes formula :: Formula and clause :: Clause and valuation :: Valuation
  assumes formulaTrue formula valuation
  shows formulaTrue (removeAll clause formula) valuation
using assms
by (induct formula) auto

lemma formulaFalseAppend:
  fixes formula :: Formula and formula' :: Formula and valuation :: Valuation
  assumes formulaFalse formula valuation
  shows formulaFalse (formula @ formula') valuation
using assms
by (induct formula) auto

lemma formulaTrueAppendValuation:

44
fixes formula :: Formula and valuation :: Valuation and valuation' :: Valuation

assumes formulaTrue formula valuation

shows formulaTrue formula (valuation @ valuation')

using assms

by (induct formula) (auto simp add: clauseTrueAppendValuation)

lemma formulaFalseAppendValuation:

fixes formula :: Formula and valuation :: Valuation and valuation' :: Valuation

assumes formulaFalse formula valuation

shows formulaFalse formula (valuation @ valuation')

using assms

by (induct formula) (auto simp add: clauseFalseAppendValuation)

lemma trueFormulaWithSingleLiteralClause:

fixes formula :: Formula and literal :: Literal and valuation :: Valuation

assumes formulaTrue (removeAll [literal] formula) (valuation @ [literal])

shows formulaTrue formula (valuation @ [literal])

proof –

{ fix clause :: Clause
  assume clause el formula
  with assms
  have clauseTrue clause (valuation @ [literal])
  proof (cases clause = [literal])
    case True
    thus ?thesis
    by simp
  next
    case False
    with (clause el formula)
    have clause el (removeAll [literal] formula)
    by simp
    with (formulaTrue (removeAll [literal] formula) (valuation @ [literal]))
    show ?thesis
    by (simp add: formulaTrueIffAllClausesAreTrue)
  qed

  thus ?thesis
  by (simp add: formulaTrueIffAllClausesAreTrue)

  qed

  thus ?thesis
  by (simp add: formulaTrueIffAllClausesAreTrue)

  qed
2.2.5 Valuation viewed as a formula

Converts a valuation (the list of literals) into formula (list of single member lists of literals)

primrec
val2form :: Valuation ⇒ Formula
where
val2form [] = []
| val2form (literal # valuation) = [literal] # val2form valuation

lemma val2FormEl:
  fixes literal :: Literal and valuation :: Valuation
  shows literal el valuation = [literal] el val2form valuation
  by (induct valuation) auto

lemma val2FormAreSingleLiteralClauses:
  fixes clause :: Clause and valuation :: Valuation
  shows clause el valuation −→ (∃ literal. clause = [literal] ∧ literal el valuation)
  by (induct valuation) auto

lemma val2formOfSingleLiteralValuation:
  assumes length v = 1
  shows val2form v = [[hd v]]
  using assms
  by (induct v) auto

lemma val2formRemoveAll:
  fixes literal :: Literal and valuation :: Valuation
  shows removeAll [literal] (val2form valuation) = val2form (removeAll literal valuation)
  by (induct valuation) auto

lemma val2formAppend:
  fixes valuation1 :: Valuation and valuation2 :: Valuation
  shows val2form (valuation1 @ valuation2) = (val2form valuation1 @ val2form valuation2)
  by (induct valuation1) auto

lemma val2formFormulaTrue:
  fixes valuation1 :: Valuation and valuation2 :: Valuation
  shows formulaTrue (val2form valuation1) valuation2 = (∀ (literal :: Literal). literal el valuation1 −→ literal el valuation2)
  by (induct valuation1) auto

2.2.6 Consistency of valuations

Valuation is inconsistent if it contains both a literal and its opposite.
primrec

inconsistent :: Valuation ⇒ bool

where

inconsistent [] = False
| inconsistent (literal # valuation) = (opposite literal el valuation ∨ inconsistent valuation)

definition [simp]: consistent valuation == ¬ inconsistent valuation

lemma inconsistentCharacterization:

fixes valuation :: Valuation

shows inconsistent valuation = (∃ literal. literalTrue literal valuation ∧ literalFalse literal valuation)

by (induct valuation) auto

lemma clauseTrueAndClauseFalseImpliesInconsistent:

fixes clause :: Clause and valuation :: Valuation

assumes clauseTrue clause valuation and clauseFalse clause valuation

shows inconsistent valuation

proof −

from ⟨clauseTrue clause valuation⟩ obtain literal :: Literal

where literal el clause and literalTrue literal valuation

by (auto simp add: clauseTrueIffContainsTrueLiteral)

with ⟨clauseFalse clause valuation⟩

have literalFalse literal valuation

by (auto simp add: clauseFalseIffAllLiteralsAreFalse)

from ⟨literalTrue literal valuation⟩ ⟨literalFalse literal valuation⟩

show ?thesis

by (auto simp add: inconsistentCharacterization)

qed

lemma formulaTrueAndFormulaFalseImpliesInconsistent:

fixes formula :: Formula and valuation :: Valuation

assumes formulaTrue formula valuation and formulaFalse formula valuation

shows inconsistent valuation

proof −

from ⟨formulaFalse formula valuation⟩ obtain clause :: Clause

where clause el formula and clauseFalse clause valuation

by (auto simp add: formulaFalseIffContainsFalseClause)

with ⟨formulaTrue formula valuation⟩

have clauseTrue clause valuation

by (auto simp add: formulaTrueIffAllClausesAreTrue)

from ⟨clauseTrue clause valuation⟩ ⟨clauseFalse clause valuation⟩

show ?thesis

by (auto simp add: clauseTrueAndClauseFalseImpliesInconsistent)

qed

lemma inconsistentAppend:

fixes valuation1 :: Valuation and valuation2 :: Valuation
assumes inconsistent (valuation1 @ valuation2)
shows inconsistent valuation1 ∨ inconsistent valuation2 ∨ (∃ literal. literalTrue literal valuation1 ∧ literalFalse literal valuation2)
using assms
proof (cases inconsistent valuation1)
  case True
  thus ?thesis
  by simp
next
  case False
  thus ?thesis
proof (cases inconsistent valuation2)
  case True
  thus ?thesis
  by simp
next
  case False
from inconsistent (valuation1 @ valuation2) obtain literal :: Literal
where literalTrue literal (valuation1 @ valuation2) and literalFalse literal (valuation1 @ valuation2)
by (auto simp add: inconsistentCharacterization)
hence (∃ literal. literalTrue literal valuation1 ∧ literalFalse literal valuation2)
proof (cases literalTrue literal valuation1)
  case True
  with ¬ inconsistent valuation1
  have ¬ literalFalse literal valuation1
  by (auto simp add: inconsistentCharacterization)
  with (literalFalse literal (valuation1 @ valuation2))
  have literalFalse literal valuation2
  by auto
  with True
  show ?thesis
  by auto
next
  case False
  with (literalTrue literal (valuation1 @ valuation2))
  have literalTrue literal valuation2
  by auto
  with ¬ inconsistent valuation2
  have ¬ literalFalse literal valuation2
  by (auto simp add: inconsistentCharacterization)
  with (literalFalse literal (valuation1 @ valuation2))
  have literalFalse literal valuation1
  by auto
  with (literalTrue literal valuation2)
  show ?thesis
  by auto

48
\textbf{lemma} consistentAppendElement:
\textbf{assumes} consistent \( v \) \textbf{and} \( \neg \) literalFalse \( l \) \( v \)
\textbf{shows} consistent \( (v @ [l]) \)
\textbf{proof} -
\{ 
\textbf{assume} \( \neg \) \( \text{thesis} \)
\textbf{with} \( \) (consistent \( v \)) 
\textbf{have} (opposite \( l \)) \( el \) \( v \)
\textbf{using} inconsistentAppend\[of \ v \ [l] \]
\textbf{by} auto
\textbf{with} \( \neg \) literalFalse \( l \) \( v \)
\textbf{have} False
\textbf{by} simp
\}
\textbf{thus} \( \text{thesis} \)
\textbf{by} auto
\textbf{qed}

\textbf{lemma} inconsistentRemoveAll:
\textbf{fixes} literal :: Literal \textbf{and} valuation :: Valuation
\textbf{assumes} inconsistent \( (\text{removeAll literal valuation}) \)
\textbf{shows} inconsistent valuation
\textbf{using} \textbf{assms}
\textbf{proof} -
\textbf{from} \( \) \( \) \( \) \( \text{inconsistent} \) \( (\text{removeAll literal valuation}) \) \textbf{obtain} literal' :: Literal
\textbf{where} \( l'\)True: literalTrue \( l'\) (removeAll literal valuation) \textbf{and}
\( l'\)False: literalFalse \( l'\) (removeAll literal valuation)
\textbf{by} (auto simp add:inconsistentCharacterization)
\textbf{from} \( l'\)True
\textbf{have} literalTrue \( l'\) \textbf{valuation}
\textbf{by} simp
\textbf{moreover}
\textbf{from} \( l'\)False
\textbf{have} literalFalse \( l'\) \textbf{valuation}
\textbf{by} simp
\textbf{ultimately}
\textbf{show} \( \text{thesis} \)
\textbf{by} (auto simp add:inconsistentCharacterization)
\textbf{qed}

\textbf{lemma} inconsistentPrefix:
\textbf{assumes} isPrefix \( \text{valuation1} \) \textbf{valuation2} \textbf{and} inconsistent \( \text{valuation1} \)
shows inconsistent valuation2 using assms by (auto simp add: inconsistentCharacterization isPrefix-def)

lemma consistentPrefix:
  assumes isPrefix valuation1 valuation2 and consistent valuation2
  shows consistent valuation1
  using assms
  by (auto simp add: inconsistentCharacterization isPrefix-def)

2.2.7 Totality of valuations

Checks if the valuation contains all the variables from the given set of variables

definition total where
[simp]: total valuation variables == variables ⊆ vars valuation

lemma totalSubset:
  fixes A :: Variable set and B :: Variable set and valuation :: Valuation
  assumes A ⊆ B and total valuation B
  shows total valuation A
  using assms
  by auto

lemma totalFormulaImpliesTotalClause:
  fixes clause :: Clause and formula :: Formula and valuation :: Valuation
  assumes clauseEl: clause el formula and totalFormula: total valuation (vars formula)
  shows totalClause: total valuation (vars clause)
  proof =
    from clauseEl
    have vars clause ⊆ vars formula
      using formulaContainsItsClausesVariables [of clause formula]
      by simp
    with totalFormula
    show ?thesis
      by (simp add: totalSubset)
  qed

lemma totalValuationForClauseDefinesAllItsLiterals:
  fixes clause :: Clause and valuation :: Valuation and literal :: Literal
  assumes totalClause: total valuation (vars clause) and
          literalEl: literal el clause
  shows trueOrFalse: literalTrue literal valuation ∨ literalFalse literal valuation
  proof =
from literalEl
have var literal ∈ vars clause
  using clauseContainsItsLiteralsVariable
  by auto
with totalClause
have var literal ∈ vars valuation
  by auto
thus ?thesis
  using variableDefinedImpliesLiteralDefined [of literal valuation]
  by simp
qed

lemma totalValuationForClauseDefinesItsValue:
  fixes clause :: Clause and valuation :: Valuation
  assumes totalClause: total valuation (vars clause)
  shows clauseTrue clause valuation ∨ clauseFalse clause valuation
proof (cases clauseFalse clause valuation)
case True
  thus ?thesis
  by (rule disjI2)
next
case False
  hence ¬ (∀ l. l el clause → literalFalse l valuation)
  by (auto simp add:clauseFalseIffAllLiteralsAreFalse)
then obtain l :: Literal
  where l el clause and ¬ literalFalse l valuation
  by auto
with totalClause
have literalTrue l valuation ∨ literalFalse l valuation
  using totalValuationForClauseDefinesAllItsLiterals [of valuation clause l]
  by auto
with (¬ literalFalse l valuation)
have literalTrue l valuation
  by simp
with (l el clause)
have (clauseTrue clause valuation)
  by (auto simp add:clauseTrueIffContainsTrueLiteral)
thus ?thesis
  by (rule disjI1)
qed

lemma totalValuationForFormulaDefinesAllItsLiterals:
  fixes formula::Formula and valuation::Valuation
  assumes totalFormula: total valuation (vars formula) and
    literalElFormula: literal el formula
  shows literalTrue literal valuation ∨ literalFalse literal valuation
proof –
  from literalElFormula
have \( \text{var literal} \in \text{vars formula} \)
by (rule formulaContainsItsLiteralsVariable)
with totalFormula
have \( \text{var literal} \in \text{vars valuation} \)
by auto
thus \(?\)thesis using variableDefinedImpliesLiteralDefined [of literal valuation]
by simp
qed

lemma totalValuationForFormulaDefinesAllItsClauses:
fixes formula :: Formula and valuation :: Valuation and clause :: Clause
assumes totalFormula: total valuation (vars formula) and
clauseElFormula: clause el formula
shows clauseTrue clause valuation \(\lor\) clauseFalse clause valuation
proof
  from clauseElFormula totalFormula
  have total valuation (vars clause)
  by (rule totalFormulaImpliesTotalClause)
  thus \(?\)thesis
  by (rule totalValuationForClauseDefinesItsValue)
qed

lemma totalValuationForFormulaDefinesItsValue:
assumes totalFormula: total valuation (vars formula)
shows formulaTrue formula valuation \(\lor\) formulaFalse formula valuation
proof (cases formulaTrue formula valuation)
case True
thus \(?\)thesis
by simp
next
case False
then obtain clause :: Clause
where clauseElFormula: clause el formula and notClauseTrue: \(\neg\)
clauseTrue clause valuation
by (auto simp add: formulaTrueIffAllClausesAreTrue)
from clauseElFormula totalFormula
have total valuation (vars clause)
using totalFormulaImpliesTotalClause [of clause formula valuation]
by simp
with notClauseTrue
have clauseFalse clause valuation
using totalValuationForClauseDefinesItsValue [of valuation clause]
by simp
with clauseElFormula
show \(?\)thesis
by (auto simp add: formulaFalseIffContainsFalseClause)
qed
lemma totalRemoveAllSingleLiteralClause:
  fixes literal :: Literal and valuation :: Valuation and formula :: Formula
  assumes varLiteral: var literal ∈ vars valuation and totalRemoveAll: total valuation (vars (removeAll [literal] formula))
  shows total valuation (vars formula)
proof —
  have vars formula − vars [literal] ⊆ vars (removeAll [literal] formula)
    by (rule varsRemoveAllClauseSuperset)
  with assms
  show ?thesis
    by auto
qed

2.2.8 Models and satisfiability

Model of a formula is a consistent valuation under which formula/clause is true
consts model :: Valuation ⇒ 'a ⇒ bool
defs (overloaded)
modelFormula-def [simp]: model valuation (formula::Formula)== consistent valuation ∧ (formulaTrue formula valuation)
modelClause-def [simp]: model valuation (clause::Clause)== consistent valuation ∧ (clauseTrue clause valuation)

Checks if a formula has a model
definition satisfiable :: Formula ⇒ bool
where
  satisfiable formula == ∃ valuation. model valuation formula

lemma formulaWithEmptyClauseIsUnsatisfiable:
  fixes formula0 :: Formula
  assumes ([]:Clause) el formula
  shows ¬ satisfiable formula
using assms
by (auto simp add: satisfiable-def formulaTrueIffAllClausesAreTrue)

lemma satisfiableSubset:
  fixes formula0 :: Formula and formula :: Formula
  assumes subset: ∀ (clause::Clause). clause el formula0 →→ clause el formula
  shows satisfiable formula →→ satisfiable formula0
proof
  assume satisfiable formula
  show satisfiable formula0
  proof —
    from (satisfiable formula) obtain valuation :: Valuation
where model valuation formula
by (auto simp add: satisfiable-def)
{
  fix clause :: Clause
  assume clause el formula0
  with subset
  have clause el formula
    by simp
  with (model valuation formula)
  have clauseTrue clause valuation
    by (simp add: formulaTrueIffAllClausesAreTrue)
} hence formulaTrue formula0 valuation
by (simp add: formulaTrueIffAllClausesAreTrue)
with (model valuation formula)
have model valuation formula0
  by simp
thus ?thesis
by (auto simp add: satisfiable-def)
qed

lemma satisfiableAppend:
  fixes formula1 :: Formula and formula2 :: Formula
  assumes satisfiable (formula1 @ formula2)
  shows satisfiable formula1 satisfiable formula2
using assms
unfolding satisfiable-def
by (auto simp add: formulaTrueAppend)

lemma modelExpand:
  fixes formula :: Formula and literal :: Literal and valuation :: Valuation
  assumes model valuation formula and var literal /∈ vars valuation
  shows model (valuation @ [literal]) formula
proof –
  from (model valuation formula)
  have formulaTrue formula (valuation @ [literal])
    by (simp add: formulaTrueAppendValuation)
  moreover
  from (model valuation formula)
  have consistent valuation
    by simp
  with (var literal /∈ vars valuation)
  have consistent (valuation @ [literal])
  proof (cases inconsistent (valuation @ [literal]))
    case True
    hence inconsistent valuation ∨ inconsistent [literal] ∨ (∃ l. literalTrue l valuation ∧ literalFalse l [literal])
      by (rule inconsistentAppend)
  qed
  qed
with \(\langle\text{consistent valuation}\rangle\)

have \(\exists l. \text{literalTrue } l \text{ valuation} \land \text{literalFalse } l \text{ [literal]}\)
  by auto

hence \(\text{literalFalse} \text{ literal valuation}\)
  by auto

hence \(\text{var} (\text{opposite literal}) \in (\text{vars valuation})\)
  using \(\text{valuationContainsItsLiteralsVariable}\) [of opposite literal valuation]
  by simp

with \(\langle\text{var literal} \notin \text{vars valuation}\rangle\)

have False
  by simp

thus \(\text{？thesis}..\)

qed simp

ultimately

show \(\text{？thesis}\)
  by auto

qed

2.2.9 Tautological clauses

lemma \(\text{tautologyNotFalse}\):
  fixes \(\text{clause} :: \text{Clause}\) and \(\text{valuation} :: \text{Valuation}\)
  assumes \(\text{clauseTautology clause consistent valuation}\)
  shows \(\neg \text{clauseFalse clause valuation}\)
  using \(\text{assms}\)
    \(\text{clauseTautologyCharacterization}[\text{of clause}]\)
    \(\text{clauseFalseIffAllLiteralsAreFalse}[\text{of clause valuation}]\)
    \(\text{inconsistentCharacterization}\)
  by auto

lemma \(\text{tautologyInTotalValuation}\):
  assumes
    \(\text{clauseTautology clause}\)
    \(\text{vars clause} \subseteq \text{vars valuation}\)
  shows
    \(\text{clauseTrue clause valuation}\)
  proof–
    from \(\langle\text{clauseTautology clause}\rangle\)
    obtain literal
      where \(\text{literal el clause opposite literal el clause}\)
      by \(\langle\text{auto simp add: clauseTautologyCharacterization}\rangle\)
    hence \(\text{var literal} \in \text{vars clause}\)
      using \(\text{clauseContainsItsLiteralsVariable}[\text{of literal clause}]\)
      \(\text{using clauseContainsItsLiteralsVariable}[\text{of opposite literal clause}]\)
    hence \(\text{var literal} \in \text{vars valuation}\)
      \(\langle\text{vars clause} \subseteq \text{vars valuation}\rangle\)
by auto
hence literalTrue literal valuation \lor literalFalse literal valuation
  using varInClauseVars[of var literal valuation]
  using varInClauseVars[of var (opposite literal) valuation]
  using literalsWithSameVariableAreEqualOrOpposite
by auto
thus \?thesis
  using \langle literal el clause \rangle \langle opposite literal el clause \rangle
by (auto simp add: clauseTrueIffContainsTrueLiteral)
qed

lemma modelAppendTautology:
assumes
  model valuation F clauseTautology c
  vars valuation \supseteq vars F \cup vars c
shows
  model valuation \langle F @ [c] \rangle
using assms
using tautologyInTotalValuation[of c valuation]
by (auto simp add: formulaTrueAppend)

lemma satisfiableAppendTautology:
assumes
  satisfiable F clauseTautology c
shows
  satisfiable \langle F @ [c] \rangle
proof
  from \langle clauseTautology c \rangle
  obtain l
  where l el c opposite l el c
by (auto simp add: clauseTautologyCharacterization)
from \langle satisfiable F \rangle
  obtain valuation
  where consistent valuation formulaTrue F valuation
  unfolding satisfiable-def
by auto
show \?thesis
proof (cases var l \in vars valuation)
case True
hence literalTrue l valuation \lor literalFalse l valuation
  using varInClauseVars[of var l valuation]
by (auto simp add: literalsWithSameVariableAreEqualOrOpposite)
hence clauseTrue c valuation
  using \langle l el c \rangle \langle opposite l el c \rangle
by (auto simp add: clauseTrueIffContainsTrueLiteral)
thus \?thesis
  using \langle consistent valuation \rangle \langle formulaTrue F valuation \rangle
  unfolding satisfiable-def
by (auto simp add: formulaTrueIffAllClausesAreTrue)

56
next  
case False  
let ?valuation' = valuation @ [l]  
have model ?valuation' F  
  using (\var l \notin \vars valuation)  
  using \langle formulaTrue F valuation \rangle (consistent valuation)  
  using modelExpand[of valuation F l]  
  by simp  
moreover  
have formulaTrue [c] ?valuation'  
  using (l el c)  
  using clauseTrueIffContainsTrueLiteral[of c ?valuation']  
  using formulaTrueIffAllClausesAreTrue[of [c] ?valuation']  
  by auto  
ultimately  
show ?thesis  
  unfolding satisfiable-def  
  by (auto simp add: formulaTrueAppend)  
qed  
qed

lemma modelAppendTautologicalFormula:  
fixes F :: Formula and F' :: Formula  
assumes  
  model valuation F \forall c. c \in F' \rightarrow clauseTautology c  
  \vars valuation \supseteq \vars F \cup \vars F'  
shows  
  model valuation (F @ F')  
using assms  
proof (induct F')  
case Nil  
  thus ?case  
  by simp  
next  
case (Cons c F'')  
hence model valuation (F @ F'')  
  by simp  
hence model valuation ((F @ F'') @ [c])  
  using Cons(3)  
  using Cons(4)  
  using modelAppendTautology[of valuation F @ F'' c]  
  using varsAppendFormulae[of F F'']  
  by simp  
thus ?case  
  by (simp add: formulaTrueAppend)  
qed
lemma satisfiable_APPEND_Tautological_Formula:
assumes
  satisfiable F \forall c. c \in F' \rightarrow \text{clauseTautology } c
shows
  satisfiable (F @ F')
using assms
proof (induct F')
case Nil
  thus ?case
    by simp
next
case (Cons c F')
hence satisfiable (F @ F')
  by simp
  thus ?case
    using Cons(3)
    unfolding satisfiable_APPEND_Tautology[of F @ F' c]
    unfolding satisfiable-def
    by (simp add: formulaTrueIffAllClausesAreTrue)
qed

lemma satisfiable_FILTER_Tautologies:
shows satisfiable F = satisfiable (filter (\% c. \neg \text{clauseTautology } c) F)
proof (induct F)
case Nil
  thus ?case
    by simp
next
case (Cons c F')
let ?filt = \lambda F. filter (\% c. \neg \text{clauseTautology } c) F
let ?filt' = \lambda F. filter (\% c. \text{clauseTautology } c) F
show ?case
proof
  assume satisfiable (c' \# F')
  thus satisfiable (?filt (c' \# F'))
    unfolding satisfiable-def
    by (auto simp add: formulaTrueIffAllClausesAreTrue)
next
assume satisfiable (?filt (c' \# F'))
thus satisfiable (c' \# F')
proof (cases clauseTautology c')
case True
  hence ?filt (c' \# F') = ?filt F'
    by auto
  hence satisfiable (?filt F')
    using satisfiable (?filt (c' \# F'))
    by simp
  hence satisfiable F'
    using Cons
by simp
thus \$\text{thesis}\$
  using \text{satisfiableAppendTautology}[of F' c']
  using \text{clauseTautology c'}
  unfolding \text{satisfiable-def}
by (auto simp add: \text{formulaTrueIffAllClausesAreTrue})

next
  case False
  hence ?\$\text{filt}\$ (c' # F') = c' # ?\$\text{filt}\$ F'
    by auto
  hence \text{satisfiable} (c' # ?\$\text{filt}\$ F')
    using \text{satisfiable} (?\$\text{filt}\$ (c' # F'))
    by simp
  moreover
  have \forall c. c \in \text{?filt} F' \longrightarrow \text{clauseTautology c}
    by simp
  ultimately
  have \text{satisfiable} ((c' # ?\$\text{filt}\$ F') @ ?\$\text{filt}\$ F')
    using \text{satisfiableAppendTautologicalFormula}[of c' # ?\$\text{filt}\$ F' ?\$\text{filt}\$ F']
    by (simp (no-asm-use))
  thus \$\text{thesis}\$
  unfolding \text{satisfiable-def}
  by (auto simp add: \text{formulaTrueIffAllClausesAreTrue})
qed

lemma \text{modelFilterTautologies}: 
assumes
  model valuation (filter (% c. \neg \text{clauseTautology c}) F)
  vars F \subseteq vars valuation 
shows
  model valuation F 
using \text{assms}
proof (induct F)
  case Nil
  thus \$\text{case}\$
    by simp
next
  case (Cons c' F')
  let ?\$\text{filt}\$ = \lambda F. filter (% c. \neg \text{clauseTautology c}) F
  let ?\$\text{filt}'\$ = \lambda F. filter (% c. \text{clauseTautology c}) F
  show \$\text{case}\$
proof (cases \text{clauseTautology c'})
  case True
  thus \$\text{thesis}\$
    using Cons
    using \text{tautologyInTotalValuation}[of c' valuation]
    by auto
Qed

59
next
  case False
  hence \( \# \text{filt} \ (c' \ # \ # \text{filt} F') = c' \ # \ # \text{filt} F' \)
    by auto
  hence model valuation \( (c' \ # \ # \text{filt} F') \)
    using \( \langle \text{model valuation} \ (\# \text{filt} (c' \ # \ # \text{filt} F')) \rangle \)
    by simp
moreover
have \( \forall \ c. \ c \ el \ ?\text{filt}' F' \rightarrow \text{clauseTautology} \ c \)
  by simp
moreover
have \( \forall \ c \ . \ c \ el \ ?\text{filt}' F' \)
  using \( \langle \text{model valuation} \ (\# \text{filt} (c' \ # \ # \text{filt} F')) \rangle \)
  by simp
ultimately
have model valuation \( \langle (c' \ # \ # \text{filt} F') \rangle \)
  using \( \langle \text{model valuation} \ \# \text{filt} F' \rangle \)
  by simp (no-asm-use) (blast)
thus \( ?\text{thesis} \)
  using \( \langle \text{formulaTrueAppend} \ (\# \text{filt} F' \ # \text{filt}' F' \ \text{valuation}) \rangle \)
  using \( \langle \text{formulaTrueIfAllClausesAreTrue} \ (\# \text{filt} F' \ \text{valuation}) \rangle \)
  using \( \langle \text{formulaTrueIfAllClausesAreTrue} \ (\# \text{filt} F' \ \text{valuation}) \rangle \)
  by auto
qed

2.2.10 Entailment

Formula entails literal if it is true in all its models

**definition** formulaEntailsLiteral :: Formula \( \Rightarrow \) Literal \( \Rightarrow \) bool
where
formulaEntailsLiteral formula literal ==
  \( \forall \ (\text{valuation}::\text{Valuation}). \ \text{model valuation} \ formula \rightarrow \text{literalTrue literal valuation} \)

Clause implies literal if it is true in all its models

**definition** clauseEntailsLiteral :: Clause \( \Rightarrow \) Literal \( \Rightarrow \) bool
where
clauseEntailsLiteral clause literal ==
  \( \forall \ (\text{valuation}::\text{Valuation}). \ \text{model valuation} \ clause \rightarrow \text{literalTrue literal valuation} \)
Formulas entail clauses if they are true in all their models

definition formulaEntailsClause :: \( \text{Formula} \rightarrow \text{Clause} \rightarrow \text{bool} \)
where
formulaEntailsClause formula clause ==
\( \forall (\text{valuation} :: \text{Valuation}). \text{model valuation} \text{ formula} \rightarrow \text{model valuation clause} \)

Formulas entail valuations if they entail their every literal

definition formulaEntailsValuation :: \( \text{Formula} \rightarrow \text{Valuation} \rightarrow \text{bool} \)
where
formulaEntailsValuation formula valuation ==
\( \forall \text{ literal}. \text{ literal el valuation} \rightarrow \text{formulaEntailsLiteral formula literal} \)

Formulas entail formulas if they are true in all their models

definition formulaEntailsFormula :: \( \text{Formula} \rightarrow \text{Formula} \rightarrow \text{bool} \)
where
formulaEntailsFormula-def: formulaEntailsFormula formula formula' ==
\( \forall (\text{valuation} :: \text{Valuation}). \text{model valuation} \text{ formula} \rightarrow \text{model valuation formula'} \)

lemma singleLiteralClausesEntailItsLiteral:
fixes clause :: \( \text{Clause} \) and literal :: \( \text{Literal} \)
assumes length clause = 1 and literal el clause
shows clauseEntailsLiteral clause literal

proof -
from assms
have onlyLiteral: \( \forall l. l \text{ el clause} \rightarrow l = \text{literal} \)
using lengthOneImpliesOnlyElement[of clause literal]
by simp
{
  fix valuation :: \( \text{Valuation} \)
  assume clauseTrue clause valuation
  with onlyLiteral
  have literalTrue literal valuation
  by (auto simp add:clauseTrueIffContainsTrueLiteral)
}
thus ?thesis
  by (simp add:clauseEntailsLiteral-def)
qed

lemma clauseEntailsLiteralThenFormulaEntailsLiteral:
fixes clause :: \( \text{Clause} \) and formula :: \( \text{Formula} \) and literal :: \( \text{Literal} \)
assumes clause el formula and clauseEntailsLiteral clause literal
shows formulaEntailsLiteral formula literal

proof -
{
  fix valuation :: \( \text{Valuation} \)

assume modelFormula: model valuation formula

with ⟨clause el formula⟩
have clauseTrue clause valuation
  by (simp add: formulaTrueIffAllClausesAreTrue)
with modelFormula ⟨clauseEntailsLiteral clause literal⟩
have literalTrue literal valuation
  by (auto simp add: clauseEntailsLiteral-def)
}

thus ?thesis
  by (simp add: formulaEntailsLiteral-def)
qed

lemma formulaEntailsLiteralAppend:
  fixes formula :: Formula and formula' :: Formula and literal :: Literal
  assumes formulaEntailsLiteral formula literal
  shows formulaEntailsLiteral (formula @ formula') literal
proof –
  { fix valuation :: Valuation
    assume modelF': model valuation (formula @ formula')

    hence formulaTrue formula valuation
      by (simp add: formulaTrueAppend)
    with modelF' ⟨formulaEntailsLiteral formula literal⟩
    have literalTrue literal valuation
      by (simp add: formulaEntailsLiteral-def)
  }
  thus ?thesis
    by (simp add: formulaEntailsLiteral-def)
qed

lemma formulaEntailsLiteralSubset:
  fixes formula :: Formula and formula' :: Formula and literal :: Literal
  assumes formulaEntailsLiteral formula literal and ∨ (c::Clause).
    c el formula → c el formula'
  shows formulaEntailsLiteral formula' literal
proof –
  { fix valuation :: Valuation
    assume modelF': model valuation formula'
    with ∨ (c::Clause). c el formula → c el formula'
    have formulaTrue formula valuation
      by (auto simp add: formulaTrueIffAllClausesAreTrue)
    with modelF' ⟨formulaEntailsLiteral formula literal⟩
    have literalTrue literal valuation
      by (simp add: formulaEntailsLiteral-def)
thus \texttt{thesis}
  by \texttt{(simp add:formulaEntailsLiteral-def)}
qed

\textbf{lemma} formulaEntailsLiteralRemoveAll:
\textbf{fixes} formula :: Formula and clause :: Clause and literal :: Literal
\textbf{assumes} formulaEntailsLiteral (removeAll clause formula) literal
\textbf{shows} formulaEntailsLiteral formula literal
\textbf{proof} –
\{  
  fix valuation :: Valuation
  assume modelF: model valuation formula
  hence formulaTrue (removeAll clause formula) valuation
    by \texttt{(auto simp add:formulaTrueRemoveAll)}
  with modelF \langle formulaEntailsLiteral (removeAll clause formula) \rangle
  have literalTrue literal valuation
    by \texttt{(auto simp add:formulaEntailsLiteral-def)}
\}
thus \texttt{thesis}
  by \texttt{(simp add:formulaEntailsLiteral-def)}
qed

\textbf{lemma} formulaEntailsLiteralRemoveAllAppend:
\textbf{fixes} formula1 :: Formula and formula2 :: Formula and clause :: Clause and valuation :: Valuation
\textbf{assumes} formulaEntailsLiteral ((removeAll clause formula1) @ formula2) literal
\textbf{shows} formulaEntailsLiteral (formula1 @ formula2) literal
\textbf{proof} –
\{  
  fix valuation :: Valuation
  assume modelF: model valuation (formula1 @ formula2)
  hence formulaTrue ((removeAll clause formula1) @ formula2) valuation
    by \texttt{(auto simp add:formulaTrueRemoveAll formulaTrueAppend)}
  with modelF \langle formulaEntailsLiteral ((removeAll clause formula1) @ formula2) \rangle
  have literalTrue literal valuation
    by \texttt{(auto simp add:formulaEntailsLiteral-def)}
\}
thus \texttt{thesis}
  by \texttt{(simp add:formulaEntailsLiteral-def)}
qed

\textbf{lemma} formulaEntailsItsClauses:
\textbf{fixes} clause :: Clause and formula :: Formula
assumes clause el formula
shows formulaEntailsClause formula clause
using assms
by (simp add; formulaEntailsClause-def formulaTrueIffAllClausesAreTrue)

lemma formulaEntailsClauseAppend:
  fixes clause :: Clause and formula :: Formula and formula' :: Formula
  assumes formulaEntailsClause formula clause
  shows formulaEntailsClause (formula @ formula') clause
proof -
  { fix valuation :: Valuation
    assume model valuation (formula @ formula')
    hence model valuation formula
    by (simp add; formulaTrueAppend)
    with (formulaEntailsClause formula clause) have clauseTrue clause valuation
    by (simp add; formulaEntailsClause-def)
  }
  thus ?thesis
  by (simp add; formulaEntailsClause-def)
qed

lemma formulaUnsatIffImpliesEmptyClause:
  fixes formula :: Formula
  shows formulaEntailsClause formula [] = (¬ satisfiable formula)
  by (auto simp add; formulaEntailsClause-def satisfiable-def)

lemma formulaTrueExtendWithEntailedClauses:
  fixes formula :: Formula and formula0 :: Formula and valuation :: Valuation
  assumes formulaEntailed: ∀ (clause::Clause). clause el formula → formulaEntailsClause formula0 clause and consistent valuation
  shows formulaTrue formula0 valuation → formulaTrue formula valuation
proof
  assume formulaTrue formula0 valuation
  { fix clause :: Clause
    assume clause el formula
    with formulaEntailed have formulaEntailsClause formula0 clause
    by simp
    with (formulaTrue formula0 valuation) (consistent valuation) have clauseTrue clause valuation
    by (simp add; formulaEntailsClause-def)
  }
  thus formulaTrue formula valuation
by (simp add: formulaTrueIffAllClausesAreTrue)

qed

lemma formulaEntailsFormulaIffEntailsAllItsClauses:
  fixes formula :: Formula and formula' :: Formula
  shows formulaEntailsFormula formula formula' = (∀ clause::Clause. clause el formula' −→ formulaEntailsClause formula clause) (is ?lhs = ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof
    fix clause :: Clause
    show clause el formula' −→ formulaEntailsClause formula clause
    proof
      assume clause el formula'
      show formulaEntailsClause formula clause
      proof
      { fix valuation :: Valuation
        assume model valuation formula
        with ⟨ ?lhs ⟩
        have model valuation formula'
          by (simp add: formulaEntailsFormula-def)
        with ⟨ clause el formula' ⟩
        have clauseTrue clause valuation
          by (simp add: formulaTrueIffAllClausesAreTrue)
      }
      thus ?thesis
      by (simp add: formulaEntailsClause-def)
    qed
    qed
    qed
  next
  assume ?rhs
  thus ?lhs
  proof
  { fix valuation :: Valuation
    assume model valuation formula
    { fix clause :: Clause
      assume clause el formula'
      with ⟨ ?rhs ⟩
      have formulaEntailsClause formula clause
      by auto
      with ⟨ model valuation formula ⟩
      have clauseTrue clause valuation
    }
  }

  qed

next

by (simp add: formulaEntailsClause-def)

} hence (formulaTrue formula’ valuation)
  by (simp add: formulaTrueIffAllClausesAreTrue)

} thus ?thesis
  by (simp add: formulaEntailsFormula-def)

qed

lemma formulaEntailsFormulaThatEntailsClause:
  fixes formula1 :: Formula and formula2 :: Formula and clause :: Clause
  assumes formulaEntailsFormula formula1 formula2 and formulaEntailsClause formula2 clause
  shows formulaEntailsClause formula1 clause
  using assms
  by (simp add: formulaEntailsClause-def formulaEntailsFormula-def)

lemma
  fixes formula1 :: Formula and formula2 :: Formula and formula1’ :: Formula and literal :: Literal
  assumes formulaEntailsLiteral (formula1 @ formula2) literal and formulaEntailsFormula formula1’ formula1
  shows formulaEntailsLiteral (formula1’ @ formula2) literal
  proof -
    fix valuation :: Valuation
    assume model valuation (formula1’ @ formula2)
    hence consistent valuation and formulaTrue formula1’ valuation
      formulaTrue formula2 valuation
      by (auto simp add: formulaTrueAppend)
    with (formulaEntailsFormula formula1’ formula1)
    have model valuation formula1
      by (simp add: formulaEntailsFormula-def)
    with (formulaTrue formula2 valuation)
    have model valuation (formula1 @ formula2)
      by (simp add: formulaTrueAppend)
    with (formulaEntailsLiteral (formula1 @ formula2) literal)
    have literalTrue literal valuation
      by (simp add: formulaEntailsLiteral-def)
    } thus ?thesis
      by (simp add: formulaEntailsLiteral-def)
    qed

lemma formulaFalseInEntailedValuationIsUnsatisfiable:
fixes formula :: Formula and valuation :: Valuation
assumes formulaFalse formula valuation and formulaEntailsValuation formula valuation
shows ¬ satisfiable formula

proof —
  from ⟨formulaFalse formula valuation⟩ obtain clause :: Clause
    where clause el formula and clauseFalse clause valuation
    by (auto simp add: formulaFalseIffContainsFalseClause)
  {  
    fix valuation' :: Valuation
    assume modelV': model valuation' formula
    with ⟨clause el formula⟩ obtain literal :: Literal
      where literal el clause and literalTrue literal valuation'
      by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIf-
fContainsTrueLiteral)
    with ⟨clauseFalse clause valuation⟩
    have literalFalse literal valuation'
      by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
    with ⟨formulaEntailsValuation formula valuation⟩
    have formulaEntailsLiteral formula (opposite literal)
      unfolding formulaEntailsValuation-def
      by simp
    with modelV'
    have literalFalse literal valuation'
      by (auto simp add: formulaEntailsLiteral-def)
    from ⟨literalTrue literal valuation'⟩ ⟨literalFalse literal valuation'⟩
    modelV'
    have False
      by (simp add: inconsistentCharacterization)
  }
  thus ?thesis
  by (auto simp add: satisfiable-def)
qed

lemma formulaFalseInEntailedOrPureValuationIsUnsatisfiable:
  fixes formula :: Formula and valuation :: Valuation
  assumes formulaFalse formula valuation and
  ∀ literal'. literal' el valuation → formulaEntailsLiteral formula literal'
  shows ¬ satisfiable formula

proof —
  from ⟨formulaFalse formula valuation⟩ obtain clause :: Clause
    where clause el formula and clauseFalse clause valuation
    by (auto simp add: formulaFalseIffContainsFalseClause)
  {  
    fix valuation' :: Valuation
    assume modelV': model valuation' formula
    with ⟨clause el formula⟩ obtain literal :: Literal
      where literal el clause and literalTrue literal valuation'
  }
by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIfContainsTrueLiteral)

with ⟨clauseFalse clause valuation⟩

have literalFalse literal valuation
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)

with ⟨∀ literal′. literal′ el valuation → formulaEntailsLiteral formula literal′ ∨ ¬ opposite literal′ el formula⟩

have formulaEntailsLiteral formula (opposite literal) ∨ ¬ literal el formula
  by auto

moreover

{ 
  assume formulaEntailsLiteral formula (opposite literal)
  with modelV'
  have literalFalse literal valuation'
    by (auto simp add: formulaEntailsLiteral-def)

  from ⟨literatrue literal valuation'⟩ ⟨literalFalse literal valuation'⟩

  modelV'
  have False
    by (simp add: inconsistentCharacterization)
}

moreover

{ 
  assume ¬ literal el formula
  with ⟨clause el formula⟩ ⟨literal el clause⟩
  have False
    by (simp add: literalElFormulaCharacterization)
}

ultimately

have False
  by auto
}

thus ?thesis
  by (auto simp add: satisfiable-def)
qed

lemma unsatisfiableFormulaWithSingleLiteralClause:
  fixes formula :: Formula and literal :: Literal
  assumes ¬ satisfies formula and [literal] el formula
  shows formulaEntailsLiteral (removeAll [literal] formula) (opposite literal)
proof –

  { 
    fix valuation :: Valuation
    assume model valuation (removeAll [literal] formula)
    hence literalFalse literal valuation
    proof (cases var literal ∈ vars valuation)
      case True
{ assume literalTrue literal valuation
  with model valuation (removeAll [literal] formula)
  have model valuation formula
    by (auto simp add:formulaTrueIffAllClausesAreTrue)
  with ¬ satisfiable formula
  have False
    by (auto simp add:satisfiable-def)
}
with True
show thesis
  using variableDefinedImpliesLiteralDefined [of literal valuation]
  by auto
next
case False
  with model valuation (removeAll [literal] formula)
  have model (valuation @ [literal]) (removeAll [literal] formula)
    by (rule modelExpand)
  hence formulaTrue (removeAll [literal] formula) (valuation @ [literal])
  and consistent (valuation @ [literal])
    by auto
    from (formulaTrue (removeAll [literal] formula) (valuation @ [literal]))
      have formulaTrue formula (valuation @ [literal])
        by (rule trueFormulaWithSingleLiteralClause)
      with consistent (valuation @ [literal])
        have model (valuation @ [literal]) formula
          by simp
        with ¬ satisfiable formula
        have False
          by (auto simp add:satisfiable-def)
        thus thesis ..
      qed
    }
  thus thesis
    by (simp add:formulaEntailsLiteral-def)
qed

lemma unsatisfiableFormulaWithSingleLiteralClauses:
  fixes F::Formula and c::Clause
  assumes ¬ satisfiable (F @ val2form (oppositeLiteralList c)) ¬ clauseTautology c
  shows formulaEntailsClause F c
proof-
  { fix v::Valuation
    assume model v F
    with ¬ satisfiable (F @ val2form (oppositeLiteralList c))
    }
have \( \neg \) formulaTrue (val2form (oppositeLiteralList c)) \( v \)
unfolding satisfiable-def
by (auto simp add: formulaTrueAppend)
have clauseTrue c \( v \)
proof (cases \( \exists \) \( l \). \( l \in c \wedge \) (literalTrue \( l \) \( v \)))
case True
  thus \( \theta \)thesis
    using clauseTrueIffContainsTrueLiteral
    by simp
next
case False
let \( ?v' = v \odot (\text{oppositeLiteralList} \ c) \)

have \( \neg \) inconsistent (oppositeLiteralList \( c \))
proof-
  { 
    assume \( \neg \) \( \theta \)thesis
    then obtain \( l :: \)Literal
      where \( l \in (\text{oppositeLiteralList} \ c) \) opposite \( l \in \) (oppositeLiteralList \( c \))
      using inconsistentCharacterization [of oppositeLiteralList \( c \)]
      by auto
    hence (opposite \( l \)) \( v \) \( c \) \( l \in \) \( c \)
      using literalElListIffOppositeLiteralElOppositeLiteralList [of \( l \) \( c \)]
      using literalElListIffOppositeLiteralElOppositeLiteralList [of \( \text{opposite} \ l \) \( c \) ]
      by auto
    hence clauseTautology \( c \)
      using clauseTautologyCharacterization [of \( c \) ]
      by auto
    with \( \neg \) clauseTautology \( c \)
    have False
      by simp
  }
  thus \( \theta \)thesis
    by auto
qed
with False \( \langle \text{model} \ v \ F \rangle \)
have consistent \( ?v' \)
  using inconsistentAppend [of \( v \) oppositeLiteralList \( c \) ]
unfolding consistent-def
using literalElListIffOppositeLiteralElOppositeLiteralList
by auto
moreover
from \( \langle \text{model} \ v \ F \rangle \)
have formulaTrue \( F \) \( ?v' \)
  using formulaTrueAppendValuation
  by simp

70
moreover
have \( \text{formulaTrue} \ (\text{val2form} \ (\text{oppositeLiteralList} \ c)) \) ?v'
using \( \text{val2formFormulaTrue} \ (\text{of oppositeLiteralList} \ c \ v @ \text{oppositeLiteralList} \ c) \)
by simp
ultimately
have model ?v' \( (F @ \text{val2form} \ (\text{oppositeLiteralList} \ c)) \)
by \( \text{(simp add: formulaTrueAppend)} \)
with \( \neg \text{satisfiable} \ (F @ \text{val2form} \ (\text{oppositeLiteralList} \ c)) \)
have False
unfolding \text{satisfiable-def}
by auto
thus \?thesis
by simp
qed

thus \?thesis
unfolding \text{formulaEntailsClause-def}
by simp
qed

\text{lemma satisfiableEntailedFormula:}
fixes \text{formula0} :: \text{Formula} and \text{formula} :: \text{Formula}
assumes \text{formulaEntailsFormula} \text{formula0} \text{formula}
shows \text{satisfiable formula0} \(\rightarrow\) \text{satisfiable formula}
proof
assume \text{satisfiable formula0}
show \text{satisfiable formula}
proof
from \text{(satisfiable formula0): obtain} \text{valuation} :: \text{Valuation}
where model \text{valuation} \text{formula0}
by \text{(auto simp add: satisfiable-def)}
with \text{(formulaEntailsFormula formula0 formula)}
have model \text{valuation} \text{formula}
by \text{(simp add: formulaEntailsFormula-def)}
thus \?thesis
by \text{(auto simp add: satisfiable-def)}
qed
qed

\text{lemma val2formIsEntailed:}
shows \text{formulaEntailsValuation} \(F' @ \text{val2form valuation} @ F'') \text{ valuation}
proof
{\text{fix l::Literal}
assume l el valuation
hence \([l]\) el \text{val2form valuation}
by \text{(induct valuation) (auto)}}
have \( \text{formulaEntailsLiteral} \left( F' @ \text{val2form valuation @ } F'' \right) \) \( l \)
proof -
{  
  fix \( \text{valuation}'::\text{Valuation} \)
  assume \( \text{formulaTrue} \left( F' @ \text{val2form valuation @ } F'' \right) \) \( \text{valuation}' \)
  hence \( \text{literalTrue} \ l \) \( \text{valuation}' \)
  using \( \{ l \} \ \text{el \ val2form valuation} \)
  using \( \text{formulaTrueIffAllClausesAreTrue} \)\( \text{of } F' @ \text{val2form valuation @ } F'' \) \( \text{valuation}' \)\( \text{valuation} \)
  by (auto simp add: \( \text{clauseTrueIffContainsTrueLiteral} \))
}  
thus \( \text{thesis} \)
unfolding \( \text{formulaEntailsLiteral-def} \)
by simp
qed

\[ \textbf{2.2.11 \ Equivalency} \]

Formulas are equivalent if they have same models.

\textbf{definition} equivalentFormulae :: Formula \( \Rightarrow \) Formula \( \Rightarrow \) bool
where
\( \text{equivalentFormulae} \) \( \text{formula1} \) \( \text{formula2} \) \( \Rightarrow \)
\( \forall \) (valuation::Valuation). \( \text{model valuation formula1} = \text{model valuation formula2} \)

\textbf{lemma} equivalentFormulaeIffEntailEachOther:
  \( \text{fixes} \) \( \text{formula1} :: \text{Formula \ and \ formula2 :: Formula} \)
  \( \text{shows} \) equivalentFormulae \( \text{formula1} \) \( \text{formula2} \)\( = \) (formulaEntailsFormula \( \text{formula1} \) \( \text{formula2} \) \( \land \) formulaEntailsFormula \( \text{formula2} \) \( \text{formula1} \))
  by (auto simp add: formulaEntailsFormula-def equivalentFormulae-def)

\textbf{lemma} equivalentFormulaeReflexivity:
  \( \text{fixes} \) \( \text{formula} :: \text{Formula} \)
  \( \text{shows} \) equivalentFormulae \( \text{formula} \) \( \text{formula} \)
  unfolding equivalentFormulae-def
  by auto

\textbf{lemma} equivalentFormulaeSymmetry:
  \( \text{fixes} \) \( \text{formula1} :: \text{Formula \ and \ formula2 :: Formula} \)
  \( \text{shows} \) equivalentFormulae \( \text{formula1} \) \( \text{formula2} \)\( = \) equivalentFormulae \( \text{formula2} \) \( \text{formula1} \)
  unfolding equivalentFormulae-def
  by auto

\textbf{lemma} equivalentFormulaeTransitivity:
\begin{verbatim}
fixes formula1 :: Formula and formula2 :: Formula and formula3 :: Formula
assumes equivalentFormulae formula1 formula2 and equivalentFormulae formula2 formula3
shows equivalentFormulae formula1 formula3
using assms
unfolding equivalentFormulae-def
by auto

lemma equivalentFormulaeAppend:
fixes formula1 :: Formula and formula1' :: Formula and formula2 :: Formula
assumes equivalentFormulae formula1 formula1'
shows equivalentFormulae (formula1 @ formula2) (formula1' @ formula2)
using assms
unfolding equivalentFormulae-def
by (auto simp add: formulaTrueAppend)

lemma satisfiableEquivalent:
fixes formula1 :: Formula and formula2 :: Formula
assumes equivalentFormulae formula1 formula2
shows satisfiable formula1 = satisfiable formula2
using assms
unfolding equivalentFormulae-def
unfolding satisfiable-def
by auto

lemma satisfiableEquivalentAppend:
fixes formula1 :: Formula and formula1' :: Formula and formula2 :: Formula
assumes equivalentFormulae formula1 formula1' and satisfiable (formula1 @ formula2)
shows satisfiable (formula1' @ formula2)
using assms
proof -
from ⟨satisfiable (formula1 @ formula2)⟩ obtain valuation :: Valuation
where consistent valuation formulaTrue formula1 valuation formulaTrue formula2 valuation
unfolding satisfiable-def
by (auto simp add: formulaTrueAppend)
from ⟨equivalentFormulae formula1 formula1' ⟩ ⟨consistent valuation⟩ ⟨formulaTrue formula1 valuation⟩
⟨formulaTrue formula1' valuation⟩
have formulaTrue formula1' valuation
unfolding equivalentFormulae-def
by auto
show ?thesis
using ⟨consistent valuation⟩ ⟨formulaTrue formula1' valuation⟩ ⟨formulaTrue formula2 valuation⟩
\end{verbatim}
lemma replaceEquivalentByEquivalent:
  fixes formula :: Formula and formula' :: Formula and formula1 :: Formula and formula2 :: Formula
  assumes equivalentFormulae formula formula'
  shows equivalentFormulae (formula1 @ formula @ formula2) (formula1 @ formula' @ formula2)
unfolding equivalentFormulae-def
proof
  fix v :: Valuation
  show model v (formula1 @ formula @ formula2) = model v (formula1 @ formula' @ formula2)
  proof
    assume model v (formula1 @ formula @ formula2)
    hence *: consistent v formulaTrue formula1 v formulaTrue formula v formulaTrue formula2 v
      by (auto simp add: formulaTrueAppend)
    from ⟨consistent v ⟩ ⟨formulaTrue formula v ⟩ ⟨equivalentFormulae formula formula'⟩
      have formulaTrue formula' v
        unfolding equivalentFormulae-def
        by auto
    thus model v (formula1 @ formula' @ formula2)
      using *
      by (simp add: formulaTrueAppend)
  next
    assume model v (formula1 @ formula' @ formula2)
    hence *: consistent v formulaTrue formula1 v formulaTrue formula' v formulaTrue formula2 v
      by (auto simp add: formulaTrueAppend)
    from ⟨consistent v ⟩ ⟨formulaTrue formula' v ⟩ ⟨equivalentFormulae formula formula'⟩
      have formulaTrue formula v
        unfolding equivalentFormulae-def
        by auto
    thus model v (formula1 @ formula @ formula2)
      using *
      by (simp add: formulaTrueAppend)
  qed
qed

lemma clauseOrderIrrelevant:
  shows equivalentFormulae (F1 @ F @ F' @ F2) (F1 @ F' @ F @ F2)
unfolding equivalentFormulae-def
by (auto simp add: formulaTrueIffAllClausesAreTrue)

lemma extendEquivalentFormulaWithEntailedClause:
  fixes formula1 :: Formula and formula2 :: Formula and clause :: Clause
  assumes equivalentFormulae formula1 formula2 and formulaEntailsClause formula2 clause
  shows equivalentFormulae formula1 (formula2 @ [clause])
  unfolding equivalentFormulae-def
proof
  fix valuation :: Valuation
  show model valuation formula1 = model valuation (formula2 @ [clause])
  proof
    assume model valuation formula1
    hence consistent valuation
      by simp
    from ⟨model valuation formula1⟩ ⟨equivalentFormulae formula1 formula2⟩
    have model valuation formula2
      unfolding equivalentFormulae-def
      by simp
    moreover
    from ⟨model valuation formula2⟩ ⟨formulaEntailsClause formula2 clause⟩
    have clauseTrue clause valuation
      unfolding formulaEntailsClause-def
      by simp
    ultimately show
      model valuation (formula2 @ [clause])
      by (simp add: formulaTrueAppend)
  next
  assume model valuation (formula2 @ [clause])
  hence consistent valuation
    by simp
  from ⟨model valuation (formula2 @ [clause])⟩
  have model valuation formula2
    by (simp add: formulaTrueAppend)
  with ⟨equivalentFormulae formula1 formula2⟩
  show model valuation formula1
    unfolding equivalentFormulae-def
    by auto
qed

lemma entailsLiteralReplacePartWithEquivalent:
  assumes equivalentFormulae F F' and formulaEntailsLiteral (F1 @ F @ F2) l
  shows formulaEntailsLiteral (F1 @ F' @ F2) l
proof
{
  fix v::Valuation
  assume model v (F1 @ F' @ F2)
  hence consistent v and formulaTrue F1 v and formulaTrue F' v
  and formulaTrue F2 v
  by (auto simp add:formulaTrueAppend)
  with (equivalentFormulae F F')
  have formulaTrue F v
    unfolding equivalentFormulae-def
    by auto
  with (consistent v) (formulaTrue F1 v) (formulaTrue F2 v)
  have model v (F1 @ F @ F2)
    by (auto simp add:formulaTrueAppend)
  with (formulaEntailsLiteral (F1 @ F @ F2) l)
  have literalTrue l v
    unfolding formulaEntailsLiteral-def
    by auto
}
thus ?thesis
  unfolding formulaEntailsLiteral-def
  by auto
qed

2.2.12 Remove false and duplicate literals of a clause

definition
removeFalseLiterals :: Clause ⇒ Valuation ⇒ Clause
where
removeFalseLiterals clause valuation = filter (λ l. ¬ literalFalse l valuation) clause

lemma clauseTrueRemoveFalseLiterals:
  assumes consistent v
  shows clauseTrue c v = clauseTrue (removeFalseLiterals c v) v
using assms
unfolding removeFalseLiterals-def
by (auto simp add: clauseTrueIffContainsTrueLiteral inconsistentCharacterization)

lemma clauseTrueRemoveDuplicateLiterals:
  shows clauseTrue c v = clauseTrue (remdups c) v
by (induct c) (auto simp add: clauseTrueIffContainsTrueLiteral)

lemma removeDuplicateLiteralsEquivalentClause:
  shows equivalentFormulae [remdups clause] [clause]
unfolding equivalentFormulae-def
by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIffContainsTrueLiteral)
lemma falseLiteralsCanBeRemoved:

fixes F :: Formula and F' :: Formula and v :: Valuation
assumes equivalentFormulae (F1 @ val2form v @ F2) F' 
shows equivalentFormulae (F1 @ val2form v @ [removeFalseLiterals c v] @ F2) (F' @ [c])
(is equivalentFormulae lhs ?rhs)

unfolding equivalentFormulae-def

proof
  fix v' :: Valuation
  show model v' ?lhs = model v' ?rhs
  proof
    assume model v' ?lhs
    hence consistent v' and
    formulaTrue (F1 @ val2form v @ F2) v' and
    clauseTrue (removeFalseLiterals c v) v'
    by (auto simp add: formulaTrueAppend formulaTrueIffAllClausesAreTrue)

    from ⟨consistent v' ⟩ ⟨formulaTrue (F1 @ val2form v @ F2) v' ⟩ ⟨equivalentFormulae (F1 @ val2form v @ F2) F' ⟩
    have model v' F'
      unfolding equivalentFormulae-def
      by auto
    moreover
    from ⟨clauseTrue (removeFalseLiterals c v) v' ⟩
    have clauseTrue c v'
      unfolding removeFalseLiterals-def
      by (auto simp add: clauseTrueIffContainsTrueLiteral)
    ultimately
    show model v' ?rhs
      by (simp add: formulaTrueAppend)
  next
    assume model v' ?rhs
    hence consistent v' and formulaTrue F' v' and clauseTrue c v'
    by (auto simp add: formulaTrueAppend formulaTrueIffAllClausesAreTrue)

    from ⟨consistent v' ⟩ ⟨formulaTrue F' v' ⟩ ⟨equivalentFormulae (F1 @ val2form v @ F2) F' ⟩
    have model v' (F1 @ val2form v @ F2)
      unfolding equivalentFormulae-def
      by auto
    moreover
    have clauseTrue (removeFalseLiterals c v) v'
      proof
        from ⟨clauseTrue c v' ⟩
        obtain l :: Literal
      end

77
where \( l \in c \) and \( \text{literalTrue} \ l \ v' \)
by (auto simp add: clauseTrueIffContainsTrueLiteral)
have \( \neg \text{literalFalse} \ l \ v \)
proof
  
  assume \( \neg \ ?\thesis \)
  hence \( \text{opposite} \ l \in v \)
  by simp
  with \langle \\text{model} \ v' @ (F_1 @ \text{val2form} \ v @ F_2) \rangle
  have \( \text{opposite} \ l \in v' \)
  using \text{val2formFormulaTrue}[of \ v' \]
  by auto (simp add: \text{formulaTrueAppend})
  with \langle \text{literalTrue} \ l \ v' \rangle \langle \text{consistent} \ v' \rangle
  have \( \text{False} \)
  by (simp add: \text{inconsistentCharacterization})
  
  thus \( ?\thesis \)
  by auto
qeda
with \( l \in c \)
have \( l \in \text{(removeFalseLiterals} \ c \ v) \)
  unfolding \text{removeFalseLiterals-def}
  by simp
  with \langle \text{literalTrue} \ l \ v' \rangle
  show \( ?\thesis \)
  by (auto simp add: clauseTrueIffContainsTrueLiteral)
qeda
ultimately
show \( \text{model} \ v' \ ?\lhs \)
  by (simp add: \text{formulaTrueAppend})
qeda

lemma \text{falseAndDuplicateLiteralsCanBeRemoved}: 

assumes \( \text{equivalentFormulae} \ (F_1 @ \text{val2form} \ v @ F_2) \ F' \)
shows \( \text{equivalentFormulae} \ (F_1 @ \text{val2form} \ v @ [\text{remdups (removeFalseLiterals} \ c \ v]) @ F_2) \ (F' @ [c]) \)
  (is \text{equivalentFormulae} \ ?\lhs \ ?rhs)
proof
  
  from \langle \\text{equivalentFormulae} \ (F_1 @ \text{val2form} \ v @ F_2) \ F' \rangle
  have \\text{equivalentFormulae} \ (F_1 @ \text{val2form} \ v @ [\text{removeFalseLiterals} \ c \ v]) @ F_2 \ (F' @ [c])
  using \text{falseLiteralsCanBeRemoved}
  by simp
  have \text{equivalentFormulae} \ [\text{remdups (removeFalseLiterals} \ c \ v)] [\text{removeFalseLiterals} \ c \ v]
  using \text{removeDuplicateLiteralsEquivalentClause}
  by simp
hence equivalentFormulae (F1 @ val2form v @ [remdups (removeFalseLiterals c v)]) @ F2)
  (F1 @ val2form v @ [removeFalseLiterals c v]) @ F2)
using replaceEquivalentByEquivalent
[of [remdups (removeFalseLiterals c v)] [removeFalseLiterals c v]
F1 @ val2form v F2]
by auto
thus ?thesis
using (equivalentFormulae (F1 @ val2form v @ [removeFalseLiterals c v]) @ F2)
using equivalentFormulaeTransitivity[of
(F1 @ val2form v @ [remdups (removeFalseLiterals c v)])
@ F2)
  (F1 @ val2form v @ [removeFalseLiterals c v]) @ F2)
F' @ [c]]
by simp
qed

lemma satisfiedClauseCanBeRemoved:
assumes
equivalentFormulae (F @ val2form v) F'
clauseTrue c v
shows equivalentFormulae (F @ val2form v) (F' @ [c])
unfolding equivalentFormulae-def
proof
fix v' :: Valuation
show model v' (F @ val2form v) = model v' (F' @ [c])
proof
assume model v' (F @ val2form v)
from ⟨model v' (F @ val2form v)⟩(equivalentFormulae (F @ val2form v) F'\)
  have model v' F'
  unfolding equivalentFormulae-def
  by auto
moreover
have clauseTrue c v'
proof
  from :clauseTrue c v
  obtain l :: Literal
    where literalTrue l v and l el c
    by (auto simp add:clauseTrueIffContainsTrueLiteral)
  with ⟨formulaTrue (F @ val2form v) v'⟩\n    have literalTrue l v'
      using val2formFormulaTrue[of v v']
      using formulaTrueAppend[of F val2form v]
      by simp
thus \theta
\text{thesis}
using \{ l \in c \}
by (auto simp add: \text{clauseTrueIffContainsTrueLiteral})
qed
ultimately
show \text{model v'} (F' @ [c])
by (simp add: \text{formulaTrueAppend})
next
assume \text{model v'} (F' @ [c])
thus \text{model v'} (F @ \text{val2form v})
using (\text{equivalentFormulae (F @ \text{val2form v}) F'})
unfolding \text{equivalentFormulae-def}
using \text{formulaTrueAppend[of F' [c] v']}
by auto
qed
qed

lemma \text{formulaEntailsClauseRemoveEntailedLiteralOpPosites:}
assumes \text{formulaEntailsClause F clause}
\text{formulaEntailsValuation F valuation}
shows \text{formulaEntailsClause F (list-diff clause (oppositeLiteralList valuation))}
proof -
{\begin{itemize}
  \item fix valuation'
  \item assume \text{model valuation'} F
  \item hence consistent valuation' formulaTrue F valuation'
    by (auto simp add: \text{formulaTrueAppend})
  have \text{model valuation'} clause
    using (consistent valuation'\}
    using (formulaTrue F valuation'\}
    using (formulaEntailsClause F clause)\}
    unfolding \text{formulaEntailsClause-def}
    by simp
  then obtain l::Literal
    where l el clause literalTrue l valuation'
    by (auto simp add: \text{clauseTrueIffContainsTrueLiteral})
  moreover
  hence \neg l el (oppositeLiteralList valuation)
  proof -
  {\begin{itemize}
    \item assume l el (oppositeLiteralList valuation)
    \item hence (opposite l) el valuation
      using literalElListIffOppositeLiteralElOppositeLiteralList[of l oppositeLiteralList valuation]
  \end{itemize}}
\end{itemize}}

80
by simp

hence \text{formulaEntailsLiteral } F \text{ (opposite } l) using \langle \text{formulaEntailsValuation } F \text{ valuation} \rangle

unfolding \text{formulaEntailsValuation-def} by simp

hence literal\text{False } l \text{ valuation'} using \langle \text{consistent valuation'} \rangle

using \langle \text{formulaTrue } F \text{ valuation}' \rangle

unfolding \text{formulaEntailsLiteral-def} by simp

with \langle \text{literalTrue } l \text{ valuation'} \rangle \text{ (consistent valuation')}

have False by (simp add: inconsistentCharacterization)

\}

thus \?thesis

by auto

qed

ultimately

have model \text{valuation'} \text{(list-diff clause (oppositeLiteralList valuation))}

using \text{consistent valuation'}

using \text{listDiffIff}[of } l \text{ clause oppositeLiteralList valuation]

by (auto simp add: clauseTrueIffContainsTrueLiteral)

\}

thus \?thesis

unfolding \text{formulaEntailsClause-def} by simp

qed

2.2.13 Resolution

definition resolve clause1 clause2 literal == removeAll literal clause1 @ removeAll (opposite literal) clause2

lemma resolventIsEntailed:

fixes clause1 :: Clause and clause2 :: Clause and literal :: Literal

shows formulaEntailsClause [clause1, clause2] (resolve clause1 clause2 literal)

proof -

\{

fix valuation :: Valuation

assume model \text{valuation} [clause1, clause2]

from \langle \text{model valuation [clause1, clause2]} \rangle obtain l1 :: Literal

where l1 \in clause1 and literalTrue l1 \text{ valuation}

by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIffContainsTrueLiteral)

from \langle \text{model valuation [clause1, clause2]} \rangle obtain l2 :: Literal

where l2 \in clause2 and literalTrue l2 \text{ valuation}

by (auto simp add: formulaTrueIffAllClausesAreTrue clauseTrueIff-
fContainsTrueLiteral

have clauseTrue (resolve clause1 clause2 literal) valuation
proof (cases literal = l1)
  case False
  with ⟨l1 el clause1⟩
  have l1 el (resolve clause1 clause2 literal)
    by (auto simp add:resolve-def)
  with ⟨literalTrue l1 valuation⟩
  show ?thesis
    by (auto simp add: clauseTrueIffContainsTrueLiteral)
next
  case True
  from ⟨model valuation [clause1, clause2]⟩
  have consistent valuation
    by simp
  from True ⟨literalTrue l1 valuation⟩ ⟨literalTrue l2 valuation⟩
  ⟨consistent valuation⟩
  have literal ≠ opposite l2
    by (auto simp add: inconsistentCharacterization)
  with ⟨l2 el clause2⟩
  have l2 el (resolve clause1 clause2 literal)
    by (auto simp add: resolve-def)
  with ⟨literalTrue l2 valuation⟩
  show ?thesis
    by (auto simp add: clauseTrueIffContainsTrueLiteral)
qed

thus ?thesis
  by (simp add: formulaEntailsClause-def)
qed

lemma formulaEntailsResolvent:
fixes formula :: Formula and clause1 :: Clause and clause2 :: Clause
assumes formulaEntailsClause formula clause1 and formulaEntailsClause formula clause2
shows formulaEntailsClause formula (resolve clause1 clause2 literal)
proof −
{
  fix valuation :: Valuation
  assume model valuation formula
  hence consistent valuation
    by simp
  from ⟨model valuation formula ⟩ ⟨formulaEntailsClause formula clause1⟩
  have clauseTrue clause1 valuation
    by (simp add: formulaEntailsClause-def)
  from ⟨model valuation formula ⟩ ⟨formulaEntailsClause formula clause2⟩
  have clauseTrue clause2 valuation

  82
by (simp add: formulaEntailsClause-def)

thus ?thesis
by (simp add: formulaEntailsClause-def)

qed

lemma resolveFalseClauses:
  fixes literal :: Literal and clause1 :: Clause and clause2 :: Clause
  and valuation :: Valuation
  assumes clauseFalse (removeAll literal clause1) valuation
  and clauseFalse (removeAll (opposite literal) clause2) valuation
  shows clauseFalse (resolve clause1 clause2 literal) valuation

proof –
{ fix l :: Literal
  assume l el (resolve clause1 clause2 literal)
  have literalFalse l valuation
  proof –
    from : l el (resolve clause1 clause2 literal):
    have l el (removeAll literal clause1) \or l el (removeAll (opposite literal) clause2)
    unfolding resolve-def
    by simp
    thus ?thesis
  proof
    assume l el (removeAll literal clause1)
    thus literalFalse l valuation
    using ⟨clauseFalse (removeAll literal clause1) valuation⟩
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
  next
    assume l el (removeAll (opposite literal) clause2)
    thus literalFalse l valuation
    using ⟨clauseFalse (removeAll (opposite literal) clause2) valuation⟩
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
  qed
  qed
  } thus ?thesis
by (simp add: clauseFalseIffAllLiteralsAreFalse)
Definition isUnitClause :: Clause ⇒ Literal ⇒ Valuation ⇒ bool
where
isUnitClause uClause uLiteral valuation ==
uLiteral el uClause ∧
¬ (literalTrue uLiteral valuation) ∧
¬ (literalFalse uLiteral valuation) ∧
(∀ literal. literal el uClause ∧ literal ≠ uLiteral → literalFalse literal valuation)

Lemma unitLiteralIsEntailed:
fixes uClause :: Clause and uLiteral :: Literal and formula :: Formula and valuation :: Valuation
assumes isUnitClause uClause uLiteral valuation and formulaEntailsClause formula uClause
shows formulaEntailsLiteral (formula @ val2form valuation) uLiteral
proof |
{ fix valuation'
assume model valuation' (formula @ val2form valuation)
hence consistent valuation'
by simp
from ⟨model valuation' (formula @ val2form valuation)⟩
have formulaTrue formula valuation' and formulaTrue (val2form valuation) valuation'
by (auto simp add: formulaTrueAppend)
from ⟨formulaTrue formula valuation'⟩ ⟨consistent valuation'⟩ ⟨formulaEntailsClause formula uClause⟩
have clauseTrue uClause valuation'
by (simp add: formulaEntailsClause-def)
then obtain l :: Literal
where l el uClause literalTrue l valuation'
by (auto simp add: clauseTrueIffContainsTrueLiteral)
hence literalTrue uLiteral valuation'
proof (cases l = uLiteral)
case True
with ⟨literalTrue l valuation'⟩
show ?thesis
by simp
next
case False
with ⟨l el uClause⟩ (isUnitClause uClause uLiteral valuation)
have literalFalse l valuation
by (simp add: isUnitClause-def)
from ⟨formulaTrue (val2form valuation) valuation' ⟩
have ∀ literal :: Literal. literal el valuation ⟷ literal el valuation'
  using val2formFormulaTrue [of valuation valuation']
  by simp
with ⟨literalFalse l valuation⟩
have literalFalse l valuation'
  by auto
with ⟨literalTrue l valuation' ⟩ (consistent valuation')
have False
  by (simp add: inconsistentCharacterization)
thus ?thesis ..
qed

thus ?thesis
  by (simp add: formulaEntailsLiteral-def)
qed

lemma isUnitClauseRemoveAllUnitLiteralIsFalse:
  fixes uClause :: Clause and uLiteral :: Literal and valuation :: Valuation
  assumes isUnitClause uClause uLiteral valuation
  shows clauseFalse (removeAll uLiteral uClause) valuation
proof −
  { fix literal :: Literal
    assume literal el (removeAll uLiteral uClause)
    hence literal el uClause and literal ≠ uLiteral
      by auto
    with ⟨isUnitClause uClause uLiteral valuation⟩
    have literalFalse literal valuation
      by (simp add: isUnitClause-def)
  }
  thus ?thesis
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
qed

lemma isUnitClauseAppendValuation:
  assumes isUnitClause uClause uLiteral valuation l ≠ uLiteral l ≠ opposite uLiteral
  shows isUnitClause uClause uLiteral (valuation @ [l])
using assms
unfolding isUnitClause-def
by auto

lemma containsTrueNotUnit:
  assumes l el c and literalTrue l v and consistent v
  shows
\[ \neg (\exists \, ul. \, \text{isUnitClause} \, c \, ul \, v) \]

**using** assms

**unfolding** isUnitClause-def

**by** (auto simp add: inconsistentCharacterization)

**lemma** unitBecomesFalse:

**assumes**

\( \text{isUnitClause} \, u\text{Clause} \, u\text{Literal} \, \text{valuation} \)

**shows**

\( \text{clauseFalse} \, u\text{Clause} \, (\text{valuation} \odot [\text{opposite} \, u\text{Literal}]) \)

**using** assms

**using** isUnitClauseRemoveAllUnitLiteralIsFalse[of uClause uLiteral valuation]

**by** (auto simp add: clauseFalseIffAllLiteralsAreFalse)

### 2.2.15 Reason clauses

A clause is *reason* for unit propagation of a given literal if it was a unit clause before it is asserted, and became true when it is asserted.

**definition**

\[
\text{isReason} :: \text{Clause} \Rightarrow \text{Literal} \Rightarrow \text{Valuation} \Rightarrow \text{bool}
\]

where

\[(\text{isReason} \, \text{clause} \, \text{literal} \, \text{valuation}) \equiv
\begin{align*}
& (\text{literal} \, \epsilon \, \text{clause}) \land \\
& (\text{clauseFalse} \, (\text{removeAll} \, \text{literal} \, \text{clause}) \, \text{valuation}) \land \\
& (\forall \, \text{literal}' \, . \, \text{literal}' \, \epsilon \, (\text{removeAll} \, \text{literal} \, \text{clause}) \rightarrow \\
& \precedes \, (\text{opposite} \, \text{literal}') \, \text{literal} \, \text{valuation} \land \text{opposite} \, \text{literal}' \neq \text{literal})
\end{align*}
\]

**lemma** isReasonAppend:

**fixes** clause :: Clause and literal :: Literal and valuation :: Valuation

**and** valuation' :: Valuation

**assumes** isReason clause literal valuation

**shows** isReason clause literal (valuation @ valuation')

**proof**

- **from** assms

  **have** literal el clause and

  clauseFalse (removeAll literal clause) valuation (is ?false valuation)

  **and**

  \( (\forall \, \text{literal}', \, \text{literal}' \, \epsilon \, (\text{removeAll} \, \text{literal} \, \text{clause}) \rightarrow \\
  \precedes \, (\text{opposite} \, \text{literal}') \, \text{literal} \, \text{valuation} \land \text{opposite} \, \text{literal}' \neq \text{literal} (\text{is} \, ?\text{precedes valuation}))
\]

  **unfolding** isReason-def

  **by** auto

  **moreover**

  **from** (?false valuation)

  **have** ?false (valuation @ valuation')

  **by** (rule clauseFalseAppendValuation)

86
moreover
from \( ?\text{precedes valuation} \)
have \( ?\text{precedes (valuation @ valuation')} \)
  by (simp add:precedesAppend)
ultimately
show \( ?\text{thesis} \)
  unfolding isReason-def
  by auto
qed

lemma isUnitClauseIsReason:
  fixes \( u\text{Clause} :: \text{Clause} \) and \( u\text{Literal} :: \text{Literal} \) and \( \text{valuation} :: \text{Valuation} \)
  assumes isUnitClause \( u\text{Clause} \) \( u\text{Literal} \) \( \text{valuation} \) \( \text{valuation}' \)
  shows isReason \( u\text{Clause} \) \( u\text{Literal} \) (valuation @ valuation')
proof –
  from assms
  have \( u\text{Literal} \) el \( u\text{Clause} \) and \( \neg \text{literalTrue} \) \( u\text{Literal} \) \( \text{valuation} \) and
    \( \neg \text{literalFalse} \) \( u\text{Literal} \) \( \text{valuation} \)
    and \( \forall \) \( \text{literal} \).
      \( \text{literal} \) el \( u\text{Clause} \) \( \land \) \( \text{literal} \neq \) \( u\text{Literal} \) \( \rightarrow \) \( \text{literalFalse} \) \( \text{litera} \)
    unfolding isUnitClause-def
    by auto
  hence \( \text{clauseFalse} \) (removeAll \( u\text{Literal} \) \( u\text{Clause} \) \( \text{valuation} \))
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
  hence \( \text{clauseFalse} \) (removeAll \( u\text{Literal} \) \( u\text{Clause} \) \( \text{valuation} \))
    by (simp add: clauseFalseAppendValuation)
  moreover
  have \( \forall \) \( \text{literal'} \).
    \( \text{literal'} \) el (removeAll \( u\text{Literal} \) \( u\text{Clause} \) ) \( \rightarrow \)
    precedes (opposite literal') \( u\text{Literal} \) (valuation @ valuation') \( \land \)
    (opposite literal') \( \neq \) \( u\text{Literal} \)
proof –
  { 
    fix \( \text{literal'} :: \text{Literal} \)
    assume \( \text{literal'} \) el (removeAll \( u\text{Literal} \) \( u\text{Clause} \) )
    with (clauseFalse (removeAll \( u\text{Literal} \) \( u\text{Clause} \) ) \( \text{valuation} \))
    have \( \text{literalFalse} \) \( \text{literal'} \) \( \text{valuation} \)
      by (simp add: clauseFalseIffAllLiteralsAreFalse)
    with \( \neg \text{literalTrue} \) \( u\text{Literal} \) \( \text{valuation} \) \( \land \) \( \neg \text{literalFalse} \) \( u\text{Literal} \) \( \text{valuation} \)
    using \( \text{uLiteral el valuation'} \)
    using precedesMemberHeadMemberTail | of opposite literal' \( \text{valuation} \)
    \( \text{valuation} \) \( u\text{Literal} \) \( \text{valuation'} \)
    by auto
  }
thus \( \text{thesis} \)
  by simp
qed
ultimately
show \( \text{thesis} \) using (\( u\text{Literal} \) el \( u\text{Clause} \))
  by (auto simp add: isReason-def)
qed

lemma isReasonHoldsInPrefix:
  fixes \( \text{prefix} :: \text{Valuation} \) and \( \text{valuation} :: \text{Valuation} \) and \( \text{clause} :: \text{Clause} \) and \( \text{literal} :: \text{Literal} \)
  assumes
    literal el \( \text{prefix} \) and
    isPrefix \( \text{prefix} \) \( \text{valuation} \) and
    isReason \( \text{clause} \) \( \text{literal} \) \( \text{valuation} \)
  shows
    isReason \( \text{clause} \) \( \text{literal} \) \( \text{prefix} \)
proof −
  from (isReason \( \text{clause} \) \( \text{literal} \) \( \text{valuation} \))
  have
    literal el \( \text{clause} \) and
    clauseFalse (removeAll literal \( \text{clause} \)) \( \text{valuation} \) (is \( ?\text{false} \) \( \text{valuation} \))
and
    \( \forall \: \text{literal} \: \prime. \: \text{literal} \: \prime \: \text{el} \: (\text{removeAll} \: \text{literal} \: \text{clause}) \rightarrow \)
      precedes (opposite \( \text{literal} \: \prime \)) \( \text{literal} \) \( \text{valuation} \) \( \land \) opposite \( \text{literal} \: \prime \)
\( \neq \) literal (is \( ?\text{precedes} \) \( \text{valuation} \))
  unfolding isReason-def
  by auto
{ 
  fix \( \text{literal} \: \prime \) :: Literal
  assume \( \text{literal} \: \prime \: \text{el} \: (\text{removeAll} \: \text{literal} \: \text{clause}) \)
  with (is\( ?\text{precedes} \) \( \text{valuation} \))
  have precedes (opposite \( \text{literal} \: \prime \)) \( \text{literal} \) \( \text{valuation} \) (opposite \( \text{literal} \: \prime \))
  \( \neq \) literal
  by auto
  with (\( \text{litera}l \: \text{el} \: \text{prefix} \) \( \text{isPrefix} \) \( \text{prefix} \) \( \text{valuation} \))
  have precedes (opposite \( \text{literal} \: \prime \)) \( \text{literal} \) \( \text{prefix} \) \( \land \) (opposite \( \text{literal} \: \prime \))
  \( \neq \) literal
  using laterInPrefixRetainsPrecedes [of \( \text{prefix} \) \( \text{valuation} \) opposite \( \text{literal} \: \prime \) \( \text{literal} \)]
  by auto
}

note * = this
hence \( ?\text{precedes} \) \( \text{prefix} \)
  by auto
moreover
have \( ?\text{false} \) \( \text{prefix} \)
proof −
{ 

88
fix \texttt{literal'} :: \texttt{Literal}

assume \texttt{literal'} el (removeAll literal clause)

from \texttt{literal'} el (removeAll literal clause): *

have precedes (opposite literal') literal prefix
  by simp
with (literal el prefix)

have literalFalse literal' prefix
  unfolding precedes-def
  by (auto split: split-if asm)

} thus \texttt{?thesis} by (auto simp add: clauseFalseIffAllLiteralsAreFalse)

qed ultimately

show \texttt{?thesis} using (literal el clause)

unfolding isReason-def
by auto

qed

2.2.16 Last asserted literal of a list

lastAssertedLiteral from a list is the last literal from a clause
that is asserted in a valuation.

definition
isLastAssertedLiteral :: \texttt{Literal} \Rightarrow \texttt{Literal list} \Rightarrow \texttt{Valuation} \Rightarrow \texttt{bool}

where

isLastAssertedLiteral literal clause valuation ==
  literal el clause ∧
  literalTrue literal valuation ∧
  (\forall \texttt{literal'}. literal' el clause ∧ literal' ≠ literal → ¬ precedes literal literal' valuation)

Function that gets the last asserted literal of a list - specified
only by its postcondition.

definition
getLastAssertedLiteral :: \texttt{Literal list} \Rightarrow \texttt{Valuation} \Rightarrow \texttt{Literal}

where

getLastAssertedLiteral clause valuation ==
  last (filter (\texttt{l}:\texttt{Literal}. l el clause) valuation)

lemma getLastAssertedLiteralCharacterization:
assumes
  clauseFalse clause valuation
  clause ≠ []
  uniq valuation

shows
  isLastAssertedLiteral (getLastAssertedLiteral (oppositeLiteralList clause) valuation) (oppositeLiteralList clause) valuation

89
proof
let \( \text{oppc} = \text{oppositeLiteralList} \text{ clause} \)
let \( \text{l} = \text{getLastAssertedLiteral} \text{ ?oppc valuation} \)
let \( \text{f} = \text{filter} (\lambda l. l \text{ el } \text{?oppc}) \text{ valuation} \)

have \( \text{?oppc} \neq [] \)
  using \( \langle \text{clause} \neq [] \rangle \)
  using \( \text{oppositeLiteralListNonempty[of clause]} \)
  by simp
then obtain \( \text{l'}::\text{Literal} \)
  where \( \text{l'} \text{ el } \text{?oppc} \)
  by force

have \( \forall \text{l::Literal}. l \text{ el } \text{?oppc} \rightarrow l \text{ el valuation} \)
proof
fix \( \text{l::Literal} \)
show \( l \text{ el } \text{?oppc} \rightarrow l \text{ el valuation} \)
proof
assume \( l \text{ el } \text{?oppc} \)
  hence \( \text{opposite } l \text{ el clause} \)
  using \( \text{literalElListIffOppositeLiteralElOppositeLiteralList[of } \text{l } \text{?oppc]} \)
  by simp
thus \( l \text{ el valuation} \)
  using \( \langle \text{clauseFalse clause valuation} \rangle \)
  using \( \text{clauseFalseIffAllLiteralsAreFalse[of clause valuation]} \)
  by auto
qed
qed

hence \( \text{l'} \text{ el valuation} \)
  using \( \langle \text{l'} \text{ el } \text{?oppc} \rangle \)
  by simp
hence \( \text{l'} \text{ el } \text{f} \)
  using \( \langle \text{l'} \text{ el } \text{?oppc} \rangle \)
  by simp
hence \( \text{f} \neq [] \)
  using \( \text{set-empty[of } \text{f} \rangle \)
  by auto
hence \( \text{last } \text{f} \text{ el } \text{f} \)
  using \( \text{last-in-set[of } \text{f} \rangle \)
  by simp
hence \( \forall \text{l el } \text{?oppc literalTrue } \text{l valuation} \)
  unfolding \( \text{getLastAssertedLiteral-def} \)
  by auto
moreover
have \( \forall \text{literal', literal' el } \text{?oppc } \land \text{literal' } \neq \text{l} \rightarrow \)
  \( \neg \text{precedes } \text{l literal'} \text{ valuation} \)
proof
fix \( \text{literal'} \)
show \texttt{literal'} el ?oppc \land \texttt{literal'} \neq \?l \rightarrow \neg \texttt{precedes ?l literal'} \texttt{valuation} \\
\textbf{proof} \\
\textbf{assume}\ \texttt{literal'} el ?oppc \land \texttt{literal'} \neq \?l \\
show\ \neg \texttt{precedes ?l literal'} \texttt{valuation} \\
\textbf{proof}\ \langle\texttt{cases literal'True literal'} \texttt{valuation}\rangle \\
\textbf{case}\ \texttt{False} \\
\textbf{thus}\ \?thesis \\
\textbf{unfolding}\ \texttt{precedes-def} \\
\textbf{by}\ \texttt{simp} \\
\textbf{next} \\
\textbf{case}\ \texttt{True} \\
\textbf{with}\ \langle\texttt{literal'} el ?oppc \land \texttt{literal'} \neq \?l\rangle \\
\textbf{have}\ \texttt{literal'} el \?f \\
\textbf{by}\ \texttt{simp} \\
\textbf{have}\ \texttt{uniq \?f} \\
\textbf{using}\ \langle\texttt{uniq \texttt{valuation}}\rangle \\
\textbf{by}\ \langle\texttt{simp add: uniqDistinct}\rangle \\
\textbf{hence}\ \neg \texttt{precedes ?l literal'} \?f \\
\textbf{using}\ \texttt{lastPrecedesNoElement[of \?f]} \\
\textbf{using}\ \langle\texttt{literal'} el ?oppc \land \texttt{literal'} \neq \?l\rangle \\
\textbf{unfolding}\ \texttt{getLastAssertedLiteral-def} \\
\textbf{by}\ \texttt{auto} \\
\textbf{thus}\ \?thesis \\
\textbf{using}\ \texttt{precedesFilter[of \?l \texttt{valuation} \lambda \ l. \ l el ?oppc]} \\
\textbf{using}\ \langle\texttt{literal'} el ?oppc \land \texttt{literal'} \neq \?l\rangle \\
\textbf{using}\ \langle\texttt{?f el ?oppc}\rangle \\
\textbf{by}\ \texttt{auto} \\
\textbf{qed} \\
\textbf{qed} \\
\textbf{qed} \\
\textbf{ultimately} \\
\textbf{show}\ \?thesis \\
\textbf{unfolding}\ \texttt{isLastAssertedLiteral-def} \\
\textbf{by}\ \texttt{simp} \\
\textbf{qed} \\

\textbf{lemma lastAssertedLiteralIsUniq:} \\
\textbf{fixes}\ \texttt{literal}::\ \texttt{Literal} \texttt{and}\ \texttt{literal'}::\ \texttt{Literal} \texttt{and}\ \texttt{literalList}::\ \texttt{Literal list} \texttt{and}\ \texttt{valuation}::\ \texttt{Valuation} \\
\textbf{assumes} \\
\texttt{lastL}::\ \texttt{isLastAssertedLiteral}\ \texttt{literal}\ \texttt{literalList}\ \texttt{valuation} \texttt{and} \\
\texttt{lastL'}::\ \texttt{isLastAssertedLiteral}\ \texttt{literal'}\ \texttt{literalList}\ \texttt{valuation} \\
\textbf{shows}\ \texttt{literal} = \texttt{literal'} \\
\textbf{using}\ \texttt{assms} \\
\textbf{proof}\ = \\
\textbf{from}\ \texttt{lastL have \_:} \\
\text{\texttt{literal el literalList} \\
\text{\\quad \\& l el literalList \land l \neq \texttt{literal} \rightarrow \neg \texttt{precedes \texttt{literal}}\ \texttt{l valuation}} \\
\quad 91
and
literalTrue literal valuation
by (auto simp add: isLastAssertedLiteral-def)
from lastL' have **:
literal' el literalList
∀ l. l el literalList ∧ l ≠ literal' —→ ¬ precedes literal' l valuation
and
literalTrue literal' valuation
by (auto simp add: isLastAssertedLiteral-def)
{
  assume literal' ≠ literal
  with * ** have ¬ precedes literal literal' valuation and ¬ precedes
  literal' literal valuation
  by auto
  with ⟨literalTrue literal valuation⟩ ⟨literalTrue literal' valuation⟩
  have False
    using precedesTotalOrder[of literal valuation literal']
    unfolding precedes-def
    by simp
}
thus ?thesis
by auto
qed

lemma isLastAssertedCharacterization:
  fixes literal :: Literal and literalList :: Literal list and v :: Valuation
  assumes isLastAssertedLiteral literal (oppositeLiteralList literalList)
  valuation
  shows opposite literal el literalList and literalTrue literal valuation
proof —
  from assms have
    *: literal el (oppositeLiteralList literalList) and **: literalTrue literal valuation
    by (auto simp add: isLastAssertedLiteral-def)
  from * show opposite literal el literalList
    using literalELListIffOppositeLiteralELOppositeLiteralList[of literal
    oppositeLiteralList literalList]
    by simp
  from ** show literalTrue literal valuation
    by simp
qed

lemma isLastAssertedLiteralSubset:
  assumes
    isLastAssertedLiteral l c M
    set c' ⊆ set c
    l el c'
  shows
    isLastAssertedLiteral l c' M

92
using assms
unfolding isLastAssertedLiteral-def
by auto

lemma lastAssertedLastInValuation:
  fixes literal :: Literal and literalList :: Literal list and valuation :: Valuation
  assumes literal el literalList and ¬ literalTrue literal valuation
  shows isLastAssertedLiteral literal literalList (valuation @ [literal])
proof –
  have literalTrue literal [literal]
    by simp
  hence literalTrue literal (valuation @ [literal])
    by simp
  moreover
  have ∀ l. l el literalList ∧ l ≠ literal → ¬ precedes literal l
    (valuation @ [literal])
proof –
  { fix l
    assume l el literalList l ≠ literal
    have ¬ precedes literal l (valuation @ [literal])
      proof (cases literalTrue l valuation)
      case False
        with ⟨ l̸= literal ⟩
        show ?thesis
          unfolding precedes-def
          by simp
      next
      case True
      from ⟨ ¬ literalTrue literal valuation ⟩ ⟨ literalTrue literal [literal] ⟩
      ⟨ literalTrue l valuation ⟩
      have precedes l literal (valuation @ [literal])
        using precedesMemberHeadMemberTail[of l valuation literal
        [literal]]
        by auto
      with ⟨ l̸= literal ⟩ ⟨ literalTrue l valuation ⟩ ⟨ literalTrue literal
        [literal] ⟩
      show ?thesis
        using precedesAntisymmetry[of l valuation @ [literal] literal]
        unfolding precedes-def
        by auto
    qed
  } thus ?thesis
  by simp
qed
ultimately
show ?thesis using (literal el literalList)
  by (simp add:isLastAssertedLiteral-def)
3 Trail datatype definition and its properties

theory Trail
imports MoreList
begin

Trail is a list in which some elements can be marked.

type-synonym 'a Trail = ('a*bool) list

abbreviation element :: ('a*bool) ⇒ 'a
where element x == fst x

abbreviation marked :: ('a*bool) ⇒ bool
where marked x == snd x

3.1 Trail elements

Elements of the trail with marks removed

primrec elements :: 'a Trail ⇒ 'a list
where
  elements [] = []
| elements (h#t) = (element h) # (elements t)

lemma elements t = map fst t
by (induct t) auto

lemma eitherMarkedOrNotMarkedElement:
  shows a = (element a, True) ∨ a = (element a, False)
by (cases a) auto

lemma eitherMarkedOrNotMarked:
  assumes e ∈ set (elements M)
  shows (e, True) ∈ set M ∨ (e, False) ∈ set M
using assms
proof (induct M)
case (Cons m M′)
thus ?case
  proof (cases e = element m)
case True
thus ?thesis
  using eitherMarkedOrNotMarkedElement [of m]
  by auto
next
  case False
  with Cons
  show ?thesis
  by auto
qed
qed simp

lemma elementMemElements [simp]:
  assumes x ∈ set M
  shows element x ∈ set (elements M)
using assms
by (induct M) (auto split: split-if-asm)

lemma elementsAppend [simp]:
  shows elements (a @ b) = elements a @ elements b
by (induct a) auto

lemma elementsEmptyIffTrailEmpty [simp]:
  shows (elements list = []) = (list = [])
by (induct list) auto

lemma elementsButlastTrailIsButlastElementsTrail [simp]:
  shows elements (butlast M) = butlast (elements M)
by (induct M) auto

lemma elementLastTrailIsLastElementsTrail [simp]:
  assumes M ≠ []
  shows element (last M) = last (elements M)
using assms
by (induct M) auto

lemma isPrefixElements:
  assumes isPrefix a b
  shows isPrefix (elements a) (elements b)
using assms
unfolding isPrefix-def
by auto

lemma prefixElementsAreTrailElements:
  assumes isPrefix p M
  shows set (elements p) ⊆ set (elements M)
using assms
unfolding \textit{isPrefix-def}  
by \textit{auto}

\textbf{lemma} \textit{uniqElementsTrailImpliesUniqElementsPrefix}:  
assumes \textit{isPrefix} \(p\ M\) and \(\text{uniq}\ (\text{elements}\ M)\)  
shows \(\text{uniq}\ (\text{elements}\ p)\)  
\textbf{proof} –  
from \(\langle \text{isPrefix}\ p\ M \rangle\)  
obtain \(s\)  
where \(M = p \circledast s\)  
unfolding \textit{isPrefix-def}  
by \textit{auto}\  
with \(\langle \text{uniq}\ (\text{elements}\ M) \rangle\)  
show \(?\text{thesis}\)  
using \textit{uniqAppendIff}[of \text{elements}\ p\ \text{elements}\ s]\  
by \textit{simp}\  
\textit{qed}

\textbf{lemma} \textit{[simp]}:  
assumes \((e, d)\ \in\ \text{set}\ M\)\  
shows \(e\ \in\ \text{set}\ (\text{elements}\ M)\)  
using \textit{assms}\  
by \((\text{induct}\ M)\) \textit{auto}\  

\textbf{lemma} \textit{uniqImpliesExclusiveTrueOrFalse}:  
assumes \((e, d)\ \in\ \text{set}\ M\) and \(\text{uniq}\ (\text{elements}\ M)\)  
shows \(\neg\ (e, \neg\ d)\ \in\ \text{set}\ M\)\  
using \textit{assms}\  
\textbf{proof} \((\text{induct}\ M)\)  
\textbf{case} \((\text{Cons}\ m\ M')\)  
\{  
\textit{assume} \((e, d) = m\)  
\textit{hence} \((e, \neg\ d) \neq m\)  
\textit{by} \textit{auto}\  
from \(\langle (e, d) = m \rangle\) \(\langle \text{uniq}\ (\text{elements}\ (m \# M')) \rangle\)  
\textit{have} \(\neg\ (e, d)\ \in\ \text{set}\ M'\)  
\textit{by} \((\text{auto} \text{ simp add: uniqAppendIff})\)  
\textit{with} \textit{Cons}\  
\textit{have} \(?\text{case}\)  
\textit{by} \((\text{auto} \text{ split: split-if-asm})\)  
\}\  
\textit{moreover}\  
\{  
\textit{assume} \((e, \neg\ d) = m\)  
\textit{hence} \((e, d) \neq m\)  
\}  
96
by auto
from \((e, \neg d) = m\)  \langle uniq\ (elements\ (m \# M'))\rangle
have \(\neg (e, \neg d) \in set\ M'\)
  by (auto simp add: uniqAppendIff)
with Cons
have ?case
  by (auto split: split-if-asm)
}\nmoreover
\{ 
  assume \((e, d) \neq m\) \((e, \neg d) \neq m\)
from \((e, d) \neq m\): \((e, d) \in set\ (m \# M')\)
  have \((e, d) \in set\ M'\)
    by simp
with \langle uniq\ (elements\ (m \# M'))\rangle: Cons(1)
  have \(\neg (e, \neg d) \in set\ M'\)
    by simp
with \((e, \neg d) \neq m\)
  have ?case
    by simp
\}
moreover
\{ 
  have \((e, d) = m \vee (e, \neg d) = m \vee (e, d) \neq m \land (e, \neg d) \neq m\)
    by auto
\}
ultimately
show ?case
  by auto
qed simp

3.2 Marked trail elements

primrec
markedElements :: 'a Trail \Rightarrow 'a list
where
markedElements [] = []
| markedElements (h#t) = (if marked h then (element h) \# (markedElements t) else (markedElements t))

lemma
markedElements t = (elements (filter snd t))
by (induct t) auto

lemma markedElementIsMarkedTrue:
  shows \((m \in set\ (markedElements\ M)) = ((m, True) \in set\ M)\)
using assms
by (induct M) (auto split: split-if-asm)
lemma markedElementsAppend:
  shows markedElements (M1 @ M2) = markedElements M1 @ markedElements M2
by (induct M1) auto

lemma markedElementsAreElements:
  assumes m ∈ set (markedElements M)
  shows m ∈ set (elements M)
using assms markedElementIsMarkedTrue[of m M]
by auto

lemma markedAndMemberImpliesIsMarkedElement:
  assumes marked m m ∈ set M
  shows (element m) ∈ set (markedElements M)
proof
  have m = (element m, marked m)
    by auto
  with ⟨marked m⟩
  have m = (element m, True)
    by simp
  with ⟨m ∈ set M⟩
  have (element m, True) ∈ set M
    by simp
  thus ?thesis
    using markedElementIsMarkedTrue[of element m M]
    by simp
qed

lemma markedElementsPrefixAreMarkedElementsTrail:
  assumes isPrefix p M m ∈ set (markedElements p)
  shows m ∈ set (markedElements M)
proof
  from ⟨m ∈ set (markedElements p)⟩
  have (m, True) ∈ set p
    by (simp add: markedElementIsMarkedTrue)
  with ⟨isPrefix p M⟩
  have (m, True) ∈ set M
    using prefixIsSubset[of p M]
    by auto
  thus ?thesis
    by (simp add: markedElementIsMarkedTrue)
qed

lemma markedElementsTrailMemPrefixAreMarkedElementsPrefix:
  assumes uniq (elements M) and
  isPrefix p M and
  m ∈ set (elements p) and
  m ∈ set (markedElements M)
shows 
\[ m \in \text{set} (\text{markedElements } p) \]

proof –
\[
\text{from } \langle m \in \text{set} (\text{markedElements } M) \rangle \text{ have } (m, \text{True}) \in \text{set } M \\
\text{by (simp add: markedElementIsMarkedTrue)}
\]
with \(\langle \text{uniq } (\text{elements } M) \rangle \cap m \in \text{set } (\text{elements } p)\)
have \( (m, \text{True}) \in \text{set } p \)
proof –
\{
assume \( (m, \text{False}) \in \text{set } p \)
with \(\langle \text{isPrefix } p M \rangle \)
have \( (m, \text{False}) \in \text{set } M \)
using \(\text{prefixIsSubset[of } p M]\)
by auto
with \( (m, \text{True}) \in \text{set } M \) \(\langle \text{uniq } (\text{elements } M) \rangle \)
have False
using \(\text{uniqImpliesExclusiveTrueOrFalse[of } m \text{ True } M]\)
by simp
\}
with \( (m \in \text{set } (\text{elements } p)) \)
show \(?thesis\)
using \(\text{eitherMarkedOrNotMarked[of } m p]\)
by auto
qed

thus \(?thesis\)
using \(\text{markedElementIsMarkedTrue[of } m p]\)
by simp
qed

3.3 Prefix before/upto a trail element

Elements of the trail before the first occurrence of a given element
- not including it

primrec
\text{prefixBeforeElement} :: 'a \Rightarrow 'a Trail \Rightarrow 'a Trail
where
\text{prefixBeforeElement} e [] = []
| \text{prefixBeforeElement} e (h#t) =
(\text{if } (element h) = e \text{ then}
  []
else
  (h # (prefixBeforeElement e t)))
\}

lemma \text{prefixBeforeElement} e t = \text{takeWhile } (\lambda e'. \text{element } e' \neq e) t
by (induct t) auto

lemma \text{prefixBeforeElement} e t = \text{take } (\text{firstPos } e \text{ (elements } t)) t
by (induct t) auto
Elements of the trail before the first occurrence of a given element - including it

**primrec**

\[
\text{prefixToElement} :: \text{'}a \Rightarrow \text{'}a \\text{Trail} \Rightarrow \text{'}a \\text{Trail}
\]

**where**

\[
\text{prefixToElement} \ e \ [] = []
\]

\[
\mid \text{prefixToElement} \ e \ (h \# t) =
\]

\[
\begin{cases}
\text{if} \ (\text{element} \ h) = e \ \text{then} \\
\ [h] \\
\ \text{else} \\
\ (h \# (\text{prefixToElement} \ e \ t))
\end{cases}
\]

**lemma** \( \text{prefixToElement} \ e \ t = \text{take} ((\text{firstPos} \ e \ (\text{elements} \ t)) + 1) \ t \)

\by (induct \ t) \ auto

**lemma** \( \text{isPrefixPrefixToElement} : \)

\shows \( \text{isPrefix} \ (\text{prefixToElement} \ e \ t) \ t \)

\unfolding \( \text{isPrefix-def} \)

\by (induct \ t) \ auto

**lemma** \( \text{isPrefixPrefixBeforeElement} : \)

\shows \( \text{isPrefix} \ (\text{prefixBeforeElement} \ e \ t) \ t \)

\unfolding \( \text{isPrefix-def} \)

\by (induct \ t) \ auto

**lemma** \( \text{prefixToElementContainsTrailElement} : \)

\assumes \( e \in \text{set} \ (\text{elements} \ M) \)

\shows \( e \in \text{set} \ (\text{elements} \ (\text{prefixToElement} \ e \ M)) \)

\using \( \text{assms} \)

\by (induct \ M) \ auto

**lemma** \( \text{prefixBeforeElementDoesNotContainTrailElement} : \)

\assumes \( e \in \text{set} \ (\text{elements} \ M) \)

\shows \( e \notin \text{set} \ (\text{elements} \ (\text{prefixBeforeElement} \ e \ M)) \)

\using \( \text{assms} \)

\by (induct \ M) \ auto

**lemma** \( \text{prefixToElementAppend} : \)

\shows \( \text{prefixToElement} \ e \ (M1 @ M2) = \)

\(\begin{cases}
\text{if} \ e \in \text{set} \ (\text{elements} \ M1) \ \text{then} \\
\ (\text{prefixToElement} \ e \ M1)
\ \text{else} \\
\ M1 @ \text{prefixToElement} \ e \ M2
\end{cases}\)

\by (induct \ M1) \ auto
lemma prefixToElementToPrefixElement:
  assumes isPrefix p M and e ∈ set (elements p)
  shows prefixToElement e M = prefixToElement e p
  using assms
  unfolding isPrefix-def
proof (induct p arbitrary: M)
  case (Cons a p')
  then obtain s
    where (a # p') @ s = M
    by auto
  show ?case
  proof (cases (element a) = e)
    case True
    from True ⟨(a # p') @ s = M⟩ have prefixToElement e M = [a]
    by auto
    moreover
    from True have prefixToElement e (a # p') = [a]
    by auto
    ultimately
    show ?thesis
    by simp
  next
    case False
    from False ⟨(a # p') @ s = M⟩ have prefixToElement e M = a
    # prefixToElement e (p' @ s)
    by auto
    moreover
    from False have prefixToElement e (a # p') = a # prefixToElement e p'
    by simp
    moreover
    from False ⟨e ∈ set (elements (a # p'))⟩ have e ∈ set (elements p')
    by simp
    have ? s . (p' @ s = p' @ s)
    by simp
    from ⟨e ∈ set (elements p')⟩ ⟨? s . (p' @ s = p' @ s)⟩
    have prefixToElement e (p' @ s) = prefixToElement e p'
    using Cons(1) [of p' @ s]
    by simp
    ultimately show ?thesis
    by simp
  qed
qed simp
3.4 Marked elements upto a given trail element

Marked elements of the trail upto the given element (which is also included if it is marked)

definition
markedElementsTo :: 'a ⇒ 'a Trail ⇒ 'a list
where
markedElementsTo e t = markedElements (prefixToElement e t)

lemma markedElementsToArePrefixOfMarkedElements:
  shows isPrefix (markedElementsTo e M) (markedElements M)
unfolding isPrefix-def
unfolding markedElementsTo-def
by (induct M) auto

lemma markedElementsToAreMarkedElements:
  assumes m ∈ set (markedElementsTo e M)
  shows m ∈ set (markedElements M)
using assms
using markedElementsToArePrefixOfMarkedElements[of e M]
using prefixIsSubset
by auto

lemma markedElementsToNonMemberAreAllMarkedElements:
  assumes e /∈ set (elements M)
  shows markedElementsTo e M = markedElements M
using assms
unfolding markedElementsTo-def
by (induct M) auto

lemma markedElementsToAppend:
  shows markedElementsTo e (M1 @ M2) =
    (if e ∈ set (elements M1) then
     markedElementsTo e M1
   else
    markedElements M1 @ markedElementsTo e M2
  )
unfolding markedElementsTo-def
by (auto simp add: prefixToElementAppend markedElementsAppend)

lemma markedElementsEmptyImpliesMarkedElementsToEmpty:
  assumes markedElements M = []
  shows markedElementsTo e M = []
using assms
using markedElementsToArePrefixOfMarkedElements[of e M]
unfolding isPrefix-def
by auto

lemma markedElementIsMemberOfItsMarkedElementsTo:
assumes
uniq (elements M) and marked e and e ∈ set M
shows
element e ∈ set (markedElementsTo (element e) M)
using assms
unfolding markedElementsTo-def
by (induct M) (auto split: split-if-asm)

lemma markedElementsToPrefixElement:
assumes isPrefix p M and e ∈ set (elements p)
shows markedElementsTo e M = markedElementsTo e p
unfolding markedElementsTo-def
using assms
by (simp add: prefixToElementToPrefixElement)

3.5 Last marked element in a trail

definition
lastMarked :: 'a Trail ⇒ 'a
where
lastMarked t = last (markedElements t)

lemma lastMarkedIsMarkedElement:
assumes markedElements M ≠ []
shows lastMarked M ∈ set (markedElements M)
using assms
unfolding lastMarked-def
by simp

lemma removeLastMarkedFromMarkedElementsToLastMarkedAreAllMarkedElementsInPrefixLastMarked:
assumes
markedElements M ≠ []
shows
removeAll (lastMarked M) (markedElementsTo (lastMarked M) M) = markedElements (prefixBeforeElement (lastMarked M) M)
using assms
unfolding lastMarked-def
unfolding markedElementsTo-def
by (induct M) auto

lemma markedElementsToLastMarkedAreAllMarkedElements:
assumes
uniq (elements M) and markedElements M ≠ []
shows
markedElementsTo (lastMarked M) M = markedElements M
using assms
unfolding lastMarked-def
unfolding markedElementsTo-def
by (induct M) (auto simp add: markedElementsAreElements)

lemma lastTrailElementMarkedImpliesMarkedElementsToLastElementAreAllMarkedElements:
  assumes marked (last M) and last (elements M) \notin set (butlast (elements M))
  shows markedElementsTo (last (elements M)) M = markedElements M
using assms unfolding markedElementsTo-def
by (induct M) auto

lemma lastMarkedIsMemberOfItsMarkedElementsTo:
  assumes uniq (elements M) and markedElements M \neq []
  shows lastMarked M \in set (markedElementsTo (lastMarked M) M)
using assms unfolding markedElementsToLastMarkedAreAllMarkedElements [of M]
using lastMarkedIsMarkedElement [of M]
by auto

lemma lastTrailElementNotMarkedImpliesMarkedElementsToLAreMarkedElementsToLInButlastTrail:
  assumes \neg marked (last M)
  shows markedElementsTo e M = markedElementsTo e (butlast M)
using assms unfolding markedElementsTo-def
by (induct M) auto

3.6 Level of a trail element

Level of an element is the number of marked elements that precede it

definition elementLevel :: 'a \Rightarrow 'a Trail \Rightarrow nat
where elementLevel e t = length (markedElementsTo e t)

lemma elementLevelMarkedGeq1:
  assumes uniq (elements M) and e \in set (markedElements M)
  shows elementLevel e M \geq 1
proof-
  from (e \in set (markedElements M)) have (e, True) \in set M
  by (simp add: markedElementIsMarkedTrue)
  with (uniq (elements M)) have e \in set (markedElementsTo e M)
using \( \text{markedElementIsMemberOfItsMarkedElementsTo}(e, M, True) \)

by simp

hence \( \text{markedElementsTo } e \ M \neq [] \)

by auto

thus \( ?\text{thesis} \)

unfolding \( \text{elementLevel-def} \)

using \( \text{length-greater-0-conv}[\text{of markedElementsTo } e \ M] \)

by arith

qed

lemma \( \text{elementLevelAppend} \):

assumes \( a \in \text{set}(\text{elements } M) \)

shows \( \text{elementLevel } a \ M = \text{elementLevel } a \ (M @ M') \)

using \( \text{assms} \)

unfolding \( \text{elementLevel-def} \)

by (simp add: \( \text{markedElementsToAppend} \))

lemma \( \text{elementLevelPrecedesLeq} \):

assumes \( \text{precedes } a \ b (\text{elements } M) \)

shows \( \text{elementLevel } a \ M \leq \text{elementLevel } b \ M \)

using \( \text{assms} \)

proof (induct \( M \))

case \( (\text{Cons } m \ M') \)

\{ 
  assume \( a = \text{element } m \)
  hence \( ?\text{case} \)
    unfolding \( \text{elementLevel-def} \)
    unfolding \( \text{markedElementsTo-def} \)
    by simp
  \}

moreover \{ 
  assume \( b = \text{element } m \)
  \{
    assume \( a \neq b \)
    hence \( \neg \text{precedes } a \ b (b \# (\text{elements } M')) \)
      by (rule \( \text{noElementsPrecedesFirstElement} \))
    with \( b = \text{element } m \) \( \text{precedes } a \ b (\text{elements } (m \# M')) \)
    have \( \text{False} \)
      by simp
  \}
  hence \( a = b \)
    by auto
  hence \( ?\text{case} \)
    by simp
\}

qed
moreover 

{ 
  assume \( a \neq \text{element } m \neq \text{element } b \) 
  moreover 
  from \( \prec \) \( a \ \prec \ b \) (\( \text{elements } (m \neq M') \)) 
  have \( a \in \text{set } (\text{elements } (m \neq M')) \) \( b \in \text{set } (\text{elements } (m \neq M')) \) 
    unfolding \( \prec \) \( \text{def} \) 
    by (auto split: split-if-alist) 
  from \( a \neq \text{element } m \) \( a \in \text{set } (\text{elements } (m \neq M')) \) 
  have \( a \in \text{set } (\text{elements } M) \) 
    by simp 
  moreover 
  from \( b \neq \text{element } m \) \( b \in \text{set } (\text{elements } (m \neq M')) \) 
  have \( b \in \text{set } (\text{elements } M) \) 
    by simp 
  ultimately 
  have \( \text{elementLevel } a \ M' \leq \text{elementLevel } b \ M' \) 
    using \text{Cons} 
    unfolding \( \prec \) \( \text{def} \) 
    by auto 
  hence \(? \) \( \text{case} \) 
    using \( a \neq \text{element } m \neq \text{element } m \) \( b \neq \text{element } m \) 
    unfolding \( \text{elementLevel-def} \) 
    unfolding \( \text{markedElementsTo-def} \) 
    by auto 
} 
ultimately 
show \(? \) \( \text{case} \) 
by auto 
next 
  case \( \text{Nil} \) 
thus \(? \) \( \text{case} \) 
  unfolding \( \prec \) \( \text{def} \) 
  by simp 
qed

lemma \text{elementLevelPrecedesMarkedElementLt}: 
assumes 
  uniq (\( \text{elements } M \)) \ and 
  \( e \neq d \) \ and 
  \( d \in \text{set } (\text{markedElements } M) \) \ and 
  \( \text{precedes } e \ d \) (\( \text{elements } M \)) 
shows 
  \( \text{elementLevel } e \ M < \text{elementLevel } d \ M \) 
using \( \text{assms} \) 
proof (induct \( M \)) 
  case (\( \text{Cons } m \ M' \))
\{ 
  assume \( e = \text{element} \ m \)
  moreover 
  with \( e \neq d \) have \( d \neq \text{element} \ m \)
    by simp 
  moreover 
  from \( \langle \text{uniq} (\text{elements} \ (m \# M')) \rangle \langle d \in \text{set} \ (\text{markedElements} \ (m \# M')) \rangle \)
  have \( 1 \leq \text{elementLevel} \ d \ (m \# M') \)
    using \( \text{elementLevelMarkedGeq1} [\text{of} \ m \# M' \ d] \)
    by auto 
  moreover 
  from \( \langle d \neq \text{element} \ m \rangle \langle d \in \text{set} \ (\text{markedElements} \ (m \# M')) \rangle \)
  have \( d \in \text{set} \ (\text{markedElements} \ M') \)
    by (simp split: split-if-asm) 
  from \( \langle \text{uniq} (\text{elements} \ (m \# M')) \rangle \langle d \in \text{set} \ (\text{markedElements} \ M') \rangle \)
  have \( 1 \leq \text{elementLevel} \ d M' \)
    using \( \text{elementLevelMarkedGeq1} [\text{of} \ M' \ d] \)
    by auto 
  ultimately 
  have ?case 
    unfolding \( \text{elementLevel-def} \)
    unfolding \( \text{markedElementsTo-def} \)
    by (auto split: split-if-asm) 
\} 
moreover 
\{ 
  assume \( d = \text{element} \ m \)
  from \( \langle e \neq d \rangle \text{ have } \neg \text{precedes} \ e \ d \ (d \neq (\text{elements} \ M')) \)
  using \( \text{noElementsPrecedesFirstElement} [\text{of} \ e \ d \ \text{elements} \ M'] \)
    by simp 
  with \( \langle d = \text{element} \ m \rangle \langle \text{precedes} \ e \ d \ (\text{elements} \ (m \# M')) \rangle \)
  have \( \text{False} \)
    by simp 
  hence ?case 
    by simp 
\} 
moreover 
\{ 
  assume \( e \neq \text{element} \ m \ d \neq \text{element} \ m \)
  moreover 
  from \( \langle \text{precedes} \ e \ d \ (\text{elements} \ (m \# M')) \rangle \)
  have \( e \in \text{set} \ (\text{elements} \ (m \# M')) \ d \in \text{set} \ (\text{elements} \ (m \# M')) \)
    unfolding \( \text{precedes-def} \)
    by (auto split: split-if-asm) 
  from \( \langle e \neq \text{element} \ m \ e \in \text{set} \ (\text{elements} \ (m \# M')) \rangle \)
  have \( e \in \text{set} \ (\text{elements} \ M') \)
    by simp 
  moreover 
\}
from \( \langle d \neq \text{element } m \rangle \mid d \in \text{set } (\text{elements } (m \neq M')) \rangle \\
\text{have } d \in \text{set } (\text{elements } M') \\
\text{by simp}

moreover
from \( \langle d \neq \text{element } m \rangle \mid d \in \text{set } (\text{markedElements } (m \neq M')) \rangle \\
\text{have } d \in \text{set } (\text{markedElements } M') \\
\text{by } (\text{simp split: split-if-asn})

ultimately
\text{have } \text{elementLevel } e M' < \text{elementLevel } d M' \\
\text{using } \langle \text{uniq } (\text{elements } (m \neq M')) \rangle \text{ Cons}
\text{unfolding precedes-def}
\text{by auto}
\text{hence } ?\text{case}
\text{using } \langle e \neq \text{element } m \rangle \text{ and } d \neq \text{element } m
\text{unfolding } \text{elementLevel-def}
\text{unfolding } \text{markedElementsTo-def}
\text{by auto}

} \\
\text{ultimately}
\text{show } ?\text{case}
\text{by auto}
\text{qed simp}

\text{lemma } \text{differentMarkedElementsHaveDifferentLevels:}
\text{assumes}
\text{uniq } (\text{elements } M) \text{ and}
\text{a } \in \text{set } (\text{markedElements } M) \text{ and}
\text{b } \in \text{set } (\text{markedElements } M) \text{ and}
\text{a } \neq b
\text{shows } \text{elementLevel } a M \neq \text{elementLevel } b M

\text{proof }–
\text{from } \langle a \in \text{set } (\text{markedElements } M) \rangle \\
\text{have } a \in \text{set } (\text{elements } M) \\
\text{by } (\text{simp add: markedElementsAreElements})
\text{moreover}
\text{from } \langle b \in \text{set } (\text{markedElements } M) \rangle \\
\text{have } b \in \text{set } (\text{elements } M) \\
\text{by } (\text{simp add: markedElementsAreElements})
\text{ultimately}
\text{have } \text{precedes } a b (\text{elements } M) \lor \text{precedes } b a (\text{elements } M) \\
\text{using } (a \neq b)
\text{using } \text{precedesTotalOrder}[\text{of } a \text{ elements } M b]
\text{by simp}
\text{moreover}
\{ \\
\text{assume } \text{precedes } a b (\text{elements } M)
\text{with } \text{assms}
\text{have } ?\text{thesis}
\text{using } \text{elementLevelPrecedesMarkedElementLt}[\text{of } M a b]

108
by auto
}

moreover
{
assume precedes b a (elements M)
with assms
have thesis
  using elementLevelPrecedesMarkedElementLt[of M b a]
  by auto
}

ultimately
show thesis
  by auto
qed

3.7 Current trail level

Current level is the number of marked elements in the trail

definition
  currentLevel :: 'a Trail ⇒ nat

where
  currentLevel t = length (markedElements t)

lemma currentLevelNonMarked:
  shows currentLevel M = currentLevel (M @ [(l, False)])
  by (auto simp add: currentLevel-def markedElementsAppend)

lemma currentLevelPrefix:
  assumes isPrefix a b
  shows currentLevel a <= currentLevel b
  using assms
  unfolding isPrefix-def
  unfolding currentLevel-def
  by (auto simp add: markedElementsAppend)

lemma elementLevelLeqCurrentLevel:
  shows elementLevel a M <= currentLevel M

proof−
  have isPrefix (prefixToElement a M) M
    using isPrefixPrefixToElement[of a M]
  .
  then obtain s
    where prefixToElement a M @ s = M
    unfolding isPrefix-def
    by auto
  hence M = prefixToElement a M @ s
    by (rule sym)
  hence currentLevel M = currentLevel (prefixToElement a M @ s)
    by simp
hence currentLevel \( M \) = length (markedElements (prefixToElement a M)) + length (markedElements s)

unfolding currentLevel-def
by (simp add: markedElementsAppend)
thus \(?thesis
unfolding elementLevel-def
unfolding markedElementsTo-def
by simp
qed

lemma elementOnCurrentLevel:
assumes a \( \in \) set (elements M)
shows elementLevel a (M @ [(a, d)]) = currentLevel (M @ [(a, d)])
using assms
unfolding currentLevel-def
unfolding elementLevel-def
unfolding markedElementsTo-def
by (auto simp add: prefixToElementAppend)

3.8 Prefix to a given trail level

Prefix is made or elements of the trail up to a given element level

primrec
prefixToLevel-aux :: 'a Trail \Rightarrow nat \Rightarrow nat \Rightarrow 'a Trail
where
(prefixToLevel-aux [] l cl) = []
| (prefixToLevel-aux (h#t) l cl) =
  (if (marked h) then
    (if (cl > l) then [] else (h # (prefixToLevel-aux t l (cl+1))))
  else
    (h # (prefixToLevel-aux t l cl)))
)
definition
prefixToLevel :: nat \Rightarrow 'a Trail \Rightarrow 'a Trail
where
prefixToLevel-def: (prefixToLevel l t) == (prefixToLevel-aux t l 0)

lemma isPrefixPrefixToLevel-aux:
shows \( \exists \ s . \ prefixToLevel-aux t l i \ @ s = t \)
by (induct t arbitrary: i) auto

lemma isPrefixPrefixToLevel:
shows (isPrefix (prefixToLevel l t) t)
using isPrefixPrefixToLevel-aux[of t l]
unfolding isPrefix-def
unfolding prefixToLevel-def
lemma currentLevelPrefixToLevel-aux:
  assumes \( l \geq i \)
  shows \( \text{currentLevel} \ (\text{prefixToLevel-aux} \ M \ l \ i) \leq l - i \)
using assms
proof (induct \( M \) arbitrary: \( i \))
case (\( \text{Cons} \ m \ M \))
  {
    assume \( \text{marked} \ m \ i = l \)
    hence \ ?case
      unfolding currentLevel-def
    by simp
  }
moreover
  {
    assume \( \text{marked} \ m \ i < l \)
    hence \ ?case
      using Cons \( \{\text{of} \ i + 1\} \)
      unfolding currentLevel-def
    by simp
  }
moreover
  {
    assume \( \neg \text{marked} \ m \)
    hence \ ?case
      using Cons
      unfolding currentLevel-def
    by simp
  }
ultimately
show \ ?case
  using \( \langle i \leq l \rangle \)
  by auto
next
case Nil
  thus \ ?case
    unfolding currentLevel-def
    by simp
qed

lemma currentLevelPrefixToLevel:
  shows \( \text{currentLevel} \ (\text{prefixToLevel} \ \text{level} \ M) \leq \text{level} \)
using currentLevelPrefixToLevel-aux[\( \text{of} \ 0 \ \text{level} \ M \)]
unfolding prefixToLevel-def
by simp

lemma currentLevelPrefixToLevelEq-aux:
  assumes \( l \geq i \ \text{currentLevel} \ M \geq l - i \)
shows $\text{currentLevel} \ (\text{prefixToLevel-aux} \ M \ l \ i) = l - i$

using assms

proof (induct $M$ arbitrary: $i$)

  case (Cons $m$ $M$)

  \{ 
    assume marked $m$ $i$ = $l$
    hence:?case
      unfolding $\text{currentLevel-def}$
      by simp
  \}

moreover

  \{ 
    assume marked $m$ $i$ < $l$
    hence:?case
      using Cons(1) [of $i$+1]
      using Cons(3)
      unfolding $\text{currentLevel-def}$
      by simp
  \}

moreover

  \{ 
    assume $\neg$ marked $m$
    hence:?case
      using Cons
      unfolding $\text{currentLevel-def}$
      by simp
  \}

ultimately

  show:?case
    using (i <= b)
    by auto

next

  case Nil
  thus:?case
    unfolding $\text{currentLevel-def}$
    by simp

qed

lemma $\text{currentLevelPrefixToLevelEq}$:

assumes
    \( \text{level} \leq \text{currentLevel} \ M \)

shows
    $\text{currentLevel} \ (\text{prefixToLevel} \ \text{level} \ M) = \text{level}$

using assms

unfolding $\text{prefixToLevel-def}$

using $\text{currentLevelPrefixToLevelEq-aux}[\text{of 0} \ \text{level} \ M]$

by simp

lemma $\text{prefixToLevel-auxIncreaseAuxiliaryCounter}$:
assumes $k \geq i$

shows $\text{prefixToLevel-aux } M \cup i = \text{prefixToLevel-aux } M \cup (l + (k - i))$

proof (induct $M$ arbitrary: $i$ $k$)

\begin{itemize}
\item case ($\text{Cons } m \ M'$)
\end{itemize}

\begin{itemize}
\item assume $\neg \text{marked } m$
\item hence $\text{?case}$
\item using $\text{Cons}(1)\{|i \ k| \text{ Cons}(2)$
\item by simp
\end{itemize}

moreover \begin{itemize}
\item assume $i \geq l \text{ marked } m$
\item hence $\text{?case}$
\item using $(k \geq i)$
\item by simp
\end{itemize}

moreover \begin{itemize}
\item assume $i < l \text{ marked } m$
\item hence $\text{?case}$
\item using $\text{Cons}(1)\{|i+1 \ k+1| \text{ Cons}(2)$
\item by simp
\end{itemize}

ultimately

\begin{itemize}
\item show $\text{?case}$
\item by (auto split: split-if-asm)
\end{itemize}

qed simp

lemma isPrefixPrefixToLevel-auxLowerLevel:

\begin{align*}
\text{assumes } &i \leq j \\
\text{shows } &\text{isPrefix } (\text{prefixToLevel-aux } M \cup i) (\text{prefixToLevel-aux } M \cup j) \text{ k}
\end{align*}

using assms

by (induct $M$ arbitrary: $k$) (auto simp add:isPrefix-def)

lemma isPrefixPrefixToLevelLowerLevel:

\begin{align*}
\text{assumes } &\text{level } < \text{ level}' \\
\text{shows } &\text{isPrefix } (\text{prefixToLevel } \text{ level } M) (\text{prefixToLevel } \text{ level}' M)
\end{align*}

using assms

unfolding prefixToLevel-def

using isPrefixPrefixToLevel-auxLowerLevel[of level level' M 0]

by simp

lemma prefixToLevel-auxPrefixToLevel-auxHigherLevel:

\begin{align*}
\text{assumes } &i \leq j \\
\text{shows } &\text{prefixToLevel-aux } a \cup i \ k = \text{prefixToLevel-aux } (\text{prefixToLevel-aux } a \cup j \ k) \cup i \ k
\end{align*}
using assms
by (induct a arbitrary: k) auto

lemma prefixToLevelPrefixToLevelHigherLevel:
  assumes level \leq level'
  shows prefixToLevel level M = prefixToLevel level (prefixToLevel level' M)
using assms
unfolding prefixToLevel-def
using prefixToLevel-auxPrefixToLevel-auxHigherLevel[of level level' M 0]
by simp

lemma prefixToLevelAppend-aux1:
  assumes l \geq i \and l - i < currentLevel a
  shows prefixToLevel-aux (a @ b) l i = prefixToLevel-aux a l i
using assms
proof (induct a arbitrary: i)
case (Cons a a')
{
  assume \neg marked a
  hence \?case
  using Cons[1][of i] \langle i \leq l \mid l - i < currentLevel (a # a') \rangle
  unfolding currentLevel-def
  by simp
}
moreover
{
  assume marked a l = i
  hence \?case
  by simp
}
moreover
{
  assume marked a l > i
  hence \?case
  using Cons[1][of i + 1] \langle i \leq l \mid l - i < currentLevel (a # a') \rangle
  unfolding currentLevel-def
  by simp
}
ultimately
show \?case
  using \langle i \leq l \rangle
  by auto
next
case Nil
thus \?case
lemma prefixToLevelAppend-aux2:
assumes
  \( i \leq l \) and \( \text{currentLevel}(a + i) \leq l \)
shows \( \text{prefixToLevel}(a \circ b) \cdot l \cdot i = a \circ \text{prefixToLevel}(b \circ l \cdot (i + \text{currentLevel}(a))) \)
using assms
proof (induct a arbitrary: \( i \))
  case (Cons \( a \) \( a' \))
  {
    assume \( \neg \text{marked}(a) \)
    hence ?case
      using Cons
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume \( \text{marked}(a) \cdot l = i \)
    hence ?case
      using \( \langle \text{currentLevel}(a \# a') + i \leq b \rangle \)
      unfolding currentLevel-def
      by simp
  }
  moreover
  {
    assume \( \text{marked}(a) \cdot l > i \)
    hence prefixToLevel-aux \( a' \circ b \) \( l \cdot (i + 1) = a' \circ \text{prefixToLevel-aux} \)
      \( b \circ l \cdot (i + 1 + \text{currentLevel}(a')) \)
      using Cons \( \langle \text{length} \cdot (\text{markedElements}(a')) = i + (1 + \text{length}(\text{markedElements}(a'))) \rangle \)
      unfolding currentLevel-def
      by simp
    moreover
    have \( i + 1 + \text{length}(\text{markedElements}(a')) = i + (1 + \text{length}(\text{markedElements}(a'))) \)
      by simp
    ultimately
    have ?case
      using \( \langle \text{marked}(a) \cdot \circ \cdot l > i \rangle \)
      unfolding currentLevel-def
      by simp
  }
ultimately
show ?case
using \( \langle \cdot l \geq i \rangle \)
by auto
next
  case Nil
  thus True
    unfolding currentLevel-def
    by simp
  qed

lemma prefixToLevelAppend:
  shows prefixToLevel level (a @ b) =
    (if level < currentLevel a then
      prefixToLevel level a
    else
      a @ prefixToLevel-aux b level (currentLevel a))
proof (cases level < currentLevel a)
  case True
  thus True
    unfolding prefixToLevel-def
    using prefixToLevelAppend-aux1[of 0 level a]
    by simp
next
  case False
  thus True
    unfolding prefixToLevel-def
    using prefixToLevelAppend-aux2[of 0 level a]
    by simp
  qed

lemma isProperPrefixPrefixToLevel:
  assumes level < currentLevel t
  shows ∃ s. (prefixToLevel level t) @ s = t ∧ s ≠ [] ∧ (marked (hd s))
proof –
  have isPrefix (prefixToLevel level t) t
    by (simp add:isPrefixPrefixToLevel)
  then obtain s::'a Trail
    where (prefixToLevel level t) @ s = t
    unfolding isPrefix-def
    by auto
  moreover
  have s ≠ []
  proof –
    { assume s = []
      with ((prefixToLevel level t) @ s = t)
      have prefixToLevel level t = t
        by simp
      hence currentLevel (prefixToLevel level t) ≤ level
    }
using `currentLevelPrefixToLevel` of `level t` by simp
with (`prefixToLevel level t = t`) have `currentLevel t ≤ level` by simp
with (`level < currentLevel t`) have `False` by simp

thus `?thesis` by auto
qed

moreover have `marked (hd s)` proof –
{
  assume ¬ `marked (hd s)`
  have `currentLevel (prefixToLevel level t) ≤ level` by (simp add: `currentLevelPrefixToLevel`)
  from `s ≠ []` have `s = [hd s] @ (tl s)` by simp
  with (`prefixToLevel level t) @ s = t` have `t = (prefixToLevel level t) @ [hd s] @ (tl s)` by simp
  hence `(prefixToLevel level t) = (prefixToLevel level ((prefixToLevel level t) @ [hd s] @ (tl s)))` by simp
  also have `... = (prefixToLevel-aux (hd s) level (currentLevel (prefixToLevel level t)) = (hd s) # prefixToLevel-aux (tl s) level (currentLevel (prefixToLevel level t)))` proof –
  from `(currentLevel (prefixToLevel level t) ≤ level) ¬ `marked (hd s)`
  have `prefixToLevel-aux ([hd s] @ (tl s)) level (currentLevel (prefixToLevel level t)) = (hd s) # prefixToLevel-aux (tl s) level (currentLevel (prefixToLevel level t))` by simp
  thus `?thesis` by simp
  qed
ultimately have `(prefixToLevel level t) = (prefixToLevel level t) @ (hd s) # prefixToLevel-aux (tl s) level (currentLevel (prefixToLevel level t))` by simp
hence \( \text{False} \)
  
  by \( \text{auto} \)

\}

thus \( ?\text{thesis} \)
  
  by \( \text{auto} \)

qed

ultimately

show \( ?\text{thesis} \)
  
  by \( \text{auto} \)

qed

lemma \( \text{prefixToLevelElementsElementLevel} : \)

assumes

\( e \in \text{set} (\text{elements (prefixToLevel level } M)) \)

shows

\( \text{elementLevel } e \ M \leq \text{level} \)

proof –

have \( \text{elementLevel } e \ (\text{prefixToLevel level } M) \leq \text{currentLevel (prefixToLevel level } M) \)
  
  by \( (\text{simp add: elementLevelLeqCurrentLevel}) \)

moreover

hence \( \text{currentLevel (prefixToLevel level } M) \leq \text{level} \)
  
  using \( \text{currentLevelPrefixToLevel[of level } M] \)
  
  by \( \text{simp} \)

ultimately have \( \text{elementLevel } e \ (\text{prefixToLevel level } M) \leq \text{level} \)
  
  by \( \text{simp} \)

moreover

have \( \text{isPrefix (prefixToLevel level } M) \ M \)
  
  by \( (\text{simp add: isPrefixPrefixToLevel}) \)

then obtain \( s \)

where \( \text{(prefixToLevel level } M) @ s = M \)

unfolding \( \text{isPrefix-def} \)
  
  by \( \text{auto} \)

with \( \langle e \in \text{set (elements (prefixToLevel level } M)) \rangle \)

have \( \text{elementLevel } e \ (\text{prefixToLevel level } M) = \text{elementLevel } e \ M \)
  
  using \( \text{elementLevelAppend [of } e \text{ prefixToLevel level } M \ s] \)
  
  by \( \text{simp} \)

ultimately

show \( ?\text{thesis} \)
  
  by \( \text{simp} \)

qed

lemma \( \text{elementLevelLtLevelImpliesMemberPrefixToLevel-aux} : \)

assumes

\( e \in \text{set} (\text{elements } M) \ \text{and} \)

\( \text{elementLevel } e \ M + i \leq \text{level} \ \text{and} \)

\( i \leq \text{level} \)

shows

\( e \in \text{set} (\text{elements (prefixToLevel-aux } M \ \text{level } i)) \)

118
using assms

proof (induct M arbitrary: i)
  case (Cons m M')
  thus ?case

  proof (cases e = element m)
    case True
    thus ?thesis
    using (elementLevel e (m # M') + i ≤ level)
    unfolding prefixToLevel-def
    unfolding elementLevel-def
    unfolding markedElementsTo-def
    by (simp split: split-if-asm)
  next
    case False
    with (e ∈ set (elements (m # M')))
    have e ∈ set (elements M')
    by simp

    show ?thesis
    proof (cases marked m)
      case True
      with Cons (e ≠ element m)
      have (elementLevel e M') + i + 1 ≤ level
      unfolding elementLevel-def
      unfolding markedElementsTo-def
      by (simp split: split-if-asm)
      moreover
      have elementLevel e M' ≥ 0
      by auto
      ultimately
      have i + 1 ≤ level
      by simp
      with (e ∈ set (elements M')) (elementLevel e M') + i + 1 ≤ level
      cons(1)[of i+1]
      have e ∈ set (elements (prefixToLevel-aux M' level (i + 1)))
      by simp
      with (e ≠ element m) (i + 1 ≤ level) True
      show ?thesis
      by simp
    next
    case False
    with (e ≠ element m) (elementLevel e (m # M') + i ≤ level)
    have elementLevel e M' + i ≤ level
    unfolding elementLevel-def
    unfolding markedElementsTo-def
    by (simp split: split-if-asm)
    with (e ∈ set (elements M')) have e ∈ set (elements (prefixToLevel-aux M' level i))
    using Cons

  next
by (auto split: split-if-asm)
with (e ≠ element m) False show ?thesis
by simp
qed
qed
qed simp

lemma elementLevelLtLevelImpliesMemberPrefixToLevel:
assumes e ∈ set (elements M) and elementLevel e M ≤ level
shows e ∈ set (elements (prefixToLevel level M))
using assms
using elementLevelLtLevelImpliesMemberPrefixToLevel-aux[of e M 0 level]
unfolding prefixToLevel-def
by simp

lemma literalNotInEarlierLevelsThanItsLevel:
assumes level < elementLevel e M
shows e /∈ set (elements (prefixToLevel level M))
proof−
{ assume ¬ ?thesis
  hence level ≥ elementLevel e M
    by (simp add: prefixToLevelElementsElementLevel)
  with (level < elementLevel e M)
  have False
    by simp
}
thus ?thesis
by auto
qed

lemma elementLevelPrefixElement:
assumes e ∈ set (elements (prefixToLevel level M))
shows elementLevel e (prefixToLevel level M) = elementLevel e M
using assms
proof−
  have isPrefix (prefixToLevel level M) M
    by (simp add: isPrefixPrefixToLevel)
  then obtain s where (prefixToLevel level M) ⊕ s = M
    unfolding isPrefix-def
    by auto
  with assms show ?thesis
    using elementLevelAppend[of e prefixToLevel level M s]
by auto
qed

lemma currentLevelZeroTrailEqualsItsPrefixToLevelZero:
   assumes currentLevel M = 0
   shows M = prefixToLevel 0 M
using assms
proof (induct M)
case (Cons a M')
show ?case
proof -
  from Cons have currentLevel M' = 0 and markedElements M' = [] and ¬ marked a
  unfolding currentLevel-def
  by (auto split: split-if-asm)
  thus ?thesis
  using Cons unfolding prefixToLevel-def
  by auto
qed
next
case Nil
thus ?case
  unfolding currentLevel-def
  unfolding prefixToLevel-def
  by simp
qed

3.9 Number of literals of every trail level

primrec
levelsCounter-aux :: 'a Trail ⇒ nat list ⇒ nat list
where
  levelsCounter-aux [] l = l
| levelsCounter-aux (h # t) l =
    (if (marked h) then
      levelsCounter-aux t (l @ [1])
    else
      levelsCounter-aux t (butlast l @ [Suc (last l)])
    )

definition
levelsCounter :: 'a Trail ⇒ nat list
where
levelsCounter t = levelsCounter-aux t [0]

lemma levelsCounter-aux-startIrrelevant:
\forall y. y \neq [] \rightarrow \text{levelsCounter-aux} a (x \otimes y) = (x \otimes \text{levelsCounter-aux} a y)
by (induct a) (auto simp add: butlastAppend)

\text{lemma levelsCounter-auxSuffixContinues}: \forall l. \text{levelsCounter-aux} (a \otimes b) l = \text{levelsCounter-aux} b \text{levelsCounter-aux} a l
by (induct a) auto

\text{lemma levelsCounter-auxNotEmpty}: \forall l. l \neq [] \rightarrow \text{levelsCounter-aux} a l \neq []
by (induct a) auto

\text{lemma levelsCounter-auxIncreasesFirst}: \forall m n l1 l2. \text{levelsCounter-aux} a (m \# l1) = n \# l2 \rightarrow m \leq n
\text{proof (induct a)}
\hspace{1em} \text{case Nil}
\hspace{2em} \{ 
\hspace{3em} \text{fix } m::\text{nat} \text{ and } n::\text{nat} \text{ and } l1::\text{nat list} \text{ and } l2::\text{nat list}
\hspace{3em} \text{assume levelsCounter-aux} [] (m \# l1) = n \# l2
\hspace{3em} \text{hence } m = n
\hspace{3em} \text{by simp}
\hspace{2em} \}
\hspace{2em} \text{thus } ?\text{case}
\hspace{2em} \text{by simp}
\hspace{1em} \text{next}
\hspace{2em} \text{case } (\text{Cons } a \text{ list})
\hspace{3em} \{ 
\hspace{4em} \text{fix } m::\text{nat} \text{ and } n::\text{nat} \text{ and } l1::\text{nat list} \text{ and } l2::\text{nat list}
\hspace{4em} \text{assume levelsCounter-aux} (a \# \text{list}) (m \# l1) = n \# l2
\hspace{4em} \text{have } m \leq n
\hspace{4em} \text{proof (cases marked } a)
\hspace{5em} \text{case True}
\hspace{6em} \text{with } \text{levelsCounter-aux} (a \# \text{list}) (m \# l1) = n \# l2;
\hspace{6em} \text{have levelsCounter-aux list} (m \# l1 @ [\text{Suc 0}]) = n \# l2
\hspace{6em} \text{by simp}
\hspace{6em} \text{with } \text{Cons}
\hspace{6em} \text{show } ?\text{thesis}
\hspace{6em} \text{by } \text{auto}
\hspace{5em} \text{next}
\hspace{5em} \text{case False}
\hspace{6em} \text{show } ?\text{thesis}
\hspace{6em} \text{proof (cases } l1 = [])
\hspace{7em} \text{case True}
\hspace{8em} \text{with } \neg \text{marked } a \text{ levelsCounter-aux} (a \# \text{list}) (m \# l1) = n \# l2)
\hspace{8em} \text{have levelsCounter-aux list} [\text{Suc } m] = n \# l2
\hspace{8em} \text{by simp}
\hspace{8em} \text{with } \text{Cons}
\hspace{8em} \text{have } \text{Suc } m \leq n
\hspace{4em} \}
\hspace{1em} \}
\text{122}
by auto
thus ?thesis
by simp
next
case False
with (¬ marked a) (levelsCounter-aux (a ≠ list) (m ≠ l1) = n # l2)
  have levelsCounter-aux list (m ≠ butlast l1 @ [Suc (last l1)]) = n ≠ l2
    by simp
    with Cons
    show ?thesis
    by auto
qed
qed

lemma levelsCounterPrefix:
asumes (isPrefix p a)
shows ? rest. rest ≠ [] ∧ levelsCounter a = butlast (levelsCounter p) @ rest ∧ last (levelsCounter p) ≤ hd rest
proof-
from asms
obtain s :: 'a Trail where p @ s = a
  unfolding isPrefix-def
  by auto
from p @ s = a' have levelsCounter a = levelsCounter (p @ s)
  by simp
show ?thesis
proof (cases s = [])
case True
have (levelsCounter a) = (butlast (levelsCounter p)) @ [last (levelsCounter p)] ∧
  (last (levelsCounter p)) ≤ hd [last (levelsCounter p)]
  using (p @ s = a) (s = [])
  unfolding levelsCounter-def
  using levelsCounter-auxNotEmpty[of p]
  by auto
thus ?thesis
  by auto
next
case False
show ?thesis
proof (cases marked (hd s))
case True
from p @ s = a' have levelsCounter a = levelsCounter (p @ s)
by simp
also
have ... = levelsCounter-aux s (levelsCounter-aux p [0])
unfolding levelsCounter-def
by (simp add: levelsCounter-auxSuffixContinues)
also
have ... = levelsCounter-aux (tl s) ((levelsCounter-aux p [0]) @ [1])
proof-
  from ⟨s ≠ []⟩ have s = hd s # tl s
  by simp
  then have levelsCounter-aux s (levelsCounter-aux p [0]) =
levelsCounter-aux (hd s # tl s) (levelsCounter-aux p [0])
  by simp
  with ⟨marked (hd s)⟩ show ?thesis
  by simp
qed
also
have ... = levelsCounter-aux p [0] @ (levelsCounter-aux (tl s) [1])
  by (simp add: levelsCounter-auxStartIrrelevant)
finally
have levelsCounter a = levelsCounter p @ (levelsCounter-aux (tl s) [1])
  unfolding levelsCounter-def
  by simp
  hence (levelsCounter a) = (butlast (levelsCounter p)) @ ([last (levelsCounter p)] @ (levelsCounter-aux (tl s) [1])) ∧
(butlast (levelsCounter p)) <= hd ([last (levelsCounter p)] @ (levelsCounter-aux (tl s) [1]))
  unfolding levelsCounter-def
  using levelsCounter-auxNotEmpty[of p]
  by auto
  thus ?thesis
  by auto
next
case False
from (p @ s = a) have levelsCounter a = levelsCounter (p @ s)
  by simp
also
have ... = levelsCounter-aux s (levelsCounter-aux p [0])
unfolding levelsCounter-def
by (simp add: levelsCounter-auxSuffixContinues)
also
have ... = levelsCounter-aux (tl s) ((butlast (levelsCounter-aux p [0])) @ [Suc (last (levelsCounter-aux p [0]))])
proof-
from ⟨s ≠ []⟩ have s = hd s # tl s
  by simp
then have levelsCounter-aux s (levelsCounter-aux p [0]) =
levelsCounter-aux (hd s ≠ tl s) (levelsCounter-aux p [0])
  by simp
with ¬marked (hd s): show ?thesis
  by simp
qed
also have...
by simp
qed
finally have levelsCounter a = butlast (levelsCounter-aux p [0]) @
(levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux p [0]))])
unfolding levelsCounter-def
by simp
moreover have hd (levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux p [0]))]) >=
(Suc (last (levelsCounter-aux p [0])))
proof
  have (levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux p [0]))]) ≠ []
  using levelsCounter-auxNotEmpty[of tl s]
  by simp
  then obtain h t where (levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux p [0]))]) = h # t
    using neq-Nil-cong[of (levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux p [0]))])] by auto
  hence h ≥ Suc (last (levelsCounter-aux p [0]))
    using levelsCounter-auxIncreasesFirst[of tl s] by auto
  with (levelsCounter-aux (tl s) [Suc (last (levelsCounter-aux p [0]))]) = h # t
    show ?thesis
    by simp
qed
ultimately have levelsCounter a = butlast (levelsCounter-aux p) @ (levelsCounter-aux
(tl s) [Suc (last (levelsCounter-aux p [0]))]) ∧
last (levelsCounter p) ≤ hd (levelsCounter-aux (tl s) [Suc (last
(levelsCounter-aux p [0]))])
unfolding levelsCounter-def
by simp
thus ?thesis
  using levelsCounter-auxNotEmpty[of tl s] by auto
qed
qed
lemma levelsCounterPrefixToLevel:
  assumes p = prefixToLevel level a level ≥ 0 level < currentLevel a
  shows ? rest . rest ≠ [] ∧ (levelsCounter a) = (levelsCounter p) @ rest
proof -
  from assms
  obtain s :: 'a Trail where p @ s = a s ≠ [] marked (hd s)
    using isProperPrefixPrefixToLevel[of level a]
    by auto
  from (p @ s = a) have levelsCounter a = levelsCounter (p @ s)
    by simp
  also
  have ... = levelsCounter-aux s (levelsCounter-aux p [0])
    unfolding levelsCounter-def
    by (simp add: levelsCounter-auxSuffixContinues)
  also
  have ... = levelsCounter-aux (tl s) ((levelsCounter-aux p [0]) @ [1])
    proof -
    from (s ≠ []) have s = hd s # tl s
      by simp
    then have levelsCounter-aux s (levelsCounter-aux p [0]) = levelsCounter-aux
      (hd s # tl s) (levelsCounter-aux p [0])
      by simp
    with (marked (hd s)) show ?thesis
      by simp
    qed
  also
  have ... = levelsCounter-aux (tl s) [1]
    (levelsCounter-aux (tl s) [1])
    unfolding levelsCounter-def
    by simp
  moreover
  have levelsCounter-aux (tl s) [1] ≠ []
    by (simp add: levelsCounter-auxNotEmpty)
  ultimately
  show ?thesis
    by simp
qed

3.10 Prefix before last marked element

primrec
prefixBeforeLastMarked :: 'a Trail ⇒ 'a Trail
where
  prefixBeforeLastMarked [] = []
prefixBeforeLastMarked (h#t) = (if (marked h) ∧ (markedElements t) = [] then [] else (h#(prefixBeforeLastMarked t)))

**lemma** prefixBeforeLastMarkedIsPrefixBeforeLastLevel:

- **assumes** markedElements M ≠ []
- **shows** prefixBeforeLastMarked M = prefixToLevel ((currentLevel M) - 1) M

**using** assms

**proof (induct M)**

- **case** Nil
  - **thus** ?case
    - **by** simp

- **next**
  - **case** (Cons a M')
    - **thus** ?case
      **proof (cases marked a)**
      - **case** True
        - **hence** currentLevel (a # M') ≥ 1
          - **unfolding** currentLevel-def
            - **by** simp
        - **with** True Cons show ?thesis
          - **using** prefixToLevel-auxIncreaseAuxiliaryCounter[of 0 1 M' currentLevel M' - 1]
            - **unfolding** prefixToLevel-def
            - **unfolding** currentLevel-def
            - **by** auto
      - **next**
        - **case** False
          - **with** Cons show ?thesis
            - **unfolding** prefixToLevel-def
            - **unfolding** currentLevel-def
            - **by** auto
      - **qed**
    - **qed**

**lemma** isPrefixPrefixBeforeLastMarked:

- **shows** isPrefix (prefixBeforeLastMarked M) M

**unfolding** isPrefix-def

**by** (induct M) auto

**lemma** lastMarkedNotInPrefixBeforeLastMarked:

- **assumes** uniq (elements M) and markedElements M ≠ []
- **shows** ¬ (lastMarked M) ∈ set (elements (prefixBeforeLastMarked M))

**using** assms

**unfolding** lastMarked-def

**by** (induct M) (auto split: split-if-asm simp add: markedElementsAreElements)
lemma uniqImpliesPrefixBeforeLastMarkedIsPrefixBeforeLastMarked:
  assumes markedElements M ≠ [] and (lastMarked M) ∉ set (elements M)
  shows prefixBeforeLastMarked M = prefixBeforeElement (lastMarked M) M
  using assms unfolding lastMarked-def
proof (induct M)
  case Nil
  thus ?case by auto
next
  case (Cons a M')
  show ?case proof (cases marked a ∧ (markedElements M') = [])
    case True
    thus ?thesis unfolding lastMarked-def by auto
  next
    case False hence last (markedElements (a # M')) = last (markedElements M')
    by auto
    thus ?thesis using Cons by (auto split: split-if-asm simp add: markedElementsAreElements)
  qed
qed

lemma markedElementsAreElementsBeforeLastDecisionAndLastDecision:
  assumes markedElements M ≠ []
  shows (markedElements M) = (markedElements (prefixBeforeLastMarked M)) @ [lastMarked M]
  using assms unfolding lastMarked-def
by (induct M) (auto split: split-if-asm)

end

4 Verification of DPLL based SAT solvers.

theory SatSolverVerification
imports CNF Trail
begin
  This theory contains a number of lemmas used in verification
of different SAT solvers. Although this file does not contain any theorems significant on their own, it is an essential part of the SAT solver correctness proof because it contains most of the technical details used in the proofs that follow. These lemmas serve as a basis for partial correctness proof for pseudo-code implementation of modern SAT solvers described in [2], in terms of Hoare logic.

4.1 Literal Trail

LiteralTrail is a Trail consisting of literals, where decision literals are marked.

```plaintext
type-synonym LiteralTrail = Literal Trail

abbreviation isDecision :: (a × bool) ⇒ bool
  where isDecision l == marked l

abbreviation lastDecision :: LiteralTrail ⇒ Literal
  where lastDecision M == Trail.lastMarked M

abbreviation decisions :: LiteralTrail ⇒ Literal list
  where decisions M == Trail.markedElements M

abbreviation decisionsTo :: Literal ⇒ LiteralTrail ⇒ Literal list
  where decisionsTo M l == Trail.markedElementsTo M l

abbreviation prefixBeforeLastDecision :: LiteralTrail ⇒ LiteralTrail
  where prefixBeforeLastDecision M == Trail.prefixBeforeLastMarked M
```

4.2 Invariants

In this section a number of conditions will be formulated and it will be shown that these conditions are invariant after applying different DPLL-based transition rules.

```plaintext
definition InvariantConsistent (M::LiteralTrail) == consistent (elements M)

definition InvariantUniq (M::LiteralTrail) == uniq (elements M)

definition InvariantImpliedLiterals (F::Formula) (M::LiteralTrail) == ∀ l. l el elements M ⤙ formulaEntailsLiteral (F @ val2form (decisionsTo l M)) l
```
definition
InvariantEquivalent \((F_0::Formula) (F::Formula) == equivalentFormulae F_0 F\)

definition
InvariantVarsM \((M::LiteralTrail) (F_0::Formula) (Vbl::Variable set) == vars (elements M) \subseteq vars F_0 \cup Vbl\)

definition
InvariantVarsF \((F::Formula) (F_0::Formula) (Vbl::Variable set) == vars F \subseteq vars F_0 \cup Vbl\)

The following invariants are used in conflict analysis.

definition
InvariantCFalse \((\text{conflictFlag}::\text{bool}) (M::\text{LiteralTrail}) (C::\text{Clause}) == \text{conflictFlag} \rightarrow \text{clauseFalse} C (\text{elements} M)\)

definition
InvariantCEntailed \((\text{conflictFlag}::\text{bool}) (F::\text{Formula}) (C::\text{Clause}) == \text{conflictFlag} \rightarrow \text{formulaEntailsClause} F C\)

definition
InvariantReasonClauses \((F::\text{Formula}) (M::\text{LiteralTrail}) == \forall \text{ literal}. \text{ literal el (elements} M) \land \neg \text{ literal el decisions} M \rightarrow (\exists \text{ clause}. \text{formulaEntailsClause} F \text{ clause} \land \text{isReason clause literal} (\text{elements} M))\)

4.2.1 Auxiliary lemmas

This section contains some auxiliary lemmas that additionally characterize some of invariants that have been defined.

Lemmas about InvariantImpliedLiterals.

lemma InvariantImpliedLiteralsWeakerVariant:
fixes \(M::\text{LiteralTrail} \text{ and } F::\text{Formula}\)
assumes \(\forall \text{ l}. \text{ l el elements} M \rightarrow \text{formulaEntailsLiteral} (F @ \text{val2form (decisionsTo l} M)) \text{ l}\)
shows \(\forall \text{ l}. \text{ l el elements} M \rightarrow \text{formulaEntailsLiteral} (F @ \text{val2form (decisions} M)) \text{ l}\)
proof –
{
fix \(l::\text{Literal}\)
assume \(l \text{ el elements} M\)
with \(\text{assms}\)
have \(\text{formulaEntailsLiteral} (F @ \text{val2form (decisionsTo l} M)) \text{ l}\)
by simp
have \(\text{isPrefix (decisionsTo l} M) \text{ (decisions} M)\)
by (simp add: \text{markedElementsToArePrefixOfMarkedElements})
then obtain \(s::\text{Valuation}\)
where \((\text{decisionsTo } l\ M) \circ s = (\text{decisions } M)\)

using isPrefix-def [of decisionsTo \(l\ M\) decisions \(M\)]

by auto

hence \((\text{decisions } M) = (\text{decisionsTo } l\ M) \circ s\)

by (rule sym)

with \((\text{formulaEntailsLiteral} (F @ \text{val2form} (\text{decisionsTo } l\ M))) \land \text{have formulaEntailsLiteral} (F @ \text{val2form} (\text{decisions } M)) \land \text{using formulaEntailsLiteralAppend} [of F @ \text{val2form} (\text{decisionsTo } l\ M) \land \text{val2form } s]\)

by (auto simp add: formulaEntailsLiteralAppend val2formAppend)

thus ?thesis

by simp

qed

lemma InvariantImpliedLiteralsAndElementsEntailLiteralThenDecision-sEntailLiteral:

\begin{align*}
\text{fixes } M &:: \text{LiteralTrail and } F ::= \text{Formula and } \text{literal }::= \text{Literal} \\
\text{assumes } &\text{InvariantImpliedLiterals } F \text{ M and formulaEntailsLiteral} (F @ (\text{val2form} (\text{elements } M))) \land \text{literal} \\
\text{shows } &\text{formulaEntailsLiteral} (F @ \text{val2form} (\text{decisions } M)) \land \text{literal} \\
\text{proof} &\quad - \\
&\quad \{ \\
&\quad \text{fix valuation :: Valuation} \\
&\quad \text{assume model valuation} (F @ \text{val2form} (\text{decisions } M)) \\
&\quad \text{hence formulaTrue } F \text{ valuation and formulaTrue} (\text{val2form} (\text{decisions } M)) \land \text{valuation and consistent valuation} \\
&\quad \text{by (auto simp add: formulaTrueAppend)} \\
&\quad \} \\
&\quad \{ \\
&\quad \text{fix } l :: \text{Literal} \\
&\quad \text{assume } l \text{ el} (\text{elements } M) \\
&\quad \text{from \(\text{InvariantImpliedLiterals } F \text{ M}\)} \\
&\quad \text{have } \forall \ l. \ l \text{ el} (\text{elements } M) \rightarrow \text{formulaEntailsLiteral} (F @ \text{val2form} (\text{decisions } M)) \land \text{l} \\
&\quad \text{by (simp add: InvariantImpliedLiteralsWeakerVariant InvariantImpliedLiterals-def)} \\
&\quad \text{with } l \text{ el} (\text{elements } M) \\
&\quad \text{have formulaEntailsLiteral} (F @ \text{val2form} (\text{decisions } M)) \land \text{l} \\
&\quad \text{by simp} \\
&\quad \text{with (model valuation} (F @ \text{val2form} (\text{decisions } M)) \\
&\quad \text{have literalTrue } l \text{ valuation} \\
&\quad \text{by (simp add: formulaEntailsLiteral-def)} \\
&\quad \} \\
&\quad \text{hence formulaTrue} (\text{val2form} (\text{elements } M)) \text{ valuation} \\
&\quad \text{by (simp add: val2formFormulaTrue)} \\
&\quad \text{with \(\text{formulaTrue } F \text{ valuation} \land \text{consistent valuation} \)} \\
&\quad \text{have model valuation} (F @ (\text{val2form} (\text{elements } M))) \\
&\quad \text{by (auto simp add: formulaTrueAppend)} \\
&\quad \text{with \(\text{formulaEntailsLiteral} (F @ (\text{val2form} (\text{elements } M))) \land \text{literal}\)} \\
&\quad \text{have literalTrue } \text{ literal valuation}
\end{align*}

131
by (simp add: formulaEntailsLiteral-def)

} 
thus ?thesis by (simp add: formulaEntailsLiteral-def)
qed

lemma InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable:
fixes M :: LiteralTrail and F :: Formula
assumes InvariantImpliedLiterals F M and formulaFalse F (elements M)
sows \neg \text{satisfiable} (F @ val2form (decisions M))

proof –
from \langle formulaFalse F (elements M) \rangle
have formulaFalse (F @ val2form (decisions M)) (elements M)
  by (simp add: formulaFalseAppend)
moreover
from \langle InvariantImpliedLiterals F M \rangle
have formulaEntailsValuation (F @ val2form (decisions M)) (elements M)
  unfolding formulaEntailsValuation-def
  unfolding InvariantImpliedLiterals-def
  using InvariantImpliedLiteralsWeakerVariant[of M F]
  by simp
ultimately
show ?thesis
  using formulaFalseInEntailedValuationIs Unsatisfiable [of F @ val2form (decisions M) elements M]
  by simp
qed

lemma InvariantImpliedLiteralsHoldsForPrefix:
fixes M :: LiteralTrail and prefix :: LiteralTrail and F :: Formula
assumes InvariantImpliedLiterals F M and isPrefix prefix M
shows InvariantImpliedLiterals F prefix

proof –
{
  fix l :: Literal
  assume *: l el elements prefix

  from * (isPrefix prefix M)
  have l el elements M
    unfolding isPrefix-def
    by auto

  from * and (isPrefix prefix M)
  have decisionsTo l prefix = decisionsTo l M
    using markedElementsToPrefixElement [of prefix M l]
    by simp

qed
from ⟨InvariantImpliedLiterals F M⟩ and ⟨l el elements M⟩
have formulaEntailsLiteral (F @ val2form (decisionsTo l M)) l
  by (simp add: InvariantImpliedLiterals-def)
with ⟨decisionsTo l prefix = decisionsTo l M⟩
have formulaEntailsLiteral (F @ val2form (decisionsTo l prefix)) l
  by simp
thus ?thesis
by (auto simp add: InvariantImpliedLiterals-def)
qed

Lemmas about InvariantReasonClauses.

lemma InvariantReasonClausesHoldsForPrefix:
  fixes F::Formula and M::LiteralTrail and p::LiteralTrail
  assumes InvariantReasonClauses F M and InvariantUniq M and isPrefix p M
  shows InvariantReasonClauses F p
proof−
  from ⟨InvariantReasonClauses F M⟩
  have *: ∀ literal. literal el elements M ∧ ¬ literal el decisions M
    → (∃ clause. formulaEntailsClause F clause ∧ isReason clause literal (elements M))
    unfolding InvariantReasonClauses-def
    by simp
  from ⟨InvariantUniq M⟩
  have uniq (elements M)
    unfolding InvariantUniq-def
    by simp
  { fix literal::Literal
    assume literal el elements p and ¬ literal el decisions p
    from ⟨isPrefix p M⟩ ⟨literal el (elements p)⟩
    have literal el (elements M)
      by (auto simp add: isPrefix-def)
    moreover
    from ⟨isPrefix p M⟩ ⟨literal el (elements p) ⟷ literal el (decisions p)⟩ ⟨¬ literal el (decisions p)⟩ ⟨uniq (elements M)⟩
    have ¬ literal el decisions M
      using markedElementsTrailMemPrefixAreMarkedElementsPrefix [of M p literal]
      by auto
    ultimately
    obtain clause::Clause where
      formulaEntailsClause F clause isReason clause literal (elements M)
    using *
    by auto
with (\text{literal el elements p} \vdash \neg \text{literal el decisions p} \vdash \text{isPrefix p M})

have isReason clause literal (elements p)
using isReasonHoldsInPrefix[of literal elements p elements M clause]
by (simp add:isPrefixElements)
with (\text{formulaEntailsClause F clause})
have \exists clause. \text{formulaEntailsClause F clause} \land \text{isReason clause literal (elements p)}
by auto

\}
thus \thesis
unfolding InvariantReasonClauses-def
by auto
qed

\textbf{lemma} InvariantReasonClausesHoldsForPrefixElements:
fixes F::Formula and M::LiteralTrail and p::LiteralTrail
assumes InvariantReasonClauses F p and
isPrefix p M and
\text{literal el (elements p)} \land \neg \text{literal el decisions M}
shows \exists clause. \text{formulaEntailsClause F clause} \land \text{isReason clause literal (elements M)}
proof =
from (\text{isPrefix p M}) \vdash (\neg \text{literal el (decisions M)})
have \neg \text{literal el (decisions p)}
using markedElementsPrefixAreMarkedElementsTrail[of p M literal]
by auto

from (\text{InvariantReasonClauses F p (literal el (elements p))}; (\neg \text{literal el (decisions p)}); \neg \text{literal el (decisions p)}) obtain clause :: Clause
where \text{formulaEntailsClause F clause isReason clause literal (elements p)}
unfolding InvariantReasonClauses-def
by auto
with (\text{isPrefix p M})
have isReason clause literal (elements M)
using isReasonAppend[of clause literal elements p]
by (auto simp add: isPrefix-def)
with (\text{formulaEntailsClause F clause})
show \thesis
by auto
qed

\textbf{4.2.2 Transition rules preserve invariants}

In this section it will be proved that the different DPLL-based transition rules preserves given invariants. Rules are implicitly
given in their most general form. Explicit definition of transition rules will be done in theories that describe specific solvers.

*Decide* transition rule.

**lemma** InvariantUniqAfterDecide:

- **fixes** \( M :: \text{LiteralTrail} \) and \( \text{literal} :: \text{Literal} \) and \( M' :: \text{LiteralTrail} \)
- **assumes** InvariantUniq \( M \) and \( \forall \text{ literal} \not\in \text{vars (elements } M) \) and \( M' = M @ [(\text{literal}, \text{True})] \)
- **shows** InvariantUniq \( M' \)

**proof** –
- from \( \langle \text{InvariantUniq } M \rangle \)
- have uniq (elements \( M \))
  using unfolding InvariantUniq-def.
- 
  \[
  \begin{align*}
  &\text{assume } \neg \text{ uniq (elements } M') \\
  &\text{with } \langle \text{uniq (elements } M) \rangle \langle \text{M' = M @ [(\text{literal}, \text{True})]} \rangle \\
  &\text{have literal } \in \text{ (elements } M) \\
  &\text{using uniqButlastNotUniqListImpliesLastMemButlast [of elements } M'] \\
  &\text{by auto} \\
  &\text{hence var literal } \in \text{ vars (elements } M) \\
  &\text{using valuationContainsItsLiteralsVariable [of literal elements } M] \\
  &\text{by simp} \\
  &\text{with } \langle \text{var literal} \not\in \text{ vars (elements } M) \rangle \\
  &\text{have False} \\
  &\text{by simp} \\
  \end{align*}
  \]
- thus \( \approx \text{thesis} \)
  unfolding InvariantUniq-def
  by auto

**qed**

**lemma** InvariantImpliedLiteralsAfterDecide:

- **fixes** \( F :: \text{Formula} \) and \( M :: \text{LiteralTrail} \) and \( \text{literal} :: \text{Literal} \) and \( M' :: \text{LiteralTrail} \)
- **assumes** InvariantImpliedLiterals \( F \) \( M \) and \( \forall \text{ literal} \not\in \text{vars (elements } M) \) and \( M' = M @ [(\text{literal}, \text{True})] \)
- **shows** InvariantImpliedLiterals \( F \) \( M' \)

**proof** –
- fix \( l :: \text{Literal} \)
- assume \( l \in \text{elements } M' \)
- have formulaEntailsLiteral \( (F @ \text{val2form (decisionsTo } l \text{ M'}) \) \( l \)
- proof (cases \( l \in \text{elements } M\)
  - case True
    - with \( M' = M @ [(\text{literal}, \text{True})] \)
have \( \text{decisionsTo } l\ M' = \text{decisionsTo } l\ M \)
  by (simp add: markedElementsToAppend)
with (InvariantImpliedLiterals \( F \ M\); \( l\ \text{el}\ elements\ M\))
show \(?\text{thesis}\)
  by (simp add: InvariantImpliedLiterals-def)
next
  case False
  with \( \langle l\ \text{el}\ elements\ M'\rangle\ \text{and}\ \langle M' = M @ [(\text{literal}, \text{True})]\rangle\)
  have \( l = \text{literal}\)
    by (auto split: split-if_asm)
  have \( \text{clauseEntailsLiteral} \ [\text{literal}]\ \text{literal}\)
    by (simp add: clauseEntailsLiteral-def)
  moreover
  have \( \langle [\text{literal}]\ \text{el}\ (F @ \text{val2form}\ (\text{decisions} \ M) @ [[\text{literal}]])\rangle\)
    by simp
  moreover
  \{ 
  have \( \text{isDecision} \ (\text{last} (M @ [(\text{literal}, \text{True})]))\)
    by simp
  moreover
  from \( \langle \text{var}\ \text{literal} \notin \text{vars} (\text{elements} \ M)\rangle\)
  have \( \neg \text{literal}\ \text{el}\ (\text{elements} \ M)\)
    using valuationContainsItsLiteralsVariable[of \( \text{literal}\ \text{elements} \ M\)]
    by auto
  ultimately
  have \( \text{decisionsTo } \text{literal} (M @ [(\text{literal}, \text{True})]) = ((\text{decisions } M) @ [\text{literal}])\)
    using lastTrailElementMarkedImpliesMarkedElementsTo-LastElementAreAllMarkedElements[of \( M @ [(\text{literal}, \text{True})]\)]
    by (simp add: markedElementsAppend)
  \}
  ultimately
  show \(?\text{thesis}\)
    using \( \langle M' = M @ [(\text{literal}, \text{True})];\ l = \text{literal}\rangle\)
    \( \text{clauseEntailsLiteralThenFormulaEntailsLiteral} \ [\text{literal}]\ F\)
    @ \text{val2form}\ (\text{decisions} \ M) @ [[\text{literal}]\ \text{literal}]\)
    by (simp add: val2formAppend)
  qed
\}
thus \(?\text{thesis}\)
  by (simp add: InvariantImpliedLiterals-def)
qed

lemma InvariantVarsMAfterDecide:
  fixes \( F :: \text{Formula} \) and \( F0 :: \text{Formula} \) and \( M :: \text{LiteralTrail} \) and \text{literal} :: \( \text{Literal} \) and \( M' :: \text{LiteralTrail} \)
  assumes InvariantVarsM \( M\ F0\ \text{Vbl} \) and \text{var}\ \text{literal} \in \text{Vbl}\ and
  \text{value}
\[ M' = M \circ \{(\text{literal}, \text{True})\} \]

shows \( \text{InvariantVars}(M' F0 \ Vbl) \)

proof

from \( \langle \text{InvariantVars}(M F0 \ Vbl) \rangle \) have \( \text{vars}(\text{elements}(M)) \subseteq \text{vars}(F0) \cup \text{Vbl} \)

by (simp only:\text{InvariantVarsM-def})

from \( \langle M' = M \circ \{(\text{literal}, \text{True})\}\rangle \) have \( \text{vars}(\text{elements}(M')) = \text{vars}(\text{elements}(M \circ \{(\text{literal}, \text{True})\})) \)

by simp

also have \( \ldots = \text{vars}(\text{elements}(M \circ \{\text{literal}\})) \)

by simp

also have \( \ldots = \text{vars}(\text{elements}(M)) \cup \text{vars}[\text{literal}] \)

using \text{varsAppendClauses}[\text{of elements}(M \circ \{\text{literal}\})]

by simp

finally

show \( ?\text{thesis} \)

using \( \langle \text{vars}(\text{elements}(M)) \subseteq (\text{vars}(F0) \cup \text{Vbl}) \rangle \langle \text{var literal} \in \text{Vbl} \rangle \)

unfolding \text{InvariantVarsM-def}

by auto

qed

lemma \text{InvariantConsistentAfterDecide}:

\[ \text{fixes } M :: \text{LiteralTrail and literal :: \text{Literal and } M' :: \text{LiteralTrail}} \]

\[ \text{assumes } \text{InvariantConsistent}(M) \text{ and} \]

\[ \text{var literal } \notin \text{vars}(\text{elements}(M)) \text{ and} \]

\[ M' = M \circ \{(\text{literal}, \text{True})\} \text{ and} \]

\[ \text{shows } \text{InvariantConsistent}(M') \]

proof

from \( \langle \text{InvariantConsistent}(M) \rangle \) have \( \text{consistent}(\text{elements}(M)) \)

unfolding \text{InvariantConsistent-def}

\{

assume \( \text{inconsistent}(\text{elements}(M')) \)

with \( \langle M' = M \circ \{(\text{literal}, \text{True})\}\rangle \)

have \( \text{inconsistent}(\text{elements}(M)) \lor \text{inconsistent}[\text{literal}] \lor (\exists \ l). \text{litFalse l}[\text{literal}] \land \text{litTrue l}[\text{elements}(M)] \land \text{litFalse l}[\text{literal}] \)

using \text{inconsistentAppend}[\text{of elements}(M \circ \{\text{literal}\})]

by simp

with \( \langle \text{consistent}(\text{elements}(M)) \rangle \) obtain \( l :: \text{Literal} \)

where \( \text{litTrue l}[\text{elements}(M)] \land \text{litFalse l}[\text{literal}] \)

by auto

hence \( \text{opposite}(l) = \text{literal} \)

by auto

hence \( \text{var literal} = \text{var l} \)

by auto

with \( \langle \text{litTrue l}[\text{elements}(M)] \rangle \)

have \( \text{var l} \in \text{vars}(\text{elements}(M)) \)

using \text{valuationContainsItsLiteralsVariable}[\text{of l elements}(M)]
by simp
with \( \text{var literal} = \text{var} \land \text{var literal} \notin \text{vars (elements M)} \)
have False
  by simp
}
thus ?thesis
unfolding InvariantConsistent-def
by auto
qed

lemma InvariantReasonClausesAfterDecide:
  fixes \( F :: \text{Formula} \) and \( M :: \text{LiteralTrail} \) and \( M' :: \text{LiteralTrail} \)
  assumes InvariantReasonClauses \( F \) \( M \) and InvariantUniq \( M \) and
  \( \text{M}' = \text{M} @ \langle \langle \text{literals} \rangle \rangle \text{True} \)
  shows InvariantReasonClauses \( F \) \( M' \)
proof –
{  
  fix literal' :: \text{Literal}
  assume literal' \text{ el elements } \text{M'} and \neg \text{literal' el decisions } \text{M'}

  have \( \exists \text{ clause. \text{formulaEntailsClause} F \text{ clause} \land \text{isReason clause literal'} (\text{elements M}') } \)
  proof (cases literal' \text{ el elements M})
    case True
    with assms \( \neg \text{literal' el decisions M'} \) obtain clause::\text{Clause}
    where \text{formulaEntailsClause} \( F \text{ clause} \land \text{isReason clause literal'} (\text{elements M}') \)
    using InvariantReasonClausesHoldsForPrefixElements [of F M \( M' \text{ literal'} ]
    by (auto simp add:isPrefix-def)
  thus ?thesis
  by auto
next
  case False
  with \( M' = M @ \langle \langle \text{literals} \rangle \rangle \text{True} \rangle \langle \text{literals} \rangle \text{ el elements } \text{M'}
  have literal' = literal'
    by (simp split: split-if-asm)
  with \( M' = M @ \langle \langle \text{literals} \rangle \rangle \text{True} \rangle \langle \text{literals} \rangle \text{ el decisions } \text{M'}
  have \( \neg \text{literal' el decisions M'} \)
  using markedElementIsMarkedTrue[of literal M']
  by simp
  with \( \neg \text{literal' el decisions M'} \)
  have False
  by simp
  thus ?thesis
  by simp
  qed
}
thus ?thesis

138
unfolding InvariantReasonClauses-def by auto qed

lemma InvariantCFalseAfterDecide:
fixes conflictFlag :: bool and M :: LiteralTrail and C :: Clause
assumes InvariantCFalse conflictFlag M C and M' = M @ [(literal, True)]
shows InvariantCFalse conflictFlag M' C
unfolding InvariantCFalse-def
proof
assume conflictFlag
show clauseFalse C (elements M')
proof
from ⟨InvariantCFalse conflictFlag M C⟩
have conflictFlag —→ clauseFalse C (elements M)
  unfolding InvariantCFalse-def
  .
  with ⟨conflictFlag⟩
  have clauseFalse C (elements M)
    by simp
  with ⟨M' = M @ [(literal, True)]⟩
  show ?thesis
    by (simp add: clauseFalseAppendValuation)
qed
qed

UnitPropagate transition rule.

lemma InvariantImpliedLiteralsHoldsForUnitLiteral:
fixes M :: LiteralTrail and F :: Formula and uClause :: Clause and uLiteral :: Literal
assumes InvariantImpliedLiterals F M and
formulaEntailsClause F uClause and isUnitClause uClause uLiteral (elements M) and
M' = M @ [(uLiteral, False)]
shows formulaEntailsLiteral (F @ val2form (decisionsTo uLiteral M')) uLiteral
proof
  have decisionsTo uLiteral M' = decisions M
  proof
    from ⟨isUnitClause uClause uLiteral (elements M)⟩
    have ¬ uLiteral el (elements M)
      by (simp add: isUnitClause-def)
    with ⟨M' = M @ [(uLiteral, False)]⟩
    show ?thesis
      using markedElementsToAppend[of uLiteral M [(uLiteral, False)]]
      unfolding markedElementsTo-def
      by simp
  qed
qed

139
moreover
from ⟨formulaEntailsClause F uClause⟩ ⟨isUnitClause uClause uLiteral (elements M)⟩

have ⟨formulaEntailsLiteral F val2form (elements M)⟩ uLiteral
using ⟨unitLiteralIsEntailed [of uClause uLiteral elements M F]⟩
by simp
with ⟨InvariantImpliedLiterals F M⟩

have ⟨formulaEntailsLiteral F val2form (decisions M)⟩ uLiteral
by (simp add: InvariantImpliedLiteralsAndElementsEntailLiteralThenDecisionsEntailLiteral)

ultimately
show ?thesis
by simp

qed

lemma InvariantImpliedLiteralsAfterUnitPropagate:
fixes M :: LiteralTrail and F :: Formula and uClause :: Clause and uLiteral :: Literal
assumes InvariantImpliedLiterals F M and
formulaEntailsClause F uClause and isUnitClause uClause uLiteral (elements M) and
M' = M @ [(uLiteral, False)]
shows InvariantImpliedLiterals F M'
proof -
{ 
  fix l :: Literal
  assume l el (elements M')
  have ⟨formulaEntailsLiteral F val2form (decisionsTo l M')⟩ l
  proof (cases l el elements M)
    case True
    with ⟨InvariantImpliedLiterals F M⟩
    have ⟨formulaEntailsLiteral F val2form (decisionsTo l M')⟩ l
    by (simp add: InvariantImpliedLiterals-def)
  moreover
  from ⟨M' = M @ [(uLiteral, False)]⟩
  have ⟨isPrefix M M'⟩
  by (simp add: isPrefix-def)
  with True
  have ⟨decisionsTo l M' = decisionsTo l M⟩
  by (simp add: markedElementsToPrefixElement)
  ultimately
  show ?thesis
  by simp
  next
  case False
  with ⟨l el (elements M')⟩ ⟨M' = M @ [(uLiteral, False)]⟩
  have l = uLiteral
  by (auto split: split-if-asm)
  moreover

from assms
have formulaEntailsLiteral $F \odot \text{val2form} \ (\text{decisionsTo} \ u\text{Literal} M') \ u\text{Literal}$
  using InvariantImpliedLiteralsHoldsForUnitLiteral \ [of \ F \ M u\text{Clause} u\text{Literal} M']
  by simp
ultimately
show \?thesis
  by simp
qed

thus \?thesis
  by (simp add: InvariantImpliedLiterals-def)
qed

lemma InvariantVarsMAfterUnitPropagate:
  fixes \ F :: \ Formula \ and \ F0 :: \ Formula \ and \ M :: \ Literal\Trail \ and
  u\text{Clause} :: \ \text{Clause} \ and \ u\text{Literal} :: \ \text{Literal} \ and \ M' :: \ \text{Literal}\Trail
  assumes InvariantVarsM M F0 Vbl \ and
  var u\text{Literal} \in \ vars \ F0 \cup \ Vbl \ and
  M' = M @ [(u\text{Literal}, \text{False})]
  shows InvariantVarsM M' F0 Vbl
proof -
  from \ (InvariantVarsM M F0 Vbl)
  have \ vars \ (\text{elements} \ M) \subseteq \ vars \ F0 \cup \ Vbl
    unfolding InvariantVarsM-def
  .
  thus \?thesis
    unfolding InvariantVarsM-def
    using \ (\text{var} \ u\text{Literal} \in \ vars \ F0 \cup \ Vbl)
    using \ (M' = M @ [(u\text{Literal}, \text{False})])
    varsAppendClauses \ [\text{of} \ \text{elements} \ M \ [u\text{Literal}]]
    by auto
qed

lemma InvariantConsistentAfterUnitPropagate:
  fixes \ M :: \ \text{Literal}\Trail \ and \ F :: \ \text{Formula} \ and \ M' :: \ \text{Literal}\Trail \ and
  u\text{Clause} :: \ \text{Clause} \ and \ u\text{Literal} :: \ \text{Literal}
  assumes InvariantConsistent M \ and
  isUnitClause u\text{Clause} u\text{Literal} \ (\text{elements} \ M) \ and
  M' = M @ [(u\text{Literal}, \text{False})]
  shows InvariantConsistent M'
proof -
  from \ (InvariantConsistent M)
  have consistent \ (\text{elements} \ M)
    unfolding InvariantConsistent-def
  .
  from \ (isUnitClause u\text{Clause} u\text{Literal} \ (\text{elements} \ M))
  have \ ¬ \ \text{literalFalse} \ u\text{Literal} \ (\text{elements} \ M)
unfolding isUnitClause-def
by simp
{
  assume inconsistent (elements M')
  with \( M' = M \oplus \{(uLiteral, False)\} \)
  have inconsistent (elements M) \lor inconsistent [unitLiteral] \lor (\exists l. literalTrue l (elements M) \land literalFalse l [uLiteral])
    using inconsistentAppend [of elements M [uLiteral]]
    by simp
  with consistent (elements M) obtain literal::Literal
    where literalTrue literal (elements M) and literalFalse literal
    [uLiteral]
    by auto
  hence literal = opposite uLiteral
    by auto
    with literalTrue literal (elements M) \iff literalFalse uLiteral
    (elements M)
    have False
    by simp
  } thus ?thesis
unfolding InvariantConsistent-def
by auto
qed

lemma InvariantUniqAfterUnitPropagate:
  fixes M :: LiteralTrail and F :: Formula and M' :: LiteralTrail and
uClause :: Clause and uLiteral :: Literal
  assumes InvariantUniq M and
  isUnitClause uClause uLiteral (elements M) and
  M' = M \oplus \{(uLiteral, False)\}
  shows InvariantUniq M'
proof-
  from (InvariantUniq M)
  have uniq (elements M)
    unfolding InvariantUniq-def
    .
    moreover
  from (isUnitClause uClause uLiteral (elements M))
  have \(\neg literalTrue uLiteral (elements M)\)
    unfolding isUnitClause-def
    by simp
    ultimately
  show ?thesis
    using \( M' = M \oplus \{(uLiteral, False)\} \) uniqueAppendElement[of elements M uLiteral]
    unfolding InvariantUniq-def
    by simp
  qed
lemma InvariantReasonClausesAfterUnitPropagate:
  fixes M :: LiteralTrail and F :: Formula and M' :: LiteralTrail and uClause :: Clause and uLiteral :: Literal
  assumes InvariantReasonClauses F M and
  formulaEntailsClause F uClause and isUnitClause uClause uLiteral (elements M) and
  M' = M @ [(uLiteral, False)]
  shows InvariantReasonClauses F M'
proof -
  from ⟨InvariantReasonClauses F M⟩
  have ∗: (∀ literal. (literal el (elements M)) ∧ ¬ (literal el (decisions M))) →
  (∃ clause. formulaEntailsClause F clause ∧ (isReason clause literal (elements M))))
  unfolding InvariantReasonClauses-def
  by simp
  { fix literal::Literal
    assume literal el elements M' ∼ literal el decisions M'
    have ∃ clause. formulaEntailsClause F clause ∧ isReason clause literal (elements M')
      proof (cases literal el elements M)
        case True
        with assms ¬ literal el decisions M' obtiain clause::Clause
        where formulaEntailsClause F clause ∧ isReason clause literal (elements M')
        using InvariantReasonClausesHoldsForPrefixElements [of F M M' literal]
        by (auto simp add:isPrefix-def)
        thus ?thesis
        by auto
      next
        case False
        with (literal el (elements M')) (M' = M @ [(uLiteral, False)])
        have literal = uLiteral
          by simp
        with (M' = M @ [(uLiteral, False)]) (isUnitClause uClause uLiteral (elements M)) (formulaEntailsClause F uClause)
        show ?thesis
        using isUnitClauseIsReason [of uClause uLiteral elements M]
        by auto
      qed
    thus ?thesis
    unfolding InvariantReasonClauses-def
    by simp
  } thus ?thesis
  unfolding InvariantReasonClauses-def
  by simp
qed

lemma InvariantCFalseAfterUnitPropagate:
  fixes M :: LiteralTrail and F :: Formula and M' :: LiteralTrail and
uClause :: Clause and uLiteral :: Literal

assumes InvariantCFalse conflictFlag M C and
M' = M @ [(uLiteral, False)]

shows InvariantCFalse conflictFlag M' C

proof
  from ⟨InvariantCFalse conflictFlag M C⟩
  have *: conflictFlag → clauseFalse C (elements M)
    unfolding InvariantCFalse-def
  .

  { assume conflictFlag
    with ⟨M' = M @ [(uLiteral, False)]: * ⟩
    have clauseFalse C (elements M')
      by (simp add: clauseFalseAppendValuation)
  }
  thus ?thesis
  unfolding InvariantCFalse-def
  by simp

qed

Backtrack transition rule.

lemma InvariantImpliedLiteralsAfterBacktrack:
  fixes F :: Formula and M :: LiteralTrail
  assumes InvariantImpliedLiterals F M and InvariantUniq M and
  InvariantConsistent M and
  decisions M ≠ [] and formulaFalse F (elements M)
  M' = (prefixBeforeLastDecision M) @ [(opposite (lastDecision M), False)]

  shows InvariantImpliedLiterals F M'

proof
  have isPrefix (prefixBeforeLastDecision M) M
    by (simp add: isPrefixPrefixBeforeLastMarked)

  { fix l' :: Literal
    assume l' el (elements M')
    let ?p = (prefixBeforeLastDecision M)
    let ?l = lastDecision M
    have formulaEntailsLiteral F @ val2form (decisionsTo l' M') l'
      proof (cases l' el (elements ?p))
        case True
        with ⟨isPrefix ?p M⟩
        have l' el (elements M)
          using prefixElementsAreTrailElements[of ?p M]
          by auto
        with ⟨InvariantImpliedLiterals F M⟩
        have formulaEntailsLiteral F @ val2form (decisionsTo l' M') l'
          unfolding InvariantImpliedLiterals-def
          by simp
  }
moreover
from \( M' = ?p @ [(\text{opposite } ?l, \text{False})] \) True ⟨isPrefix ?p M⟩ have \((\text{decisionsTo } l' M') = (\text{decisionsTo } l' M)\)
  using prefixToElementToPrefixElement[of ?p M l'] unfolding markedElementsTo-def
by (auto simp add: prefixToElementAppend)
ultimately
show ?thesis
  by auto
next
case False
with \langle !l el (\text{elements } M') \rangle and \( M' = ?p @ [(\text{opposite } ?l, \text{False})] \)
have !l = (opposite l')
  by (auto split: split-if-asm)
hence l' = (opposite ?l)
  by simp
from :InvariantUniq M; and \( \text{markedElements } M \neq [] \)
have \( (\text{decisionsTo } ?l M) = (\text{decisions } M) \)
  unfolding InvariantUniq-def
  using markedElementsToListMarkedAreAllMarkedElements
  by auto
moreover
from \( \text{decisions } M \neq [] \).
  have !l el (\text{elements } M)
    by (simp add: lastMarkedIsMarkedElement markedElementsAreElements)
    with :InvariantConsistent M:
    have \( \neg (\text{opposite } ?l) el (\text{elements } M) \)
      unfolding InvariantConsistent-def
      by (simp add: inconsistentCharacterization)
    with :isPrefix ?p M:
    have \( \neg (\text{opposite } ?l) el (\text{elements } ?p) \)
      using prefixElementsAreTrailElements[of ?p M]
      by auto
    with \( M' = ?p @ [(\text{opposite } ?l, \text{False})] \):
    have \( (\text{decisionsTo } (\text{opposite } ?l) M') = \text{decisions } ?p \)
      using markedElementsToListAppend [of opposite ?l ?p [(opposite ?l, False)]]
      unfolding markedElementsTo-def
      by simp
moreover
from :InvariantUniq M; \( \text{decisions } M \neq [] \)
  have \( \neg !l el (\text{elements } ?p) \)
  unfolding InvariantUniq-def
  using lastMarkedNotInPrefixBeforeLastMarked[of M]
  by simp
hence \( \neg !l el (\text{decisions } ?p) \)
  by (auto simp add: markedElementsAreElements)

145
hence (removeAll ?l (decisions ?p)) = (decisions ?p)
  by (simp add: removeAll-id)
hence (removeAll ?l ((decisions ?p) @ [?l])) = (decisions ?p)
  by simp
from ⟨decisions M ≠ []⟩ False ⟨l' = (opposite ?l)⟩
have (decisions ?p) @ [?l] = (decisions M)
  using markedElementsAreElementsBeforeLastDecisionAndLast-Decision[of M]
  by simp
with ⟨removeAll ?l ((decisions ?p) @ [?l])) = (decisions ?p)⟩
have (decisions ?p) = (removeAll ?l (decisions M))
  by simp
moreover
from ⟨formulaFalse F (elements M),InvariantImpliedLiterals F M⟩
  have ¬satisfiable (F @ (val2form (decisions M)))
    using InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable[of F M]
    by simp
  from ⟨decisions M ≠ []⟩:
  have ?l el (decisions M)
    unfolding lastMarked-def
    by simp
  hence [?l] el val2form (decisions M)
    using val2FormEl[of ?l (decisions M)]
    by simp
  with ⟨¬satisfiable (F @ (val2form (decisions M)))⟩
  have formulaEntailsLiteral (removeAll [?l] (F @ val2form (decisions M))) (opposite ?l)
    using unsatisfiableFormulaWithSingleLiteralClause[of F @ val2form (decisions M) lastDecision M]
    by auto
  ultimately
  show ?thesis
    using l' = (opposite ?l)
    using formulaEntailsLiteralRemoveAllAppend[of [?l] F val2form (removeAll ?l (decisions M)) opposite ?l]
    by (auto simp add: val2FormRemoveAll)
  qed
thus ?thesis
  unfolding InvariantImpliedLiterals-def
  by auto
qed

lemma InvariantConsistentAfterBacktrack:
  fixes F::Formula and M::LiteralTrail
  assumes InvariantUniq M and InvariantConsistent M and
decisions $M \neq []$ and
$M' = (\text{prefixBeforeLastDecision } M) @ [(\text{opposite } (\text{lastDecision } M), False)]$
shows $\text{InvariantConsistent } M'$

proof–
from $\langle \text{decisions } M \neq [] \rangle \langle \text{InvariantUniq } M \rangle$
have $\neg \text{lastDecision } M \in \text{elements } (\text{prefixBeforeLastDecision } M)$
unfolding $\text{InvariantUniq-def}$
using $\text{lastMarkedNotInPrefixBeforeLastMarked}$
by simp
moreover
from $\langle \text{InvariantConsistent } M \rangle$
have $\text{consistent } (\text{elements } (\text{prefixBeforeLastDecision } M))$
unfolding $\text{InvariantConsistent-def}$
using $\text{isPrefixPrefixBeforeLastMarked[of M]}$
using $\text{isPrefixElements[of prefixBeforeLastDecision M M]}$
using $\text{consistentPrefix[of elements } (\text{prefixBeforeLastDecision } M)\text{ elements } M]$ by simp
ultimately
show $\langle \text{thesis} \rangle$
unfolding $\text{InvariantConsistent-def}$
using $\langle M' = (\text{prefixBeforeLastDecision } M) @ [(\text{opposite } (\text{lastDecision } M), False)] \rangle$
using $\text{inconsistentAppend[of elements } (\text{prefixBeforeLastDecision M}) [\text{opposite } (\text{lastDecision } M)]]$
by (auto split: split-if-asm)
qed

lemma $\text{InvariantUniqAfterBacktrack}$:
fixes $F :: \text{Formula}$ and $M :: \text{LiteralTrail}$
assumes $\text{InvariantUniq } M$ and $\text{InvariantConsistent } M$ and
decisions $M \neq []$ and
$M' = (\text{prefixBeforeLastDecision } M) @ [(\text{opposite } (\text{lastDecision } M), False)]$
shows $\text{InvariantUniq } M'$

proof–
from $\langle \text{InvariantUniq } M \rangle$
have $\text{uniq } (\text{elements } (\text{prefixBeforeLastDecision } M))$
unfolding $\text{InvariantUniq-def}$
using $\text{isPrefixPrefixBeforeLastMarked[of M]}$
using $\text{isPrefixElements[of prefixBeforeLastDecision M M]}$
using $\text{uniqListImpliesUniqPrefix}$
by simp
moreover
from $\langle \text{decisions } M \neq [] \rangle$
have $\text{lastDecision } M \in \text{elements } M$
using $\text{lastMarkedIsMarkedElement[of M]}$
using $\text{markedElementsAreElements[of lastDecision M M]}$

147
by simp
with ⟨InvariantConsistent M⟩
have ¬ opposite (lastDecision M) el (elements M)
  unfolding InvariantConsistent-def
  using inconsistentCharacterization
  by simp
hence ¬ opposite (lastDecision M) el (elements (prefixBeforeLastDecision M))
  using isPrefixPrefixBeforeLastMarked[of M]
  using isPrefixElements[of prefixBeforeLastDecision M M]
  using prefixIsSubset[of elements (prefixBeforeLastDecision M) elements M]
  by auto
ultimately
show ?thesis
using ⟨M' = (prefixBeforeLastDecision M) @ [(opposite (lastDecision M), False)]⟩
  uniqAppendElement[of elements (prefixBeforeLastDecision M) opposite (lastDecision M)]
  unfolding InvariantUniq-def
  by simp
qed

lemma InvariantVarsMAfterBacktrack:
fixes F::Formula and M::LiteralTrail
assumes InvariantVarsM M F0 Vbl
  decisions M ≠ [] and
  M' = (prefixBeforeLastDecision M) @ [(opposite (lastDecision M), False)]
shows InvariantVarsM M' F0 Vbl
proof−
  from ⟨decisions M ≠ []⟩
  have lastDecision M el (elements M)
    using lastMarkedIsMarkedElement[of M]
    using markedElementsAreElements[of lastDecision M M]
    by simp
  hence var (lastDecision M) ∈ vars (elements M)
    using valuationContainsItsLiteralsVariable[of lastDecision M elements M]
    by simp
  moreover
  have vars (elements (prefixBeforeLastDecision M)) ⊆ vars (elements M)
    using isPrefixPrefixBeforeLastMarked[of M]
    using isPrefixElements[of prefixBeforeLastDecision M M]
    using varsPrefixValuation[of elements (prefixBeforeLastDecision M) elements M]
    by auto
ultimately
show \( \text{thesis} \)
using assms
using var\text{AppendValuation}[of elements \text{(prefixBeforeLastDecision M)}] [opposite (lastDecision M)]
unfolding InvariantVarsM-def
by auto
qed

Backjump transition rule.

lemma InvariantImpliedLiteralsAfterBackjump:
fixes \( F :: \text{Formula} \) and \( M :: \text{LiteralTrail} \) and \( p :: \text{LiteralTrail} \) and \( b\text{Clause} :: \text{Clause} \)
and \( b\text{Literal} :: \text{Literal} \)
assumes InvariantImpliedLiterals \( F \) \( M \) and
\( \text{isPrefix} \) \( p \) \( M \) and \( \text{formulaEntailsClause} \) \( F \) \( b\text{Clause} \) and \( \text{isUnitClause} \)
\( b\text{Clause} \) \( b\text{Literal} \) \( (\text{elements} p) \) and
\( M' = p @ [(b\text{Literal}, \text{False})] \)
sows InvariantImpliedLiterals \( F \) \( M' \)
proof
−
from \( \langle \text{InvariantImpliedLiterals} \rangle \) \( \langle \text{isPrefix} \rangle \) \( p \) \( M \)
have InvariantImpliedLiterals \( F \) \( p \)
using InvariantImpliedLiteralsHoldsForPrefix \[of \( F \) \( M \) \( p \)\]
by simp

with assms
show \( \text{thesis} \)
using InvariantImpliedLiteralsAfterUnitPropagate \[of \( F \) \( p \) \( b\text{Clause} \) \( b\text{Literal} \) \( M' \)\]
by simp
qed

lemma InvariantVarsMAfterBackjump:
fixes \( F :: \text{Formula} \) and \( M :: \text{LiteralTrail} \) and \( p :: \text{LiteralTrail} \) and \( b\text{Clause} :: \text{Clause} \)
and \( b\text{Literal} :: \text{Literal} \)
assumes InvariantVarsM \( M \) \( F0 \) \( Vbl \) and
\( \text{isPrefix} \) \( p \) \( M \) and \( \text{var} \) \( b\text{Literal} \in \text{vars} \( F0 \cup Vbl \) and
\( M' = p @ [(b\text{Literal}, \text{False})] \)
sows InvariantVarsM \( M' \) \( F0 \) \( Vbl \)
proof
−
from \( \langle \text{InvariantVarsM} \rangle \) \( \langle \text{isPrefix} \rangle \) \( p \) \( M \)
have vars (elements \( M \) \( \subseteq \) vars \( F0 \cup Vbl \)
unfolding InvariantVarsM-def

moreover
from \( \langle \text{isPrefix} \rangle \) \( p \) \( M \)
have vars (elements \( p \) \( \subseteq \) vars (elements \( M \))
using varsPrefixValuation [of elements \( p \) elements \( M \)]
by (simp add: isPrefixElements)
ultimately

149
have \( \text{vars}(\text{elements } p) \subseteq \text{vars } F_0 \cup \text{Vbl} \)
   by simp

with \( \text{vars}(\text{elements } p) \subseteq \text{vars } F_0 \cup \text{Vbl} \) assms
show \(?\text{thesis}\)
   using InvariantVarsMAfterUnitPropagate[of \( p \) \( F_0 \) \( \text{Vbl} \) \( b\text{Literal} \) \( M' \)]
   unfolding InvariantVarsM-def
   by simp

qed

lemma InvariantConsistentAfterBackjump:
   fixes \( F::\text{Formula} \) and \( M::\text{LiteralTrail} \) and \( p::\text{LiteralTrail} \) and \( b\text{Clause}::\text{Clause} \)
   and \( b\text{Literal}::\text{Literal} \)
   assumes InvariantConsistent \( M \) and
   isPrefix \( p \) \( M \) and isUnitClause \( b\text{Clause} \) \( b\text{Literal} \) \( (\text{elements } p) \) and
   \( M' = p @ \[(b\text{Literal}, \text{False})]\]  
   shows InvariantConsistent \( M' \)
proof –
   from \( \langle \text{InvariantConsistent } M \rangle \)
   have consistent \( (\text{elements } M) \)
   unfolding InvariantConsistent-def
   
   with \( \langle \text{isPrefix } p \ M \rangle \)
   have consistent \( (\text{elements } p) \)
   using consistentPrefix [of elements \( p \) elements \( M \)]
   by (simp add: isPrefixElements)

   with assms
   show \(?\text{thesis}\)
   using InvariantConsistentAfterUnitPropagate [of \( p \) \( b\text{Clause} \) \( b\text{Literal} \) \( M' \)]
   unfolding InvariantConsistent-def
   by simp

qed

lemma InvariantUniqAfterBackjump:
   fixes \( F::\text{Formula} \) and \( M::\text{LiteralTrail} \) and \( p::\text{LiteralTrail} \) and \( b\text{Clause}::\text{Clause} \)
   and \( b\text{Literal}::\text{Literal} \)
   assumes InvariantUniq \( M \) and
   isPrefix \( p \) \( M \) and isUnitClause \( b\text{Clause} \) \( b\text{Literal} \) \( (\text{elements } p) \) and
   \( M' = p @ \[(b\text{Literal}, \text{False})]\]  
   shows InvariantUniq \( M' \)
proof –
   from \( \langle \text{InvariantUniq } M \rangle \)
   have uniq \( (\text{elements } M) \)
   unfolding InvariantUniq-def
   
   with \( \langle \text{isPrefix } p \ M \rangle \)
   have uniq \( (\text{elements } p) \)
using uniqElementsTrailImpliesUniqElementsPrefix [of p M]
by simp
with assms
show ?thesis
  using InvariantUniqAfterUnitPropagate[of p bClause bLiteral M′]
  unfolding InvariantUniq-def
by simp
qed

lemma InvariantReasonClausesAfterBackjump:
fixes F :: Formula and M :: LiteralTrail and p :: LiteralTrail and bClause :: Clause
and bLiteral ::Literal
assumes InvariantReasonClauses F M and InvariantUniq M and
isPrefix p M and isUnitClause bClause bLiteral (elements p) and
formulaEntailsClause F bClause and
M′ = p @ [(bLiteral, False)]
shows InvariantReasonClauses F M′
proof –
  from (InvariantReasonClauses F M) (InvariantUniq M) (isPrefix p M)
  have InvariantReasonClauses F p
    by (rule InvariantReasonClausesHoldsForPrefix)
  with assms
  show ?thesis
    using InvariantReasonClausesAfterUnitPropagate[of p bClause bLiteral M′]
    by simp
qed

Learn transition rule.

lemma InvariantImpliedLiteralsAfterLearn:
fixes F :: Formula and F′ :: Formula and M :: LiteralTrail and C :: Clause
assumes InvariantImpliedLiterals F M and
F′ = F @ [C]
s shows InvariantImpliedLiterals F′ M
proof –
  from (InvariantImpliedLiterals F M)
  have *: ∀ l. l el (elements M) → formulaEntailsLiteral (F @ val2form (decisionsTo l M)) l
    unfolding InvariantImpliedLiterals-def
  .
  { fix literal :: Literal
  assume literal el (elements M)
  with *
  have formulaEntailsLiteral (F @ val2form (decisionsTo literal M)) literal
  ...
by simp

**hence** formulaEntailsLiteral \( (F \at\ [C] \at\ \text{val2form} \ (\text{decisionsTo literal } M)) \) literal

**proof** -

\[ \forall \ \text{clause}::\text{Clause. clause el} \ (F \at\ \text{val2form} \ (\text{decisionsTo literal } M)) \rightarrow \text{clause el} \ (F \at\ [C] \at\ \text{val2form} \ (\text{decisionsTo literal } M)) \]

**proof** -

\{ 
  fix \ \text{clause} :: \text{Clause}
  have clause el \ (F \at\ \text{val2form} \ (\text{decisionsTo literal } M)) \rightarrow \text{clause el} \ (F \at\ [C] \at\ \text{val2form} \ (\text{decisionsTo literal } M))
  proof
    assume clause el \ (F \at\ \text{val2form} \ (\text{decisionsTo literal } M))
    thus clause el \ (F \at\ [C] \at\ \text{val2form} \ (\text{decisionsTo literal } M))
      by auto
  qed
\} thus \(?\text{thesis}\)
  by auto

qed

with \{ formulaEntailsLiteral \ (F \at\ \text{val2form} \ (\text{decisionsTo literal } M )) \} literal:

show \(?\text{thesis}\)
  by (rule formulaEntailsLiteralSubset)
qed

}\}

thus \(?\text{thesis}\)

unfolding InvariantImpliedLiterals-def

using \(F' = F \at\ [C]\)

by auto

qd

**lemma** InvariantReasonClausesAfterLearn:

fixes \( F :: \text{Formula} \) and \( F' :: \text{Formula} \) and \( M :: \text{LiteralTrail} \) and \( C :: \text{Clause} \)

**assumes** InvariantReasonClauses \( F \ M \) and

formulaEntailsClause \( F \ C \) and

\( F' = F \at\ [C]\)

**shows** InvariantReasonClauses \( F' \ M \)

**proof** -

\{ 
  fix \ \text{literal} :: \text{Literal}
  assume \text{literal el elements} \ M \land \neg \text{literal el decisions} \ M
  with \{ InvariantReasonClauses \( F \ M \); obtain clause::\text{Clause} \}
  where formulaEntailsClause \( F \ clause \) isReason \ clause literal \ (elements \ M)
    unfolding InvariantReasonClauses-def
  by auto
  from \{ formulaEntailsClause \( F \ clause \); \( F' = F \at\ [C]\) \}
\}
have \( \text{formulaEntailsClause} \ F' \ \text{clause} \)
  
  by (simp add:formulaEntailsClauseAppend)
  
  with (isReason clause literal (elements \( M \)));

have \( \exists \ \text{clause, formulaEntailsClause} \ F' \ \text{clause} \land \text{isReason clause literal (elements} \( M \)) \)
  
  by auto

\}

thus \( ?\text{thesis} \)
  
  unfolding InvariantReasonClauses-def
  
  by simp

qed

lemma InvariantVarsFAfterLearn:
  
  fixes \( F0 :: \text{Formula} \) and \( F :: \text{Formula} \) and \( F' :: \text{Formula} \) and \( C :: \text{Clause} \)

  assumes InvariantVarsF \( F \ F0 \ Vbl \) and
  
  vars \( C \subseteq (\text{vars} \ F0) \cup Vbl \) and
  
  \( F' = F @ [C] \)
  
  shows InvariantVarsF \( F' \ F0 \ Vbl \)

using assms
  
  using varsAppendFormulae[of \( F \ [C] \)]

  unfolding InvariantVarsF-def

by auto

lemma InvariantEquivalentAfterLearn:
  
  fixes \( F0 :: \text{Formula} \) and \( F :: \text{Formula} \) and \( F' :: \text{Formula} \) and \( C :: \text{Clause} \)

  assumes InvariantEquivalent \( F0 \ F \) and
  
  formulaEntailsClause \( F \ C \) and
  
  \( F' = F @ [C] \)
  
  shows InvariantEquivalent \( F0 \ F' \)

proof

  from (InvariantEquivalent \( F0 \ F \))
  
  have \( \text{equivalentFormulae} \ F0 \ F \)
    
    unfolding InvariantEquivalent-def
  
  with (formulaEntailsClause \( F \ C \)); \( \langle F' = F @ [C]\rangle \)
  
  have \( \text{equivalentFormulae} \ F0 \ (F @ [C]) \)
    
    unfolding \( \text{extendEquivalentFormulaWithEntailedClause} \) \( \text{of} \ F0 \ F \ C \)
    
    by simp

  thus \( ?\text{thesis} \)
    
    unfolding InvariantEquivalent-def
    
    using \( \langle F' = F @ [C]\rangle \)
    
    by simp

qed

lemma InvariantCEntailedAfterLearn:
  
  fixes \( F0 :: \text{Formula} \) and \( F :: \text{Formula} \) and \( F' :: \text{Formula} \) and \( C :: \text{Clause} \)
assumes InvariantCEntailed conflictFlag F C and
F’ = F ∘ [C]
shows InvariantCEntailed conflictFlag F’ C
using assms
unfolding InvariantCEntailed-def
by (auto simp add: formulaEntailsClauseAppend)

Explain transition rule.

lemma InvariantCFalseAfterExplain:
  fixes conflictFlag::bool and M::LiteralTrail and C::Clause and literal :: Literal
  assumes InvariantCFalse conflictFlag M C and
  opposite literal el C and isReason reason literal (elements M) and
  C’ = resolve C reason (opposite literal)
  shows InvariantCFalse conflictFlag M C’
unfolding InvariantCFalse-def
proof
  assume conflictFlag
  with InvariantCFalse conflictFlag M C;
  have clauseFalse C (elements M)
    unfolding InvariantCFalse-def
    by simp
  hence clauseFalse (removeAll (opposite literal) C) (elements M)
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
  moreover
  from isReason reason literal (elements M)
  have clauseFalse (removeAll literal reason) (elements M)
    unfolding isReason-def
    by simp
  ultimately
  show clauseFalse C’ (elements M)
    using C’ = resolve C reason (opposite literal)
    resolveFalseClauses [of opposite literal C elements M reason]
    by simp
qed

lemma InvariantCEntailedAfterExplain:
  fixes conflictFlag::bool and M::LiteralTrail and C::Clause and literal :: Literal and reason :: Clause
  assumes InvariantCEntailed conflictFlag F C and
  formulaEntailsClause F reason and C’ = (resolve C reason (opposite l))
  shows InvariantCEntailed conflictFlag F C’
unfolding InvariantCEntailed-def
proof
  assume conflictFlag
  with InvariantCEntailed conflictFlag F C;
  have formulaEntailsClause F C
    unfolding InvariantCEntailed-def
by simp

with ⟨formulaEntailsClause F reason⟩
show formulaEntailsClause F C'
  using ⟨C' = (resolve C reason (opposite l))⟩
  by (simp add: formulaEntailsResolvent)
qed

Conflict transition rule.

lemma invariantCFalseAfterConflict:
  fixes conflictFlag :: bool and conflictFlag' :: bool and M :: LiteralTrail
and F :: Formula and clause :: Clause and C' :: Clause
  assumes conflictFlag = False and
    formulaFalse F (elements M) and clause el F clauseFalse clause (elements M) and
    C' = clause and conflictFlag' = True
  shows InvariantCFalse conflictFlag' M C'
unfolding InvariantCFalse-def
proof
  from ⟨conflictFlag' = True⟩
  show clauseFalse C' (elements M)
    using ⟨clauseFalse clause (elements M); C' = clause⟩
    by simp
qed

lemma invariantCEntailedAfterConflict:
  fixes conflictFlag :: bool and conflictFlag' :: bool and M :: LiteralTrail
and F :: Formula and clause :: Clause and C' :: Clause
  assumes conflictFlag = False and
    formulaFalse F (elements M) and clause el F and clauseFalse clause (elements M) and
    C' = clause and conflictFlag' = True
  shows InvariantCEntailed conflictFlag' F C'
unfolding InvariantCEntailed-def
proof
  from ⟨conflictFlag' = True⟩
  show formulaEntailsClause F C'
    using ⟨clause el F; C' = clause⟩
    by (simp add: formulaEntailsItsClauses)
qed

UNSAT report

lemma unsatReport:
  fixes F :: Formula and M :: LiteralTrail and F0 :: Formula
  assumes InvariantImpliedLiterals F M and InvariantEquivalent F0
  F and
  decisions M = [] and formulaFalse F (elements M)
  shows ¬ satisfiable F0
proof−
  have formulaEntailsValuation F (elements M)
proof -
{
  fix literal::Literal
  assume literal el (elements M)
  from decisions M = []
  have decisionsTo literal M = []
  by (simp add:markedElementsEmptyImpliesMarkedElementsToEmpty)
  with literal el (elements M) (InvariantImpliedLiterals F M)
  have formulaEntailsLiteral F literal
    unfolding InvariantImpliedLiterals-def
    by auto
}
thus ?thesis
  unfolding formulaEntailsValuation-def
  by simp
qed

with (formulaFalse F (elements M))
have ~ satisfiable F
  by (simp add:formulaFalseInEntailedValuationIsUnsatisfiable)
with (InvariantEquivalent F0 F)
show ?thesis
  unfolding InvariantEquivalent-def
  by (simp add:satisfiableEquivalent)
qed

lemma unsatReportExtensiveExplain:
  fixes F :: Formula and M :: LiteralTrail and F0 :: Formula and C :: Clause and conflictFlag :: bool
  assumes InvariantEquivalent F0 F and InvariantCEntailed conflict-Flag F C and
  conflictFlag and C = []
  shows ~ satisfiable F0
proof -
  from (conflictFlag) (InvariantCEntailed conflictFlag F C)
  have formulaEntailsClause F C
    unfolding InvariantCEntailed-def
    by simp
  with (C=[])
  have ~ satisfiable F
    by (simp add:formulaUnsatisfiableImpliesEmptyClause)
  with (InvariantEquivalent F0 F)
  show ?thesis
    unfolding InvariantEquivalent-def
    by (simp add:satisfiableEquivalent)
qed

SAT Report

lemma satReport:
  fixes F0 :: Formula and F :: Formula and M::LiteralTrail
assumes \( \text{vars} F_0 \subseteq Vbl \) and \( \text{InvariantVars} F^{} F F_0 Vbl \) and \( \text{InvariantConsistent} M \) and \( \text{InvariantEquivalent} F_0 F \) and 
\( \neg \text{formulaFalse} F (\text{elements} M) \) and \( \text{vars} (\text{elements} M) \supseteq Vbl \)
shows model (\text{elements} M) F_0
proof
  from (\text{InvariantConsistent} M)
  have consistent (\text{elements} M)
    unfolding \text{InvariantConsistent-def}
  .
moreover
  from (\text{InvariantVars} F^{} F F_0 Vbl)
  have \( \text{vars} F \subseteq \text{vars} \cup Vbl \)
    unfolding \text{InvariantVars-def}
  .
with \( \text{vars} F_0 \subseteq Vbl \)
  have \( \text{vars} F \subseteq Vbl \)
    by auto
with \( \text{vars} (\text{elements} M) \supseteq Vbl \)
  have \( \text{vars} F \subseteq \text{vars} (\text{elements} M) \)
    by simp
hence \( \text{formulaTrue} F (\text{elements} M) \lor \text{formulaFalse} F (\text{elements} M) \)
  by (simp add:totalValuationForFormulaDefinesItsValue)
with \( \neg \text{formulaFalse} F (\text{elements} M) \)
  have \( \text{formulaTrue} F (\text{elements} M) \)
    by simp
ultimately
  have model (\text{elements} M) F
    by simp
with (\text{InvariantEquivalent} F_0 F)
  show \text{thesis}
    unfolding \text{InvariantEquivalent-def}
    unfolding equivalentFormulae-def
    by auto
qed

4.3 Different characterizations of backjumping

In this section, different characterization of applicability of backjumping will be given.

The clause satisfies the Unique Implication Point UIP condition if the level of all its literals is strictly lower then the level of its last asserted literal

definition
  isUIP l c M ==
    isLastAssertedLiteral (opposite l) (oppositeLiteralList c)(\text{elements} M) \land
    (\forall l'. \ l' \in c \land l' \neq l \rightarrow \text{elementLevel} (opposite l') M < \text{elementLevel} (opposite l) M)
**Backjump level** is a nonegative integer such that it is strictly lower than the level of the last asserted literal of a clause, and greater or equal than levels of all its other literals.

**definition**

\[ \text{isBackjumpLevel} \text{ level l c M} == \]
\[ \text{isContainedLiteral} (\text{opposite l}) (\text{oppositeLiteralList c})(\text{elements M}) \land \]
\[ 0 \leq \text{level} \land \text{level} < \text{elementLevel} (\text{opposite l}) \text{ M} \land \]
\[ (\forall l', l' \in c \land l' \neq l \rightarrow \text{elementLevel} (\text{opposite l'}) \text{ M} \leq \text{level}) \]

**lemma** lastAssertedLiteralHasHighestElementLevel:

*fixes literal :: Literal and clause :: Clause and M :: LiteralTrail*
*assumes isLastAssertedLiteral literal clause (elements M) and uniq (elements M)*
*shows \( \forall l'. l' \in \text{clause} \land l' \in \text{elements M} \rightarrow \text{elementLevel} l' \text{ M} <= \text{elementLevel} \text{ literal M} \)*

**proof** –

{  
fix l' :: Literal 
assume l' \( \in \) clause l' \( \in \) elements M 
hence elementLevel l' \( \in \) clause l' \( \leq \) elementLevel literal M 
proof (cases l' = literal) 
case True 
thus ?thesis 
by simp 
next 
case False 
from isLastAssertedLiteral literal clause (elements M) 
have literalTrue literal (elements M) 
\( \forall l. l \in \text{clause} \land l \neq \text{literal} \rightarrow \neg \text{precedes literal l} \) (elements M) 
by (auto simp add:isContainedLiteral-def) 
with (l' \( \in \) clause): False 
have \( \neg \text{precedes literal l'} \) (elements M) 
by simp 
with False (l' \( \in \) elements M): literalTrue literal (elements M) 
have precedes l' literal (elements M) 
using precedesTotalOrder [of l' elements M literal] 
by simp 
with uniq (elements M): 
show ?thesis 
using elementLevelPrecedesLeq [of l' literal M] 
by auto 
} 
thus ?thesis 
by simp 
qed
When backjump clause contains only a single literal, then the backjump level is 0.

**Lemma backjumpLevelZero:**

```plaintext
fixes M :: LiteralTrail and C :: Clause and l :: Literal
assumes
  isLastAssertedLiteral (opposite l) (oppositeLiteralList C) (elements M) and
  elementLevel (opposite l) M > 0
  set C = {l}
shows
  isBackjumpLevel 0 l C M
```

**Proof**

```plaintext
proof
  have ∀ l’. l’ el C ∧ l’ ≠ l → elementLevel (opposite l’) M ≤ 0
  proof
    { fix l'::Literal
      assume l' el C ∧ l' ≠ l
      hence False
        using (set C = {l})
      by auto
    } thus ?thesis
    by auto
  qed
  with (elementLevel (opposite l) M > 0)
  (isLastAssertedLiteral (opposite l) (oppositeLiteralList C) (elements M))
  shows ?thesis
    unfolding isBackjumpLevel-def
    by auto
qed
```

When backjump clause contains more than one literal, then the level of the second last asserted literal can be taken as a backjump level.

**Lemma backjumpLevelLastLast:**

```plaintext
fixes M :: LiteralTrail and C :: Clause and l :: Literal
assumes
  isUIP l C M and
  uniq (elements M) and
  clauseFalse C (elements M) and
  isLastAssertedLiteral (opposite ll) (removeAll (opposite l) (oppositeLiteralList C)) (elements M)
shows
  isBackjumpLevel (elementLevel (opposite ll) M) l C M
```

**Proof**

```plaintext
proof
  from (isUIP l C M).
  have isLastAssertedLiteral (opposite l) (oppositeLiteralList C) (elements M)
    unfolding isUIP-def
```
by simp

from (isLastAssertedLiteral (opposite ll) (removeAll (opposite l) (oppositeLiteralList C))) (elements M)

have literalTrue (opposite ll) (elements M) (opposite ll) el (removeAll (opposite l) (oppositeLiteralList C))

unfolding isLastAssertedLiteral-def
by auto

have ∀ l', l' el (oppositeLiteralList C) → literalTrue l' (elements M)
proof
{ fix l'::Literal
  assume l' el oppositeLiteralList C
  hence opposite l' el C
  using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite l' C]
  by simp
  with (clauseFalse C (elements M):
  have literalTrue l' (elements M)
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
}
thus ?thesis
by simp

qed

have ∀ l', l' el C ∧ l' ≠ l →
  elementLevel (opposite l') M ≤ elementLevel (opposite ll) M
proof
{ fix l'::Literal
  assume l' el C ∧ l' ≠ l
  hence (opposite l') el (oppositeLiteralList C) opposite l' ≠ opposite l
    using literalElListIffOppositeLiteralElOppositeLiteralList
    by auto
  hence opposite l' el (removeAll (opposite l) (oppositeLiteralList C))
    by simp

  from (opposite l' el (oppositeLiteralList C): 
  ∃ l', l' el (oppositeLiteralList C) → literalTrue l' (elements M):
  have literalTrue (opposite l') (elements M)
  by simp

  with (opposite l' el (removeAll (opposite l) (oppositeLiteralList C)))
}
\(\text{isLastAssertedLiteral} \ (\text{opposite ll}) \ (\text{removeAll} \ (\text{opposite l}) \ (\text{oppositeLiteralList} \ C)) \ (\text{elements} \ M)\): \\
\text{have} \ \text{elementLevel} \ (\text{opposite l}') \ M \leq \text{elementLevel} \ (\text{opposite ll}) \ M \\
\text{using} \ \text{lastAssertedLiteralHasHighestElementLevel}[\text{of} \ \text{opposite ll} \ \text{removeAll} \ (\text{opposite l}) \ (\text{oppositeLiteralList} \ C) \ M] \\
\text{by} \ \text{auto} \\
\} \\
\text{thus} \ ?\text{thesis} \\
\text{by} \ \text{simp} \\
\text{qed} \\
\text{moreover} \\
\text{from} \ \langle \ \text{literalTrue} \ (\text{opposite ll}) \ (\text{elements} \ M) \rangle \\
\text{have} \ \text{elementLevel} \ (\text{opposite ll}) \ M \geq 0 \\
\text{by} \ \text{simp} \\
\text{moreover} \\
\text{from} \ \langle \ (\text{opposite ll}) \ \text{el} \ (\text{removeAll} \ (\text{opposite l}) \ (\text{oppositeLiteralList} \ C)) \rangle \\
\text{have} \ l l \ \text{el} \ C \ \text{and} \ l l \neq l \\
\text{using} \ \text{literalElListIffOppositeLiteralElOppositeLiteralList}[\text{of} \ l \ C] \\
\text{by} \ \text{auto} \\
\text{from} \ \langle \ \text{isUIP} \ l \ C \ M \rangle \\
\text{have} \ \forall \ l'. \ l' \ \text{el} \ C \ \land \ l' \neq l \ \rightarrow \ \text{elementLevel} \ (\text{opposite l'}) \ M < \text{elementLevel} \ (\text{opposite l}) \ M \\
\text{unfolding} \ \text{isUIP-def} \\
\text{by} \ \text{simp} \\
\text{with} \ \langle l \ \text{el} \ C \ \text{and} \ l \neq b \rangle \\
\text{have} \ \text{elementLevel} \ (\text{opposite ll}) \ M < \text{elementLevel} \ (\text{opposite l}) \ M \\
\text{by} \ \text{simp} \\
\text{ultimately} \\
\text{show} \ ?\text{thesis} \\
\text{using} \ \langle \ \text{isLastAssertedLiteral} \ (\text{opposite l}) \ (\text{oppositeLiteralList} \ C) \ (\text{elements} \ M) \rangle \\
\text{unfolding} \ \text{isBackjumpLevel-def} \\
\text{by} \ \text{simp} \\
\text{qed} \\
\text{if UIP is reached then there exists correct backjump level.} \\
\text{lemma} \ \text{isUIPExistsBackjumpLevel}: \\
\text{fixes} \ M :: \ \text{LiteralTrail} \ \text{and} \ c :: \ \text{Clause} \ \text{and} \ l :: \ \text{Literal} \\
\text{assumes} \\
\text{clauseFalse} \ c \ (\text{elements} \ M) \ \text{and} \\
\text{isUIP} \ l \ c \ M \ \text{and} \\
\text{uniq} \ (\text{elements} \ M) \ \text{and} \\
\text{elementLevel} \ (\text{opposite l}) \ M \ > \ 0 \\
\text{shows} \\
\exists \ \text{level}. \ (\text{isBackjumpLevel} \ \text{level} \ l \ c \ M) \\
\text{proof} –
from ⟨isUIP l c M⟩
have isLastAssertedLiteral (opposite l) (oppositeLiteralList c) (elements M)
  unfolding isUIP-def
  by simp
show ?thesis
proof (cases set c = {l})
case True
  with (elementLevel (opposite l) M > 0) ⟨isLastAssertedLiteral (opposite l) (oppositeLiteralList c) (elements M)⟩
  have isBackjumpLevel 0 l c M
    using backjumpLevelZero[of l c M]
    by auto
  thus ?thesis
    by auto
next
case False
  have ∃ literal. isLastAssertedLiteral literal (removeAll (opposite l) (oppositeLiteralList c)) (elements M)
  proof −
    let ?ll = getLastAssertedLiteral (oppositeLiteralList (removeAll l c)) (elements M)
    from ⟨clauseFalse c (elements M)⟩
    have clauseFalse (removeAll l c) (elements M)
      by (simp add: clauseFalseRemove)
    moreover
    have removeAll l c ≠ []
    proof −
      have (set c) ⊆ {l} ∪ set (removeAll l c)
        by auto
    from ⟨isLastAssertedLiteral (opposite l) (oppositeLiteralList c) (elements M)⟩
    have (opposite l) el oppositeLiteralList c
      unfolding isLastAssertedLiteral-def
      by simp
    hence l ∈ c
      using literalELListIffOppositeLiteralELOppositeLiteralList[of l c]
      by simp
    hence l ∈ set c
      by simp
    { assume ¬ ?thesis
    hence set (removeAll l c) = {} 
      by simp
    with ⟨set c ⊆ {l} ∪ set (removeAll l c)⟩
    have set c ⊆ {l}
by simp
with \( l \in \text{set } c \)
have \( \text{set } c = \{ l \} \)
  by auto
with False
have False
  by simp
}
thus \( ?\text{thesis} \)
  by auto
qed
ultimately
have \( \text{isLastAssertedLiteral } ?ll (\text{oppositeLiteralList } (\text{removeAll } l \ c)) \) (elements \( M \))
  using \( \text{uniq } (\text{elements } M) \)
  using \( \text{getLast AssertedLiteral Characterization } [\text{of } \text{removeAll } l \ c \ \text{elements } M] \)
  by simp
hence \( \text{isLastAssertedLiteral } ?ll (\text{removeAll } (\text{opposite } l) \ (\text{oppositeLiteralList } c)) \) (elements \( M \))
  using \( \text{oppositeLiteralListRemove } [\text{of } l \ c] \)
  by simp
thus \( ?\text{thesis} \)
  by auto
qed
then obtain \( ll:\text{Literal} \) where \( \text{isLastAssertedLiteral } ll (\text{removeAll } (\text{opposite } l) \ (\text{oppositeLiteralList } c)) \) (elements \( M \))
  by auto
with \( \text{uniq } (\text{elements } M) \) \( \langle \text{clauseFalse } c \ (\text{elements } M) \rangle \) \( \langle \text{isUIP } l \ c \ M \rangle \)
have \( \text{isBackjumpLevel } (\text{elementLevel } ll \ M) \ l \ c \ M \)
  using \( \text{backjumpLevelLastLast } [\text{of } l \ c \ M \ \text{opposite } ll] \)
  by auto
thus \( ?\text{thesis} \)
  by auto
qed
qed

Backjump level condition ensures that the backjump clause is unit in the prefix to backjump level.

**lemma** \( \text{isBackjumpLevelEnsuresIsUnitInPrefix} \):

**fixes** \( M :: \text{LiteralTrail} \) \( \text{and } \text{conflictFlag} :: \text{bool} \) \( \text{and } c :: \text{Clause} \) \( \text{and } l :: \text{Literal} \)

**assumes** \( \text{consistent } (\text{elements } M) \) \( \text{and } \text{uniq } (\text{elements } M) \) \( \text{and } \text{clauseFalse } c \ (\text{elements } M) \) \( \text{and } \text{isBackjumpLevel } \text{level } l \ c \ M \)

**shows** \( \text{isUnitClause } c \ l \ (\text{elements } (\text{prefixToLevel } \text{level } M)) \)

**proof**

**from** \( \text{isBackjumpLevel } \text{level } l \ c \ M \)

163
have isLastAssertedLiteral (opposite l) (oppositeLiteralList c)(elements M)
  \(0 \leq \text{level} < \text{elementLevel} (\text{opposite l}) M\) and
  \(*: \forall l', l' \in c \land l' \neq l \longrightarrow \text{elementLevel} (\text{opposite l'}) M \leq \text{level}*

unfolding isBackjumpLevel-def
by auto

from isLastAssertedLiteral (opposite l)(oppositeLiteralList c) (elements M)
have l el c literalTrue (opposite l) (elements M)
using isLastAssertedCharacterization [of opposite l c elements M]
by auto

have \neg literalFalse l (elements (prefixToLevel level M))
  using \(\text{level} < \text{elementLevel} (\text{opposite l}) M\) : (0 <= \text{level} : \text{uniq} (elements M))
by (simp add: literalNotInEarlierLevelsThanItsLevel)
moreover
have \neg literalTrue l (elements (prefixToLevel level M))
proof --
  from consistent (elements M) : \neg literalTrue (opposite l) (elements M)
  have \neg literalFalse (opposite l) (elements M)
    by (auto simp add: inconsistentCharacterization)
  thus \(\text{thesis}\)
    using isPrefixPrefixToLevel[of level M]
    prefixElementsAreTrailElements[of prefixToLevel level M M]
    unfolding prefixToLevel-def
    by auto
qed
moreover
have \(\forall l', l' \in c \land l' \neq l \longrightarrow \text{literalFalse} l' (\text{elements} (\text{prefixToLevel level M}))\)
proof --
  { fix l' :: \text{Literal}
    assume l' el c l' \neq l

  from \(l' \in c\) \(\text{clauseFalse} c (\text{elements} M)\)
  have literalFalse l' (elements M)
    by (simp add: clauseFalseIffAllLiteralsAreFalse)

  have literalFalse l' (elements (prefixToLevel level M))
  proof --
    from \(l' \in c\) \(l' \neq b\)
    have elementLevel (opposite l') M <= level
      using *
      by auto

164
thus \( ?\text{thesis} \)

using \( \langle \text{literalFalse } l' (\text{elements } M) \rangle \)

\( 0 \leq \text{level} \)

\( \text{elementLevelLtLevelImpliesMemberPrefixToLevel}[\text{of opposite } l' M \text{ level}] \)

by simp

qed

\}

thus \( ?\text{thesis} \)

by auto

qed

ultimately

show \( ?\text{thesis} \)

using \( \langle l \text{ el } c \rangle \)

unfolding \( \text{isUnitClause-def} \)

by simp

qed

Backjump level is minimal if there is no smaller level which satisfies the backjump level condition. The following definition gives operative characterization of this notion.

\textbf{definition}

\( \text{isMinimalBackjumpLevel } \text{level } l \text{ c } M \equiv \)

\( \text{isBackjumpLevel } \text{level } l \text{ c } M \land \)

\( \text{(if set } c \neq \{l\} \text{ then}

\( (\exists \text{ ll. } \text{ll el } c \land \text{elementLevel } (\text{opposite } \text{ll}) M = \text{level} ) \)

\text{else}

\( \text{level} = 0 \)

\)

\textbf{lemma} \( \text{isMinimalBackjumpLevelCharacterization:} \)

\textbf{assumes}

\( \text{isUIP } l \text{ c } M \)

\( \text{clauseFalse } c \text{ (elements } M) \)

\( \text{uniq } (\text{elements } M) \)

\textbf{shows}

\( \text{isMinimalBackjumpLevel } \text{level } l \text{ c } M = \)

\( (\text{isBackjumpLevel } \text{level } l \text{ c } M \land \)

\( (\forall \text{ level'. level'} < \text{level } \rightarrow \neg \text{isBackjumpLevel } \text{level'} l \text{ c } M) ) \) \( \text{(is } ?\text{lhs } = ?\text{rhs}) \)

\textbf{proof}

assume \( ?\text{lhs} \)

show \( ?\text{rhs} \)

proof \( \text{(cases set } c = \{l\}) \)

case True

thus \( ?\text{thesis} \)

using \( ?\text{lhs} \)

unfolding \( \text{isMinimalBackjumpLevel-def} \)

by auto

next
case False
with ⟨?lhs⟩
obtain ll
where ll el c elementLevel (opposite ll) M = level isBackjumpLevel
level l c M
  unfolding isMinimalBackjumpLevel-def
  by auto
have l ≠ ll
  using (isMinimalBackjumpLevel level l c M)
  using (elementLevel (opposite ll) M = level)
  unfolding isMinimalBackjumpLevel-def
  unfolding isBackjumpLevel-def
  by auto
show ?thesis
  using (isBackjumpLevel level l c M)
  using (elementLevel (opposite ll) M = level)
  using (ll el c) l ≠ ll
  unfolding isBackjumpLevel-def
  by force
qed
next
assume ?rhs
show ?lhs
proof (cases set c = {l})
case True
  thus ?thesis
  using ⟨?rhs⟩
  using backjumpLevelZero[of l c M]
  unfolding isMinimalBackjumpLevel-def
  unfolding isBackjumpLevel-def
  by auto
next
case False
from ⟨?rhs⟩
have l el c
  unfolding isBackjumpLevel-def
  using literalElListIffOppositeLiteralElOppositeLiteralList[of l c]
  unfolding isLastAssertedLiteral-def
  by simp
let ?all = getLastAssertedLiteral (removeAll (opposite l) (oppositeLiteralList c)) (elements M)

have clauseFalse (removeAll l c) (elements M)
  using clauseFalse c (elements M);
  by (simp add: clauseFalseIffAllLiteralsAreFalse)
moreover
have removeAll l c ≠ []
proof -
{
  assume ¬thesis
  hence set (removeAll l c) = {}
    by simp
  hence set c ⊆ {l}
    by simp
  hence False
    using (set c ≠ {l})
    using (l el c)
    by auto
} thus thesis
  by auto
qed
ultimately
have isLastAssertedLiteral ?oll (removeAll (opposite l) (oppositeLiteralList c)) (elements M)
  using (uniq (elements M))
    using getLastAssertedLiteralCharacterization[of removeAll l c elements M]
  using oppositeLiteralListRemove[of l c]
  by simp
hence isBackjumpLevel (elementLevel ?oll M) l c M
  using assms
  using backjumpLevelLastLast[of l c M opposite ?oll]
  by auto

have ?oll el (removeAll (opposite l) (oppositeLiteralList c))
  using (isLastAssertedLiteral ?oll (removeAll (opposite l) (oppositeLiteralList c)) (elements M))
    unfolding isLastAssertedLiteral-def
    by simp
  hence ?oll el (oppositeLiteralList c) ?oll ≠ opposite l
    by auto
  hence opposite ?oll el c
    using literalElListIffOppositeLiteralElOppositeLiteralList[of ?oll oppositeLiteralList c]
    by simp
  from (?oll ≠ opposite l)
  have opposite ?oll ≠ l
    using oppositeSymmetry[of ?oll l]
    by simp

have elementLevel ?oll M ≥ level
proof -
{
  assume elementLevel ?oll M < level
  hence ¬isBackjumpLevel (elementLevel ?oll M) l c M
    using ⟨⟨rhs⟩⟩
by simp
with ⟨isBackjumpLevel (elementLevel ?oll M) l c M⟩
have False
  by simp
} thus ?thesis
  by force
qed
moreover
from ⟨?rhs⟩
have elementLevel ?oll M ≤ level
  using (opposite ?oll el c);
  using (opposite ?oll ≠ l);
  unfolding isBackjumpLevel-def
  by auto
ultimately
have elementLevel ?oll M = level
  by simp
show ?thesis
  using (opposite ?oll el c);
  using (elementLevel ?oll M = level)
  using (?rhs)
  using (set c ≠ {l});
  unfolding isMinimalBackjumpLevel-def
  by (auto simp del: set-removeAll)
qed
qed

lemma isMinimalBackjumpLevelEnsuresIsNotUnitBeforePrefix:
  fixes M :: LiteralTrail and conflictFlag :: bool and c :: Clause and l :: Literal
  assumes consistent (elements M) and uniq (elements M) and clauseFalse c (elements M) isMinimalBackjumpLevel level l c M and level' < level
  shows ¬ (∃ l'. isUnitClause c l' (elements (prefixToLevel level' M)))
proof−
  from ⟨isMinimalBackjumpLevel level l c M⟩
  have isUnitClause c l (elements (prefixToLevel level M))
    using assms
    using isBackjumpLevelEnsuresIsUnitInPrefix[of M c level l]
    unfolding isMinimalBackjumpLevel-def
    by simp
  hence ¬ literalFalse l (elements (prefixToLevel level M))
    unfolding isUnitClause-def
    by auto
  hence ¬ literalFalse l (elements M) ∨ elementLevel (opposite l) M > level
    using elementLevelLtLevelImpliesMemberPrefixToLevel[of l M level]
    using elementLevelLtLevelImpliesMemberPrefixToLevel[of opposite l M level]
  qed
by (force)+

have ¬ literalFalse l (elements (prefixToLevel level' M))
proof (cases ¬ literalFalse l (elements M))
  case True
  thus ?thesis
  using prefixIsSubset[of elements (prefixToLevel level' M) elements M]
  using isPrefixPrefixToLevel[of level' M]
  using isPrefixElements[of prefixToLevel level' M M]
  by auto

next
  case False
  with (¬ literalFalse l (elements M) ∨ elementLevel (opposite l) M > level)
  have level < elementLevel (opposite l) M
    by simp
  thus ?thesis
  using prefixToLevelElementsElementLevel[of opposite l level' M]
  using (level' < level)
  by auto
qed

show ?thesis
proof (cases set c ≠ {l})
  case True
  from (isMinimalBackjumpLevel level l c M)
  obtain ll
    where ll el c elementLevel (opposite ll) M = level
    using (set c ≠ {l})
    unfolding isMinimalBackjumpLevel-def
    by auto
  hence ¬ literalFalse ll (elements (prefixToLevel level' M))
    using literalNotInEarlierLevelsThanItsLevel[of level' opposite ll M]
    using (level' < level)
    by simp

  have l ≠ ll
    using (isMinimalBackjumpLevel level l c M)
    using (elementLevel (opposite ll) M = level)
    unfolding isMinimalBackjumpLevel-def
    unfolding isBackjumpLevel-def
    by auto
  
  { assume ¬ ?thesis
    then obtain l'
      where isUnitClause c l' (elements (prefixToLevel level' M))
  }
by auto
have False
proof (cases l = l')
case True
thus ?thesis
  using (l ≠ ll: ∃ll el c)
  using (∃ literalFalse ll (elements (prefixToLevel level' M)));
  using (isUnitClause c l' (elements (prefixToLevel level' M)));
  unfolding isUnitClause-def
  by auto
next
case False
have l el c
  using (isMinimalBackjumpLevel level l c M)
  unfolding isMinimalBackjumpLevel-def
  unfolding isBackjumpLevel-def
  unfolding isLastAssertedLiteral-def
  using literalElListIffOppositeLiteralElOppositeLiteralList[of l c]
  by simp
thus ?thesis
  using False
  using (∃ literalFalse l (elements (prefixToLevel level' M)));
  using (isUnitClause c l' (elements (prefixToLevel level' M)));
  unfolding isUnitClause-def
  by auto
qed

next
case False
with (isMinimalBackjumpLevel level l c M)
have level = 0
  unfolding isMinimalBackjumpLevel-def
  by simp
with (level' < level)
show ?thesis
  by simp
qed
qed

If all literals in a clause are decision literals, then UIP is reached.

lemma allDecisionsThenUIP:
  fixes M ::LiteralTrail and c::Clause
  assumes (uniq (elements M)) and
  ∀ l', l' el c → (opposite l') el (decisions M)
  isLastAssertedLiteral (opposite l) (oppositeLiteralList c) (elements M)
  shows isUIP l c M
proof –
  from isLastAssertedLiteral (opposite l) (oppositeLiteralList c) (elements M)
    have l el c (opposite l) el (elements M)
    and *: ∀ l'. l' el (oppositeLiteralList c) ∧ l' ≠ opposite l → ¬
      precedes (opposite l) l' (elements M)
    unfolding isLastAssertedLiteral-def
    using literalElListIffOppositeLiteralElOppositeLiteralList
    by auto
  with \forall l', l' el c → (opposite l') el (decisions M):
    have (opposite l) el (decisions M)
      by simp
    { fix l' :: Literal
      assume l' el c l' ≠ l
      hence opposite l' el (oppositeLiteralList c) and opposite l' ≠
        opposite l
      using literalElListIffOppositeLiteralElOppositeLiteralList[of l' c]
      by auto
      with *
      have ¬ precedes (opposite l) (opposite l') (elements M)
        by simp
      from ⟨l' el c; ∀ l. l el c → (opposite l) el (decisions M)⟩
        have (opposite l') el (decisions M)
          by auto
        hence (opposite l') el (elements M)
          by (simp add:markedElementsAreElements)
    from ⟨(opposite l) el (elements M); (opposite l') el (elements M); l' ≠ l⟩
      ⟨¬ precedes (opposite l) (opposite l') (elements M)⟩
      have precedes (opposite l') (opposite l) (elements M)
        using precedesTotalOrder[of opposite l elements M opposite l']
        by simp
      with ⟨uniq (elements M)⟩
        have elementLevel (opposite l') M ≤ elementLevel (opposite l) M
          by (auto simp add:elementLevelPrecedesLeq)
      moreover
        from ⟨uniq (elements M); ⟨(opposite l) el (decisions M); (opposite l') el (decisions M); l' ≠ l⟩
          have elementLevel (opposite l) M ≠ elementLevel (opposite l') M
            using differentMarkedElementsHaveDifferentLevels[of M opposite l opposite l']
            by simp
        ultimately
        have elementLevel (opposite l') M < elementLevel (opposite l) M
          by simp
thus \( \text{thesis} \)

using \((\text{isLastAssertedLiteral } (\text{opposite } l) (\text{oppositeLiteralList } c) (\text{elements } M))\)

unfolding \(\text{isUIP-def}\)

by simp

qed

If last asserted literal of a clause is a decision literal, then UIP is reached.

lemma \(\text{lastDecisionThenUIP}\):

\[
\begin{align*}
\text{fixes } M &: \text{LiteralTrail} \text{ and } c &: \text{Clause} \\
\text{assumes } & (\text{uniq } (\text{elements } M)) \text{ and } \\
& (\text{opposite } l) \text{ el } (\text{decisions } M) \\
& \text{clauseFalse } c \text{ (elements } M) \\
& \text{isLastAssertedLiteral } (\text{opposite } l) (\text{oppositeLiteralList } c) (\text{elements } M) \\
\text{shows } & \text{isUIP } l \ c \ M
\end{align*}
\]

proof

\[
\begin{align*}
\text{from } & (\text{isLast AssertedLiteral } (\text{opposite } l) (\text{oppositeLiteralList } c) (\text{elements } M)) \\
\text{have } & l \ c \ l \text{ el } (\text{elements } M) \\
\text{and } & \forall \ l', l' \text{ el } (\text{oppositeLiteralList } c) \land l' \neq \text{opposite } l \rightarrow \neg \text{precedes } (\text{opposite } l) \ l' \text{ (elements } M) \\
\text{unfolding } & \text{isLast AssertedLiteral-def} \\
\text{using } & \text{literalElListIffOppositeLiteralElOppositeLiteralList} \\
\text{by } & \text{auto} \\
\{ \\
\text{fix } & l' :: \text{Literal} \\
\text{assume } & l' \text{ el } c \ l' \neq l \\
\text{hence } & \text{opposite } l' \text{ el } (\text{oppositeLiteralList } c) \text{ and } \text{opposite } l' \neq \text{opposite } l \\
\text{using } & \text{literalElListIffOppositeLiteralElOppositeLiteralList[of } l' \ c] \\
\text{by } & \text{auto} \\
\text{with } & \ast \\
\text{have } & \neg \text{precedes } (\text{opposite } l) (\text{opposite } l') \text{ (elements } M) \\
\text{by } & \text{simp} \\
\text{have } & (\text{opposite } l') \text{ el } (\text{elements } M) \\
\text{using } & (l' \ c) \text{ clauseFalse } c \text{ (elements } M) \\
\text{by } & (\text{simp add: clauseFalseIffAllLiteralsAreFalse}) \\
\text{from } & (\text{opposite } l') \text{ el } (\text{elements } M) \vdash (\text{opposite } l') \text{ el } (\text{elements } M) \\
\langle l' \neq l \rangle \\
\langle \neg \text{precedes } (\text{opposite } l) (\text{opposite } l') \text{ (elements } M) \\
\text{have } & \text{precedes } (\text{opposite } l') (\text{opposite } l) \text{ (elements } M) \\
\text{using } & \text{precedesTotalOrder[of opposite } l \text{ elements } M \text{ opposite } l'] \\
\text{by } & \text{simp}
\end{align*}
\]
hence \( \text{elementLevel}(\text{opposite } l') M < \text{elementLevel}(\text{opposite } l) M \)

using \( \text{elementLevelPrecedesMarkedElementLt[of } M \text{ opposite } l' \text{ opposite } l] \)

using \( (\text{uniq } (\text{elements } M)) \)

using \( (\text{opposite } l \text{ el } (\text{decisions } M)) \)

using \( (l' \neq l) \)

by simp

\}

thus \?thesis

using \( (\text{isLastAssertedLiteral } (\text{opposite } l) (\text{oppositeLiteralList } c)\text{ (elements } M)) \)

unfolding \( \text{SatSolverVerification.isUIP-def} \)

by simp

qed

If all literals in a clause are decision literals, then there exists a backjump level for that clause.

\begin{lemma}
allDecisionsThenBackjumpLevel:
\end{lemma}

\begin{proof}
\end{proof}

\begin{proof}
\end{proof}

\begin{proof}
\end{proof}

\begin{proof}
\end{proof}

\begin{proof}
\end{proof}

\section{173}
with \( \forall l', l' \in A \rightarrow (\text{opposite } l') \in (\text{decisions } M) \)
have \( (\text{opposite } l') \in (\text{decisions } M) \)
by simp
hence \( \text{literalFalse } l' (\text{elements } M) \)
using markedElementsAreElements
by simp
}
thus \( ?\text{thesis} \)
using clauseFalseIffAllLiteralsAreFalse
by simp
qed
ultimately
show \( ?\text{thesis} \)
using \( \text{uniq } (\text{elements } M) \)
using isUIPExistsBackjumpLevel
by simp
qed

\textit{Explain} is applicable to each non-decision literal in a clause.

\textbf{Lemma} \textbf{explainApplicableToEachNonDecision}:

\textbf{Fixes} \( F :: \text{Formula} \) and \( M :: \text{LiteralTrail} \) and \( \text{conflictFlag :: bool} \) and \( C :: \text{Clause} \) and \( \text{literal :: Literal} \)

\textbf{Assumes} \( \text{InvariantReasonClauses } F \) and \( \text{InvariantCFalse conflictFlag } M \) and \( \text{conflictFlag } = \text{True} \) and \( \text{opposite literal } el C \) and \( \neg \text{literal } el (\text{decisions } M) \)

\textbf{Shows} \( \exists \text{ clause. } \text{formulaEntailsClause } F \) and \( \text{isReason } \text{clause literal (elements } M) \)

\textbf{Proof} –

\textbf{From} \( \text{conflictFlag } = \text{True} \) \( \langle \text{InvariantCFalse conflictFlag } M \rangle \)
\textbf{Have} \( \text{clauseFalse } C \) \( \langle \text{elements } M \rangle \)
\textbf{Unfolding} \( \text{InvariantCFalse-def} \)
by simp
\textbf{With} \( \langle \text{opposite literal } el C \rangle \)
\textbf{Have} \( \text{literalTrue } \) \( \langle \text{elements } M \rangle \)
by \( \langle \text{auto simp add:clauseFalseIffAllLiteralsAreFalse} \rangle \)
\textbf{With} \( \langle \neg \text{literal } el (\text{decisions } M) \rangle \) \( \langle \text{InvariantReasonClauses } F \rangle \)
\textbf{Show} \( ?\text{thesis} \)
\textbf{Unfolding} \( \text{InvariantReasonClauses-def} \)
by \( \langle \text{auto} \rangle \)
\textbf{Qed}

\textbf{4.4 Termination}

In this section different ordering relations will be defined. These well-founded orderings will be the basic building blocks of termination orderings that will prove the termination of the SAT solving procedures
First we prove a simple lemma about acyclic orderings.

**lemma** transIrreflexiveOrderingIsAcyclic:
**assumes** trans r and $\forall \ x. (x, x) \notin r$
**shows** acyclic r

**proof** (rule acyclicI)
\[
\begin{align*}
\text{assume } & \exists \ x. (x, x) \in r^+ \\
\text{then obtain } & x \text{ where } (x, x) \in r^+ \\
& \text{by } auto \\
\text{moreover } & \text{from } (\text{trans r}) \\
& \text{have } r^+ = r \\
& \text{by } (\text{rule trancl-id}) \\
\text{ultimately } & \text{have } (x, x) \in r \\
& \text{by } simp \\
& \text{with } \forall \ x. (x, x) \notin r \\
& \text{have } False \\
& \text{by } simp \\
\text{thus } & \forall \ x. (x, x) \notin r^+ \\
& \text{by } auto
\end{align*}
\]
**qed**

4.4.1 Trail ordering

We define a lexicographic ordering of trails, based on the number of literals on the different decision levels. It will be used for transition rules that change the trail, i.e., for *Decide*, *UnitPropagate*, *Backjump* and *Backtrack* transition rules.

**definition** decisionLess = \{(l1::('a*bool), l2::('a*bool)). isDecision l1 \land \neg isDecision l2\}

**definition** lexLess = \{(M1::'a Trail, M2::'a Trail). (M2, M1) \in lexord decisionLess\}

Following several lemmas will help prove that application of some DPLL-based transition rules decreases the trail in the lexLess ordering.

**lemma** lexLessAppend:
**assumes** $b \neq []$
**shows** $(a @ b, a) \in lexLess$

**proof**
\[
\begin{align*}
& \text{from } (b \neq []) \\
& \text{have } \exists \ aa \ list. b = aa \# \ list \\
& \quad \text{by } (\text{simp add: neq-Nil-conv}) \\
& \text{then obtain } aa::'a \times \text{bool and } list :: 'a \ Trail
\end{align*}
\]

175
where $b = aa$ # list
by auto
thus $\text{thesis}$
unfolding lexLess-def
unfolding lexord-def
by simp
qed

lemma lexLessBackjump:
assumes $p = \text{prefixToLevel level a}$ and $\text{level} \geq 0$ and $\text{level} < \text{currentLevel a}$
shows $(p \@[\{(x, False)\}], a) \in \text{lexLess}$
proof–
from assms
have $\exists$ rest. $\text{prefixToLevel level a} \@[\text{rest}] = a \land \text{rest} \neq [] \land \text{isDecision (hd rest)}$
using isProperPrefixPrefixToLevel
by auto
with $(p = \text{prefixToLevel level a})$
obtain rest
where $p \@[\text{rest}] = a \land \text{rest} \neq [] \land \text{isDecision (hd rest)}$
by auto
thus $\text{thesis}$
unfolding lexLess-def
using lexord-append-left-rightI[of hd rest $(x, False)$ decisionLess $p$
$\text{tl rest} []$]
unfolding decisionLess-def
by simp
qed

lemma lexLessBacktrack:
assumes $p = \text{prefixBeforeLastDecision a decisions a} \neq []$
shows $(p \@[\{(x, False)\}], a) \in \text{lexLess}$
using assms
using prefixBeforeLastMarkedIsPrefixBeforeLastLevel[of a]
using lexLessBackjump[of $p \text{currentLevel a} - 1 a$]
unfolding currentLevel-def
by auto

The following several lemmas prove that $\text{lexLess}$ is acyclic. This property will play an important role in building a well-founded ordering based on $\text{lexLess}$. 

lemma transDecisionLess:
shows trans $\text{decisionLess}$
proof–
{fix $x::('a*bool)$ and $y::('a*bool)$ and $z::('a*bool)$
assume $(x, y) \in \text{decisionLess}$
hence $\neg \text{isDecision y}$

176
unfolding decisionLess-def
  by simp
moreover
assume \((y, z) \in \text{decisionLess}\)
hence isDecision \(y\)
  unfolding decisionLess-def
  by simp
ultimately
have False
  by simp
hence \((x, z) \in \text{decisionLess}\)
  by simp
}
thus \(?\text{thesis}\)
  unfolding trans-def
  by blast
qed

lemma translexLess:
  shows \(\text{trans lexLess}\)
proof (}
  fix \(x :: \text{a Trail}\) and \(y :: \text{a Trail}\) and \(z :: \text{a Trail}\)
  assume \((x, y) \in \text{lexLess}\) and \((y, z) \in \text{lexLess}\)
  hence \((x, z) \in \text{lexLess}\)
    using lexord-trans transDecisionLess
    unfolding lexLess-def
    by simp
  }
thus \(?\text{thesis}\)
  unfolding trans-def
  by blast
qed

lemma irreflexiveDecisionLess:
  shows \((x, x) \notin \text{decisionLess}\)
unfolding decisionLess-def
by simp

lemma irreflexiveLexLess:
  shows \((x, x) \notin \text{lexLess}\)
using lexord-irreflexive[of decisionLess x] irreflexiveDecisionLess
unfolding lexLess-def
by auto

lemma acyclicLexLess:
  shows acyclic lexLess
proof (rule transIrreflexiveOrderingIsAcyclic)

177
show \( \text{trans lexLess} \)
using \( \text{translexLess} \).

show \( \forall \ x. (x, x) \notin \text{lexLess} \)
using \( \text{irreflexiveLexLess} \)
by auto

qed

The `lexLess` ordering is not well-founded. In order to get a well-founded ordering, we restrict the `lexLess` ordering to consistent and uniq trails with fixed variable set.

**definition** `lexLessRestricted ( Vbl:: Variable set ) == \{(M1, M2). 
  \text{vars (elements M1)} \subseteq \text{Vbl} \land \text{consistent (elements M1)} \land \text{uniq (elements M1)} \land 
  \text{vars (elements M2)} \subseteq \text{Vbl} \land \text{consistent (elements M2)} \land \text{uniq (elements M2)} \land 
  (M1, M2) \in \text{lexLess}\}\`

First we show that the set of those trails is finite.

**lemma** `finiteVarsClause`:
fixes \( c :: \text{Clause} \)
shows \( \text{finite (vars c)} \)
by \( \text{(induct c) auto} \)

**lemma** `finiteVarsFormula`:
fixes \( F :: \text{Formula} \)
shows \( \text{finite (vars F)} \)
proof \( \text{(induct F)} \)
case \( \text{Cons c F} \)
thus \(?case\)
  using `finiteVarsClause[of c]`
  by simp
qed simp

**lemma** `finiteListDecompose`:
shows \( \text{finite \{(a, b). l = a @ b\}} \)
proof \( \text{(induct l)} \)
case \( \text{Nil} \)
thus \(?case\)
  by simp

next
case \( \text{Cons x l'} \)
thus \(?case\)
proof
  let \(?S\ l = \{(a, b). l = a @ b\}\)
  let \(?S'\ x l' = \{(a', b). a' = [] \land b = (x \# l') \lor 
    (\exists a. a' = x \# a \land (a, b) \in (?S\ l'))\}\)
  have \(?S\ (x \# l') = ?S'\ x l'\)
  proof
show \( ?S (x \neq l') \subseteq ?S' x l' \)

proof
  fix \( k \)
  assume \( k \in ?S (x \neq l') \)
  then obtain \( a \) and \( b \)
  where \( k = (a, b) x \neq l' = a \otimes b \)
  by auto
  then obtain \( a' \) where \( a' = x \neq a \)
  by auto
  from \( (k = (a, b), (x \neq l' = a \otimes b) \)
  show \( k \in ?S' x l' \)
    using \( SimpleLevi[of \ a \ b \ x \ l'] \)
  by auto

qed

next

show \( ?S' x l' \subseteq ?S (x \neq l') \)

proof
  fix \( k \)
  assume \( k \in ?S' x l' \)
  then obtain \( a' \) and \( b \) where
  \[
  k = (a', b) \quad a' = [] \land b = x \neq l' \lor (\exists a . a' = x \neq a \land (a, b) \in ?S l')
  \]
  by auto
  moreover
  \{
  assume \( a' = [] \land b = x \neq l' \)
  with \( \langle k = (a', b) \rangle \)
  have \( k \in ?S (x \neq l') \)
  by simp
  \}
  moreover
  \{
  assume \( \exists a . a' = x \neq a \land (a, b) \in ?S l' \)
  then obtain \( a \) where
  \( a' = x \neq a \land (a, b) \in ?S l' \)
  by auto
  with \( \langle k = (a', b) \rangle \)
  have \( k \in ?S (x \neq l') \)
  by auto
  \}
  ultimately
  show \( k \in ?S (x \neq l') \)
  by auto

qed

qed

moreover

have \( ?S' x l' = \{(a', b) . a' = [] \land b = x \neq l'\} \cup \{(a', b) . \exists a . a' = x \neq a \land (a, b) \in ?S l'\} \)

179
by auto
moreover
have finite \((a', b). \exists a. a' = x \# a \land (a, b) \in ?S \, l)\)
proof
  let ?h = \(\lambda (a, b). (x \# a, b)\)
  have \((a', b). \exists a. a' = x \# a \land (a, b) \in ?S \, l) = ?h ' \{(a, b). l' = a \mathbin{\oslash} b\}\)
  by auto
  thus ?thesis
  using Cons(1)
  by auto
qed
moreover
have finite \((a', b). a' = \[] \land b = x \# l'\)
by auto
ultimately
show ?thesis
by auto
qed

lemma finiteListDecomposeSet:
fixes \(L :: 'a list set\)
assumes finite \(L\)
sows finite \((a, b). \exists l. l \in L \land l = a \mathbin{\oslash} b\)
proof
  have \((a, b). \exists l. l \in L \land l = a \mathbin{\oslash} b\) = \(\bigcup l \in L. \{(a, b). l = a \mathbin{\oslash} b\}\)
  by auto
moreover
have finite \((\bigcup l \in L. \{(a, b). l = a \mathbin{\oslash} b\})\)
proof (rule finite-UN-I)
  from finite \(L\)
  show finite \(L\)
  next
  fix \(l\)
  assume \(l \in L\)
  show finite \((a, b). l = a \mathbin{\oslash} b\)
  by (rule finiteListDecompose)
qed
ultimately
show ?thesis
by simp
qed

lemma finiteUniqAndConsistentTrailsWithGivenVariableSet:
fixes \(V :: 'a Variable set\)
assumes finite \(V\)
shows finite \( \{M::\text{LiteralTrail}. \text{vars (elements M)} = V \land \text{uniq (elements M)} \land \text{consistent (elements M)} \} \)

using assms

proof induct

case empty

thus ?case

proof –

have ?trails \{\} = \{M. M = []\} (is ?lhs = ?rhs)

proof

show ?lhs \subseteq ?rhs

proof

fix M::\text{LiteralTrail}

assume M \in ?lhs

hence M = []

by (induct M) auto

thus M \in ?rhs

by simp

qed

next

show ?rhs \subseteq ?lhs

proof

fix M::\text{LiteralTrail}

assume M \in ?rhs

hence M = []

by simp

thus M \in ?lhs

by (induct M) auto

qed

moreover

have finite \{M. M = []\}

by auto

ultimately

show ?thesis

by auto

qed

next

case (insert v V')

thus ?case

proof –

let ?trails' V' = \{(M::\text{LiteralTrail}). \exists M' l d M''.

M = M' @ ([l, d]) @ M'' \land

M' @ M'' \in (?trails V') \land

l \in \{\text{Pos v, Neg v}\} \land

d \in \{\text{True, False}\}\}

have ?trails (insert v V') = ?trails' V'

(is ?lhs = ?rhs)

proof
show ?lhs ⊆ ?rhs

proof
  fix M::LiteralTrail
  assume M ∈ ?lhs
  hence vars (elements M) = insert v V’ uniq (elements M) consistent (elements M)
    by auto
  hence v ∈ vars (elements M)
    by simp
  hence ∃ l. l el elements M ∧ var l = v
    by (induct M) auto
  then obtain l where l el elements M var l = v
    by auto
  hence ∃ M’ M'' d. M = M’ @ [(l, d)] @ M''
proof (induct M)
  case (Cons m M1)
  thus ?case
  proof (cases l = (element m))
    case True
    then obtain d where m = (l, d)
      using eitherMarkedOrNotMarkedElement[of m]
      by auto
    hence m # M1 = [] @ [(l, d)] @ M1
      by simp
    then obtain M’ M'' d where m # M1 = M’ @ [(l, d)] @ M''
      ..
    thus ?thesis
      by auto
  next
    case False
    with l el elements (m # M1)
    have l el elements M1
      by simp
    with Cons(1) (var l = v)
    obtain M1’ M’’ d where M1 = M1’ @ [(l, d)] @ M’’
      by auto
    hence m # M1 = (m # M1’) @ [(l, d)] @ M’’
      by simp
    then obtain M’ M’’ d where m # M1 = M’ @ [(l, d)] @ M’’
      ..
    thus ?thesis
      by auto
  qed
qed simp
then obtain M’ M’’ d where M = M’ @ [(l, d)] @ M’’
  by auto

moreover
from ⟨var \( l = v \)⟩
have \( l : \{\text{Pos } v, \text{Neg } v\} \)
  by (cases \( l \)) auto
moreover
have ∗: \( \text{vars } (\text{elements } (M' @ M'')) = \text{vars } (\text{elements } M') \cup \text{vars } (\text{elements } M'') \)
  using varsAppendClauses[of elements \( M' \) elements \( M'' \)]
  by simp
from ⟨\( M = M' @ [(l, d)] @ M'' \) (var \( l = v \))⟩
have **: \( \text{vars } (\text{elements } M) = (\text{vars } (\text{elements } M')) \cup \{v\} \cup (\text{vars } (\text{elements } M'')) \)
  using varsAppendClauses[of elements \( M' \) elements \( [(l, d)] @ M'' \)]
  by simp
have ***: \( \text{vars } (\text{elements } M) = \text{vars } (\text{elements } (M' @ M'')) \cup \{v\} \)
  using **
  by simp
have \( M' @ M'' \in (?\text{trails } V) \)
proof−
  from ⟨uniq (elements \( M\)) ⟩ ⟨\( M = M' @ [(l, d)] @ M'' \)⟩
  have uniq (elements \( M' @ M'' \))
    by (auto iff: uniqAppendIff)
moreover
have consistent (elements \( M' @ M'' \))
proof−
  { 
    assume ¬ consistent (elements \( M' @ M'' \))
    then obtain \( l' \) where literalTrue \( l' \) (elements \( M' @ M'' \))
    by (auto simp add: inconsistentCharacterization)
    with ⟨\( M = M' @ [(l, d)] @ M'' \)⟩
    have literalTrue \( l' \) (elements \( M \)) literalFalse \( l' \) (elements \( M' @ M'' \))
      by auto
    hence ¬ consistent (elements \( M \))
      by (auto simp add: inconsistentCharacterization)
    with ⟨consistent (elements \( M \))⟩
    have False
      by simp
  }
thus ?thesis
  by auto
qed
moreover
have \( v \notin \text{vars } (\text{elements } (M' @ M'')) \)
proof−
  { 
  
}
assume \( v \in \text{vars} \ (\text{elements} \ (M' @ M'')) \)

with *

have \( v \in \text{vars} \ (\text{elements} \ M') \lor v \in \text{vars} \ (\text{elements} \ M'') \)

by simp

moreover

\{
  assume \( v \in (\text{vars} \ (\text{elements} \ M')) \)
  hence \( \exists \ l. \ \text{var} \ l = v \land l \ el \ \text{elements} \ M' \)
    by (induct \( M' \)) auto
  then obtain \( l' \) where \( \text{var} \ l' = v \ l' \ el \ \text{elements} \ M' \)
    by auto
  from \( \langle \text{var} \ l = v \rangle \langle \text{var} \ l' = v \rangle \)
  have \( l = l' \lor \text{opposite} \ l = l' \)
    using literalsWithSameVariableAreEqualOrOpposite[of \( l \ l' \)]
    by simp

moreover

\{
  assume \( l = l' \)
  with \( \langle l' \ el \ \text{elements} \ M' \rangle \langle M = M' @ [(l, d)] @ M''' \rangle \)
  have \( \neg \ \text{uniq} \ (\text{elements} \ M) \)
    by (auto iff: uniqAppendIff)
  with \( \langle \text{uniq} \ (\text{elements} \ M) \rangle \)
  have False
    by simp
\}

moreover

\{
  assume \( \text{opposite} \ l = l' \)
  have \( \neg \ \text{consistent} \ (\text{elements} \ M) \)
    proof
      from \( \langle l' \ el \ \text{elements} \ M' \rangle \langle M = M' @ [(l, d)] @ M''' \rangle \)
      have literalTrue \( l' \) (elements M)
        by simp
      moreover
      from \( \langle l' \ el \ \text{elements} \ M' \rangle \langle \text{opposite} \ l = l' \rangle \langle M = M' @ [(l, d)] @ M''' \rangle \)
      have literalFalse \( l' \) (elements M)
        by simp
      ultimately
      show \( \langle ?\text{thesis} \rangle \)
        by (auto simp add: inconsistentCharacterization)
    qed
  with \( \langle \text{consistent} \ (\text{elements} \ M) \rangle \)
  have False
    by simp
\}

ultimately

have False
by auto 

} moreover 
{ 
  assume \( v \in \text{vars(\text{elements M''})} \) 
  hence \( \exists \ l. \ \text{var l} = v \land l \in \text{elements M''} \) 
  by (induct M'') auto 
  then obtain \( l' \) where \( \text{var l'} = v \land l' \in \text{elements M''} \) 
  by auto 
  from \( \langle \text{var l} = v \rangle \langle \text{var l'} = v \rangle \) 
  have \( l = l' \lor \text{opposite l} = l' \) 
  using literalsWithSameVariableAreEqualOrOpposite[of \( l l' \)] 
  by simp 
} moreover 
{ 
  assume \( l = l' \) 
  with \( \langle l' \in \text{elements M''} \rangle \langle M = M' [\{l, d\}] @ M'' \rangle \) 
  have \( \neg \text{uniq(\text{elements M})} \) 
  by (auto iff: uniqAppendIff) 
  with \( \langle \text{uniq(\text{elements M})} \rangle \) 
  have \( \text{False} \) 
  by simp 
} moreover 
{ 
  assume opposite \( l = l' \) 
  have \( \neg \text{consistent (\text{elements M})} \) 
  proof- 
  from \( \langle l' \in \text{elements M''} \rangle \langle M = M' [\{l, d\}] @ M'' \rangle \) 
  have literalTrue \( l' \) (\text{elements M}) 
  by simp 
  moreover 
  from \( \langle l' \in \text{elements M''} \rangle \langle \text{opposite l} = l' \rangle \langle M = M' \rangle \) @ \( [\{l, d\}] @ M'' \rangle \) 
  have literalFalse \( l' \) (\text{elements M}) 
  by simp 
  ultimately 
  show \( \text{?thesis} \) 
  by (auto simp add: inconsistentCharacterization) 
  qed 
  with \( \langle \text{consistent (\text{elements M})} \rangle \) 
  have \( \text{False} \) 
  by simp 
} ultimately 
have \( \text{False} \) 
by auto
ultimately
have False
by auto

} thus ?thesis
by auto
qed
from

∗∗∗∗∗∗
⟨v ∉ vars (elements (M’ @ M”))⟩
⟨vars (elements M) = insert v V’⟩
(\neg v \in V’)
have vars (elements (M’ @ M”)) = V’
by (auto simp del: vars-def-clause)
ultimately
show ?thesis
by simp
qed
ultimately
show M ∈ ?rhs
by auto
qed
next
show ?rhs ⊆ ?lhs
proof
fix M :: LiteralTrail
assume M ∈ ?rhs
then obtain M’ M” l d where
M = M’ @ [(l, d)] @ M”
vars (elements (M’ @ M”)) = V’
uniq (elements (M’ @ M”)) consistent (elements (M’ @ M”))
l ∈ \{Pos v, Neg v\}
by auto
from (l ∈ \{Pos v, Neg v\})
have var l = v
by auto
have *: vars (elements (M’ @ M”)) = vars (elements M’) ∪ vars (elements M”)
using varsAppendClauses[of elements M’ elements M”]
by simp
from (var l = v) (M = M’ @ [(l, d)] @ M”)
have **: vars (elements M) = vars (elements M’) ∪ \{v\} ∪ vars (elements M”)
using varsAppendClauses[of elements M’ elements ([(l, d)] @ M”)]
using varsAppendClauses[of elements [(l, d)] elements M’”]
by simp
from * * * (vars (elements (M’ @ M”)) = V’)
have vars (elements M) = insert v V’
by (auto simp del: vars-def-clause)
moreover
from *
  (\var l = v)
  (v \notin V')
  (vars (elements (M' @ M''))) = V'
have \var l \notin vars (elements M') \var l \notin vars (elements M'')
  by auto
from (\var l \notin vars (elements M'))
  have \neg literalTrue \, l (elements M') \neg literalFalse \, l (elements M')
    using valuationContainsItsLiteralsVariable[of \, l elements M']
    using valuationContainsItsLiteralsVariable[of opposite \, l elements M']
  by auto
from (\var l \notin vars (elements M''))
  have \neg literalTrue \, l (elements M'') \neg literalFalse \, l (elements M'')
    using valuationContainsItsLiteralsVariable[of \, l elements M'']
    using valuationContainsItsLiteralsVariable[of opposite \, l elements M'']
  by auto
have uniq (elements M)
  using (\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\M')
  \neg literalTrue \, l (elements M') \neg literalFalse \, l (elements M')
    using oppositeIsDifferentFromLiteral[of \, l]
    by (auto split: split-if-asm)
  with (\neg literalFalse \, l' (elements M') )
    (\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\M'')
    show \:\:\\thesis
by auto

next
case False
with ⟨\text{literalTrue } l' (elements } M\rangle \langle M = M' @ [(l, d)] @ M''\rangle

have \text{literalTrue } l' (elements } (M' @ M''))
by (auto split: split-if-asm)
with ⟨\text{consistent } (elements } (M' @ M''))⟩
have \text{\neg literalFalse } l' (elements } (M' @ M''))
by (auto simp add: inconsistentCharacterization)
with ⟨\text{literalFalse } l' (elements } M\rangle \langle M = M' @ [(l, d)] @ M''\rangle

have \text{opposite } l' = l
by (auto split: split-if-asm)
with ⟨\text{var } l = v⟩
have \text{var } l' = v
by auto
with ⟨\text{literalTrue } l' (elements } (M' @ M''))⟩ \langle \text{vars } (elements } (M' @ M'')) = V'\rangle
have \text{\neg literalFalse } l' (elements } (M' @ M''))
using \text{valuationContainsItsLiteralsVariable }of \text{l' elements } (M' @ M'')]
by simp
with ⟨\text{v } / \text{\in } V'⟩
show \?thesis
by simp
qed

thus \?thesis
by auto
qed

ultimately
show \text{M } \in \?\text{lhs }
by auto
qed

moreover
let \text{\?f } = \lambda ((M', M''), l, d). \text{M' @ [(l, d)] @ M'' }
let \text{\?Mset } = \{(M', M'')\}. \text{M' @ M'' } \in \?\text{trails } V'\}
let \text{\?Set } = \{\text{Pos } v, \text{Neg } v\}
let \text{\?dSet } = \{\text{True}, \text{False}\}
have \?\text{trails'} V' = \?f \cdot (\?\text{Mset } \times \?\text{lSet } \times \?\text{dSet}) \text{ (is } \text{?lhs } = \?\text{rhs})
proof
show \text{?lhs } \subseteq \?\text{rhs }
proof
fix \text{M : LiteralTrail}
assume \text{M } \in \?\text{lhs }
then obtain \text{M' M'' l d}
where \text{P: } M = M' @ [(l, d)] @ M'' M' @ M'' \in \?\text{trails } V'

188
\( l \in \{\text{Pos} \: v, \text{Neg} \: v\} \quad d \in \{\text{True}, \text{False}\} \)
by auto
show \( M \in \text{?rhs} \)
proof
from \( P\)
show \( M = \text{?f} ((M', M''), l, d) \)
by simp
next
from \( P\)
show \(( (M', M''), l, d) \in \text{?Mset} \times \text{?lSet} \times \text{?dSet} \)
by auto
qed
qed
next
show \( \text{?rhs} \subseteq \text{?lhs} \)
proof
fix \( M :: \text{LiteralTrail} \)
assume \( M \in \text{?rhs} \)
then obtain \( p, l, d \) where \( P: M = \text{?f} (p, l, d) \) \( p \in \text{?Mset} \)
\( l \in \text{?lSet} \)
\( d \in \text{?dSet} \)
by auto
from \( \langle \: p \in \text{?Mset} \: \rangle \)
obtain \( M', M'' \)
where \( M' @ M'' \in \text{?trails} \: V' \)
by auto
thus \( M \in \text{?lhs} \)
using \( P\)
by auto
qed
qed
moreover
have \( \text{?Mset} = \{ (M', M''): \exists l. l \in \text{?trails} \: V' \wedge l = M' @ M'' \} \)
by auto
hence finite \( \text{?Mset} \)
using insert(3)
using finiteListDecomposeSet[\( \text{of} \text{?trails} \: V' \)]
by simp
ultimately
show \( \text{?thesis} \)
by auto
qed
qed

lemma \( \text{finiteUniqAndConsistentTrailsWithGivenVariableSuperset} \):
fixes \( V :: \text{Variable set} \)
assumes \( \text{finite} \: V \)
shows \( \text{finite} \ (\{ (M :: \text{LiteralTrail}). \text{vars} (\text{elements} \: M) \subseteq V \wedge \text{uniq} (\text{elements} \: M) \wedge \text{consistent} (\text{elements} \: M) \}) \) (is \( \text{finite} \ (\text{?trails} \: V) \))
proof
have \( \{ M. \text{vars} (\text{elements} \: M) \subseteq V \wedge \text{uniq} (\text{elements} \: M) \wedge \text{consistent} \)
\[(\text{elements } M)\] = 
\[\bigcup v \in \text{Pow } V. \{\text{M. vars (elements } M) = v \land \text{uniq (elements } M) \land \text{consistent (elements } M)\}\]
by auto

moreover
have finite \((\bigcup v \in \text{Pow } V. \{\text{M. vars (elements } M) = v \land \text{uniq (elements } M) \land \text{consistent (elements } M)\}\)\)
proof (rule finite-UN-I)
from (finite V)
show finite (Pow V)
by simp
next
fix v
assume v \in \text{Pow } V
with (finite V)
have finite v
by (auto simp add: finite-subset)
thus finite \(\{\text{M. vars (elements } M) = v \land \text{uniq (elements } M) \land \text{consistent (elements } M)\}\)
using finiteUniqAndConsistentTrailsWithGivenVariableSuper-set[of v]
by simp
qed
ultimately
show \(?\text{thesis}\)
by simp
qed

Since the restricted ordering is acyclic and its domain is finite, it has to be well-founded.

lemma \text{wfLexLessRestricted}:
assumes \text{finite Vbl}
shows \text{wf (lexLessRestricted Vbl)}
proof (rule finite-acyclic-wf)
show finite (lexLessRestricted Vbl)
proof
let \(?X = \{(M1, M2). \text{consistent (elements } M1) \land \text{uniq (elements } M1) \land \text{vars (elements } M1) \subseteq \text{Vbl} \land \text{consistent (elements } M2) \land \text{uniq (elements } M2) \land \text{vars (elements } M2) \subseteq \text{Vbl}\}\)
let \(?Y = \{\text{M. vars (elements } M) \subseteq \text{Vbl} \land \text{uniq (elements } M) \land \text{consistent (elements } M)\}\)
have \(?X = ?Y \times ?Y\)
by auto
moreover
have finite ?Y
using finiteUniqAndConsistentTrailsWithGivenVariableSuper-set[of Vbl]
(finite Vbl)
by auto
ultimately
have finite ?X
  by simp
moreover
have lexLessRestricted Vbl ⊆ ?X
  unfolding lexLessRestricted-def
  by auto
ultimately
show thesis
  by (simp add: finite-subset)
qed

next
show acyclic (lexLessRestricted Vbl)
proof -
  {
    assume ¬ thesis
    then obtain x where (x, x) ∈ (lexLessRestricted Vbl)^+
      unfolding acyclic-def
      by auto
    have lexLessRestricted Vbl ⊆ lexLess
      unfolding lexLessRestricted-def
      by auto
    have (lexLessRestricted Vbl)^+ ⊆ lexLess`
      proof
        fix a
        assume a ∈ (lexLessRestricted Vbl)^+
        with ⟨lexLessRestricted Vbl ⊆ lexLess⟩
        show a ∈ lexLess`
          using trancl mono[of a lexLessRestricted Vbl lexLess]
          by blast
      qed
      with ⟨(x, x) ∈ (lexLessRestricted Vbl)^+⟩
      have (x, x) ∈ lexLess`
        by auto
    moreover
    have trans lexLess
      using translexLess
    .
    hence lexLess` = lexLess
      by (rule trancl id)
  ultimately
  have (x, x) ∈ lexLess
    by auto
    with irreflexiveLexLess[of x]
  have False
    by simp
  }
thus thesis
by auto
qed
qed

\textit{lexLessRestricted} is also transitive.

\textbf{lemma} \texttt{transLexLessRestricted}:  
shows \(\text{trans} (\text{lexLessRestricted Vbl})\)

\textbf{proof}–
\begin{itemize}
\item fix \(x::\text{LiteralTrail}\) and \(y::\text{LiteralTrail}\) and \(z::\text{LiteralTrail}\)
\item assume \((x, y) \in \text{lexLessRestricted Vbl}\)  
\((y, z) \in \text{lexLessRestricted Vbl}\)
\item hence \((x, z) \in \text{lexLessRestricted Vbl}\)
\item unfolding \texttt{lexLessRestricted-def}
\item using \texttt{translexLess}
\item unfolding \texttt{trans-def}
\item by auto
\end{itemize}
\}{
\item thus \texttt{thesis}
\item unfolding \texttt{trans-def}
\item by \texttt{blast}
\end{itemize}
qed

4.4.2 Conflict clause ordering

The ordering of conflict clauses is the multiset ordering induced by the ordering of elements in the trail. Since, resolution operator is defined so that it removes all occurrences of clashing literal, it is also necessary to remove duplicate literals before comparison.

\textbf{definition}
\[
multLess M = \text{inv-image} \ (\text{mult} \ (\text{precedesOrder} \ (\text{elements} \ M))) \ (\lambda \ x. \ \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ x)))
\]

The following lemma will help prove that application of the \textit{Explain} DPLL transition rule decreases the conflict clause in the \texttt{multLess} ordering.

\textbf{lemma} \texttt{multLessResolve}:
\begin{itemize}
\item assumes \(\text{opposite l el C and isReason reason l (elements M)}\)
\item shows \((\text{resolve C reason l (opposite l)}, C) \in \text{multLess M}\)
\end{itemize}

\textbf{proof}–
\begin{itemize}
\item let \(\ ?X = \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ C))\)
\item let \(\ ?Y = \text{multiset-of} \ (\text{remdups} \ (\text{oppositeLiteralList} \ (\text{resolve C reason (opposite l))))})\)
\item let \(\ ?ord = \text{precedesOrder} \ (\text{elements M})\)
\end{itemize}
have \( (?Y, ?X) \in \text{(mult1 ?ord)} \)

**proof** –

let \(?Z = \text{multiset-of (remdups (oppositeLiteralList (removeAll (opposite l) C)))}\)

let \(?W = \text{multiset-of (remdups (oppositeLiteralList (removeAll l (list-diff reason C))}})\)

let \(?a = l\)

from \(\langle \text{opposite l} \rangle \) el C;

have \(?X = ?Z + \{\#?a\}\)

using \(\text{removeAll-multiset[of remdups (oppositeLiteralList C) l]}\)

using \(\text{oppositeLiteralListRemove[of opposite l C]}\)

using \(\text{literalElListIffOppositeLiteralElOppositeLiteralList[of l oppositeLiteralList C]}\)

by auto (simp add: union-commute)

moreover

have \(?Y = ?Z + ?W\)

**proof** –

have \(\text{list-diff (oppositeLiteralList (removeAll l reason)) (oppositeLiteralList (removeAll (opposite l) C))} = \text{oppositeLiteralList (removeAll l (list-diff reason C))}\)

**proof** –

from \(\langle \text{isReason reason l (elements M)} \rangle\)

have \(\text{opposite l} \notin \text{set (removeAll l reason)}\)

unfolding \(\text{isReason-def}\)

by auto

hence \(\text{list-diff (removeAll l reason) (removeAll (opposite l) C)} = \text{list-diff (removeAll l reason) C}\)

using \(\text{listDiffRemoveAllNonMember[of opposite l removeAll l reason C]}\)

by simp

thus \(?\text{thesis}\)

unfolding \(\text{oppositeLiteralList-def}\)

using \(\text{listDiffMap[of opposite removeAll l reason removeAll (opposite l) C]}\)

by auto

qed

thus \(?\text{thesis}\)

unfolding \(\text{resolve-def}\)

using \(\text{remdupsAppendMultiSet[of oppositeLiteralList (removeAll (opposite l) C) oppositeLiteralList (removeAll l reason)]}\)

unfolding \(\text{oppositeLiteralList-def}\)

by auto

qed

moreover

have \(\forall \ b. \ b :\# ?W \longrightarrow (b, ?a) \in ?\text{ord}\)

**proof** –

\{ 

fix \(b\)
assume \( b : \# ?W \)
hence opposite \( b \in \text{set (removeAll \( l \) reason) } \)
proof -
from \( \langle b : \# ?W \rangle \)
have \( b \in \text{remdups (oppositeLiteralList (removeAll \( l \) (list-diff reason \( C \))))} \)
  by (auto simp add: set-count-greater-0)
hence opposite \( b \in \text{removeAll \( l \) (list-diff reason \( C \))} \)
using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite \( b \) removeAll \( l \) (list-diff reason \( C \))]
  by auto
hence opposite \( b \in \text{list-diff (removeAll \( l \) reason \( C \))} \)
  by simp
thus ?thesis
using listDiffIff[of opposite \( b \) removeAll \( l \) reason \( C \)]
  by simp
qed

with \( \langle \text{isReason reason \( l \) (elements \( M \))} \rangle \)
have precedes \( b \in \text{elements \( M \) \( b \neq l \)} \)
unfolding isReason-def
unfolding precedes-def
  by auto
hence \( (b, \# a) \in \# \text{ord} \)
unfolding precedesOrder-def
  by simp
thus ?thesis
unfolding mult1-def
  by auto
qed

ultimately
have \( \exists a M0 K. \# X = M0 + \{\# a\} \land \# Y = M0 + K \land (\forall b. b :\# K \longrightarrow (b, a) \in \# \text{ord}) \)
  by auto
thus ?thesis
unfolding mult1-def
  by auto
qed

hence \( \langle \# Y, \# X \rangle \in (\text{mult1 \# ord})^+ \)
  by simp
thus ?thesis
unfolding multLess-def
unfolding mult-def
unfolding inv-image-def
  by auto
qed

lemma multLessListDiff:
assumes \((a, b) \in \text{multLess M}\)
shows
\((\text{list-diff } a, b) \in \text{multLess } M\)

proof—
let \(?p\text{Ord} = \text{precedesOrder } (\text{elements } M)\)
let \(?f = \lambda l. \text{remdups } (\text{map opposite } l)\)
have trans \(?p\text{Ord}\)
  using \(\text{transPrecedesOrder[of elements } M\)\]
  by simp

have \((\text{multiset-of } (?f a), \text{multiset-of } (?f b)) \in \text{mult } ?p\text{Ord}\)
  using assms
  unfolding \(\text{multLess-def}\)
  unfolding \(\text{oppositeLiteralList-def}\)
  by simp
moreover
have \(\text{multiset-le } (\text{multiset-of } (\text{list-diff } (?f a) (?f x)))\)
  \((\text{multiset-of } (?f a))\)
  \(?p\text{Ord}\)
  using \(\langle \text{trans } ?p\text{Ord} \rangle\)
  using \(\text{multisetLeListDiff[of } ?p\text{Ord } ?f a ?f x\)\]
  by simp
ultimately
have \((\text{multiset-of } (\text{list-diff } (?f a) (?f x)), \text{multiset-of } (?f b)) \in \text{mult}\)
  \(?p\text{Ord}\)
  unfolding \(\text{multiset-le-def}\)
  unfolding \(\text{mult-def}\)
  by auto

thus \(?\text{thesis}\)
  unfolding \(\text{multLess-def}\)
  unfolding \(\text{oppositeLiteralList-def}\)
  by \((\text{simp add: listDiffMap remdupsListDiff})\)
qed

lemma \(\text{multLessRemdups:}\)
assumes
\((a, b) \in \text{multLess } M\)
shows
\((\text{remdups } a, \text{remdups } b) \in \text{multLess } M \land\)
\((\text{remdups } a, b) \in \text{multLess } M \land\)
\((a, \text{remdups } b) \in \text{multLess } M\)
proof—
{
  fix l
  have \(\text{remdups } (\text{map opposite } l) = \text{remdups } (\text{map opposite } (\text{remdups } l))\)
    by \((\text{induct } l)\) auto
}
thus \(?\text{thesis}\)
Now we show that \textit{multLess} is well-founded.

\textbf{lemma} \texttt{wfMultLess:}
\begin{itemize}
\item \texttt{shows} \texttt{wf (multLess M)}
\end{itemize}
\textbf{proof—}
\begin{itemize}
\item \texttt{have} \texttt{wf (precedesOrder (elements M))}
\item \texttt{by (simp add: wellFoundedPrecedesOrder)}
\item \texttt{hence} \texttt{wf (mult (precedesOrder (elements M)))}
\item \texttt{by (simp add: wf-mult)}
\item \texttt{thus} \texttt{?thesis}
\item \texttt{unfolding multLess-def}
\item \texttt{using wf-inv-image[of (mult (precedesOrder (elements M)))]}\texttt{]}\texttt{]}
\item \texttt{by auto}
\end{itemize}
qed

\subsection{ConflictFlag ordering}

A trivial ordering on Booleans. It will be used for the 	extit{Conflict} transition rule.

\textbf{definition}
\begin{itemize}
\item \texttt{boolLess = \{ (True, False) \}}
\end{itemize}

We show that it is well-founded

\textbf{lemma} \texttt{transBoolLess:}
\begin{itemize}
\item \texttt{shows} \texttt{trans boolLess}
\end{itemize}
\textbf{proof—}
\begin{itemize}
\item \texttt{fix} \texttt{x::bool and y::bool and z::bool}
\item \texttt{assume} \texttt{(x, y) \in boolLess}
\item \texttt{hence} \texttt{x = True y = False}
\item \texttt{unfolding boolLess-def}
\item \texttt{by auto}
\item \texttt{assume} \texttt{(y, z) \in boolLess}
\item \texttt{hence} \texttt{y = True z = False}
\item \texttt{unfolding boolLess-def}
\item \texttt{by auto}
\item \texttt{from} \texttt{(y = False) \& y = True}
\item \texttt{have} \texttt{False}
\item \texttt{by simp}
\item \texttt{hence} \texttt{(x, z) \in boolLess}
\item \texttt{by simp}
\end{itemize}
\texttt{}\texttt{thus }\texttt{?thesis}

196
unfolding trans-def
by blast
qed

lemma wfBoolLess:
  shows wf boolLess
proof (rule finite-acyclic-wf)
  show finite boolLess
    unfolding boolLess-def
    by simp
  next
    have boolLess⁺⁺ = boolLess
      using transBoolLess
      by simp
    thus acyclic boolLess
      unfolding boolLess-def
      unfolding acyclic-def
      by auto
qed

4.4.4 Formulae ordering

A partial ordering of formulae, based on a membership of a single
fixed clause. This ordering will be used for the Learn transtion
rule.

definition learnLess (C::Clause) == 
{(F1::Formula), (F2::Formula)).
  C el F1 ∧¬ C el F2}

We show that it is well founded

lemma wfLearnLess:
  fixes C::Clause
  shows wf (learnLess C)
unfolding wf-eq-minimal
proof−
  show ∀ Q F. F ∈ Q → (∃ Fmin∈Q. ∀ F'. (F', Fmin) ∈ learnLess
  C → F' ∉ Q)
  proof−
    { fix F::Formula and Q::Formula set
      assume F ∈ Q
      have ∃ Fmin∈Q. ∀ F'. (F', Fmin) ∈ learnLess C → F' ∉ Q
        proof (cases ∃ Fc ∈ Q. C el Fc)
          case True
            then obtain Fc where Fc ∈ Q C el Fc
            by auto
          have ∀ F'. (F', Fc) ∈ learnLess C → F' ∉ Q
            proof
              fix F'
  197
show \((F', Fc) \in \text{learnLess } C \rightarrow F' \notin Q\)
proof
  assume \((F', Fc) \in \text{learnLess } C\)
  hence \(\sim C el Fc\)
  unfolding learnLess-def
  by auto
  with \((C el Fc)\) have False
  by simp
  thus \(F' \notin Q\)
  by simp
qed
qed
with \((Fc \in Q)\)
show \(\neg \text{thesis}\)
by auto
next
case False
have \(\forall F'. (F', F) \in \text{learnLess } C \rightarrow F' \notin Q\)
proof
  fix \(F'\)
  show \((F', F) \in \text{learnLess } C \rightarrow F' \notin Q\)
  proof
    assume \((F', F) \in \text{learnLess } C\)
    hence \(C el F'\)
    unfolding learnLess-def
    by simp
    with False
    show \(F' \notin Q\)
    by auto
  qed
qed
qed
with \((F \in Q)\)
show \(\neg \text{thesis}\)
by auto
qed
}\)
thus \(\neg \text{thesis}\)
by auto
qed
qed

4.4.5 Properties of well-founded relations.

lemma wellFoundedEmbed:
  fixes rel :: \('a \times 'a\) set and rel' :: \('a \times 'a\) set
  assumes \(\forall x y. (x, y) \in \text{rel} \rightarrow (x, y) \in \text{rel'}\) and \(\text{wf rel'}\)
  shows \(\text{wf rel}\)
  unfolding \(\text{wf-eq-minimal}\)
  proof–
∀ Q x. x ∈ Q → (∃ zmin ∈ Q. ∀ z. (z, zmin) ∈ rel → z /∈ Q)

proof −
{  
  fix x::'a and Q::'a set  
  assume x ∈ Q  
  have ∃ zmin ∈ Q. ∀ z. (z, zmin) ∈ rel → z /∈ Q  
  proof−  
  from ⟨ wf rel' ⟩ ⟨ x ∈ Q ⟩  
  obtain zmin::'a  
  where zmin ∈ Q and ∀ z. (z, zmin) ∈ rel' → z /∈ Q  
  unfolding wf-eq-minimal  
  by auto  
  {  
    fix z::'a  
    assume (z, zmin) ∈ rel  
    have z /∈ Q  
    proof−  
      from ∀ x y. (x, y) ∈ rel → (x, y) ∈ rel' ⟨ (z, zmin) ∈ rel ⟩  
      have (z, zmin) ∈ rel'  
      by simp  
      with ∀ z. (z, zmin) ∈ rel' → z /∈ Q  
      show ?thesis  
      by simp  
    qed  
  }  
  with ⟨ zmin ∈ Q ⟩  
  show ?thesis  
  by auto  
  qed  
}  
thus ?thesis  
by auto  
qed  
qed

declned

5 BasicDPLL

theory BasicDPLL
imports SatSolverVerification
begin

This theory formalizes the transition rule system BasicDPLL which is based on the classical DPLL procedure, but does not use the PureLiteral rule.
5.1 Specification

The state of the procedure is uniquely determined by its trail.

\textbf{record} \textit{State} =
\begin{verbatim}
getM :: LiteralTrail
\end{verbatim}

Procedure checks the satisfiability of the formula F0 which does not change during the solving process. An external parameter is the set \textit{decisionVars} which are the variables that branching is performed on. Usually this set contains all variables of the formula F0, but that does not always have to be the case.

Now we define the transition rules of the system

\textbf{definition} \textit{appliedDecide} :: \textit{State} \Rightarrow \textit{State} \Rightarrow \textit{Variable set} \Rightarrow \textit{bool}
\begin{verbatim}
where
appliedDecide stateA stateB decisionVars ==
\exists \ l.
\quad (\text{var \ l}) \in \textit{decisionVars} \land
\quad \neg \ l \text{ el } (\text{elements } (\text{getM stateA})) \land
\quad \neg \ \text{opposite } \ l \text{ el } (\text{elements } (\text{getM stateA})) \land
\quad \text{getM stateB } = \text{getM stateA } \oplus [(\ l, \text{True})]
\end{verbatim}

\textbf{definition} \textit{applicableDecide} :: \textit{State} \Rightarrow \textit{Variable set} \Rightarrow \textit{bool}
\begin{verbatim}
where
applicableDecide state decisionVars == \exists \ state\. appliedDecide state decisionVars
\end{verbatim}

\textbf{definition} \textit{appliedUnitPropagate} :: \textit{State} \Rightarrow \textit{State} \Rightarrow \textit{Formula} \Rightarrow \textit{bool}
\begin{verbatim}
where
appliedUnitPropagate stateA stateB F0 ==
\exists \ (uc::\textit{Clause}) (ul::\textit{Literal}).
\quad uc \text{ el } F0 \land
\quad isUnitClause uc ul (\text{elements } (\text{getM stateA})) \land
\quad \text{getM stateB } = \text{getM stateA } \oplus [(\ul, \text{False})]
\end{verbatim}

\textbf{definition} \textit{applicableUnitPropagate} :: \textit{State} \Rightarrow \textit{Formula} \Rightarrow \textit{bool}
\begin{verbatim}
where
applicableUnitPropagate state F0 == \exists \ state\. appliedUnitPropagate state state' F0
\end{verbatim}

\textbf{definition} \textit{appliedBacktrack} :: \textit{State} \Rightarrow \textit{State} \Rightarrow \textit{Formula} \Rightarrow \textit{bool}
\begin{verbatim}
where
\end{verbatim}
appliedBacktrack stateA stateB F0 ==
  formulaFalse F0 (elements (getM stateA)) \land
  decisions (getM stateA) \neq [] \land

  getM stateB = prefixBeforeLastDecision (getM stateA) \&\& ([opposite (lastDecision (getM stateA)), False])

**definition**
applicableBacktrack :: State \rightarrow Formula \rightarrow bool

**where**
applicableBacktrack state F0 == \exists state'. appliedBacktrack state state' F0

Solving starts with the empty trail.

**definition**
isInitialState :: State \rightarrow Formula \rightarrow bool

**where**
isInitialState state F0 ==
  getM state = []

Transitions are preformed only by using one of the three given rules.

**definition**
transition stateA stateB F0 decisionVars ==
  appliedDecide stateA stateB decisionVars \lor
  appliedUnitPropagate stateA stateB F0 \lor
  appliedBacktrack stateA stateB F0

Transition relation is obtained by applying transition rules iteratively. It is defined using a reflexive-transitive closure.

**definition**
transitionRelation F0 decisionVars ==
  \{(stateA, stateB). transition stateA stateB F0 decisionVars\}^*~

Final state is one in which no rules apply

**definition**
isFinalState :: State \rightarrow Formula \rightarrow Variable set \rightarrow bool

**where**
isFinalState state F0 decisionVars == \neg (\exists state'. transition state state' F0 decisionVars)

The following several lemmas give conditions for applicability of different rules.

**lemma** applicableDecideCharacterization:

**fixes**
  stateA::State

**shows**
  applicableDecide stateA decisionVars =
\( \exists \; l.
\)
\[
(\text{var } l) \in \text{decisionVars} \land \\
\neg \; l \; \text{el} \; (\text{elements} \; (\text{getM stateA})) \land \\
\neg \; \text{opposite} \; l \; \text{el} \; (\text{elements} \; (\text{getM stateA}))
\]
(is \(?lhs = ?rhs\))

proof
assume \(?rhs\)
then obtain \(l\) where
\(*:(\text{var } l) \in \text{decisionVars} \neg \; l \; \text{el} \; (\text{elements} \; (\text{getM stateA})) \neg \; \text{opposite} \; l \; \text{el} \; (\text{elements} \; (\text{getM stateA}))\)
  unfolding applicableDecide-def
  by auto
let \(?stateB = stateA(\text{getM} := (\text{getM stateA}) \; @ \; [(l, \text{True})]) \)
from * have applicableDecide stateA \(?stateB\) decisionVars
  unfolding applicableDecide-def
  by auto
thus \(?lhs\)
  unfolding applicableDecide-def
  by auto

next
assume \(?lhs\)
then obtain \(stateB \; l\) where
\(*:(\text{var } l) \in \text{decisionVars} \neg \; l \; \text{el} \; (\text{elements} \; (\text{getM stateA})) \land \\
\neg \; \text{opposite} \; l \; \text{el} \; (\text{elements} \; (\text{getM stateA}))\)
  unfolding applicableDecide-def
  unfolding applicableDecide-def
  by auto
thus \(?rhs\)
  by auto
qed

lemma applicableUnitPropagateCharacterization:
fixes \(\text{stateA}::\text{State} \; \text{and} \; F0::\text{Formula}\)
shows applicableUnitPropagate stateA \(F0 =\)
\((\exists \; \text{uc}::\text{Clause} \; \text{ul}::\text{Literal}.\)
  \text{uc} \; \text{el} \; F0 \land \\
  \text{isUnitClause} \; \text{uc} \; \text{ul} \; (\text{elements} \; (\text{getM stateA})))\)
(is \(?lhs = ?rhs\))

proof
assume \(?rhs\)
then obtain \(ul \; uc\) where
\(*: \text{uc} \; \text{el} \; F0 \; \text{isUnitClause} \; \text{uc} \; \text{ul} \; (\text{elements} \; (\text{getM stateA}))\)
  unfolding applicableUnitPropagate-def
  by auto
let \(?stateB = stateA(\text{getM} := \text{getM stateA} \; @ \; [(ul, \text{False})]) \)
from * have applicableUnitPropagate stateA \(?stateB\) \(F0\)
  unfolding applicableUnitPropagate-def
  by auto
thus \(?lhs\)
unfolding applicableUnitPropagate-def
by auto

next
assume ?lhs
then obtain stateB uc ul
where ac el F0 isUnitClause uc ul (elements (getM stateA))
unfolding applicableUnitPropagate-def
unfolding appliedUnitPropagate-def
by auto
thus ?rhs
by auto

qed

lemma applicableBacktrackCharacterization:
fixes stateA::State
shows applicableBacktrack stateA F0 =
(formulaFalse F0 (elements (getM stateA)) ∧
decisions (getM stateA) ≠ [] ) (is ?lhs = ?rhs)

proof
assume ?rhs
hence *: formulaFalse F0 (elements (getM stateA)) decisions (getM stateA) ≠ []
by auto
let ?stateB = stateA[] getM := prefixBeforeLastDecision (getM stateA)
@ [(opposite (lastDecision (getM stateA)), False)]
from * have applicableBacktrack stateA ?stateB F0
unfolding applicableBacktrack-def
by auto
thus ?lhs
unfolding applicableBacktrack-def
by auto

next
assume ?lhs
then obtain stateB
where applicableBacktrack stateA stateB F0
unfolding applicableBacktrack-def
by auto

hence
formulaFalse F0 (elements (getM stateA))
decisions (getM stateA) ≠ []
getM stateB = prefixBeforeLastDecision (getM stateA) @ [(opposite
(lastDecision (getM stateA)), False)]
unfolding applicableBacktrack-def
by auto
thus ?rhs
by auto

qed

Final states are the ones where no rule is applicable.
lemma finalStateNonApplicable:
  fixes state::State
  shows isFinalState state F0 decisionVars =
    (¬ applicableDecide state decisionVars ∧
     ¬ applicableUnitPropagate state F0 ∧
     ¬ applicableBacktrack state F0)
unfolding isFinalState-def
unfolding transition-def
unfolding applicableDecide-def
unfolding applicableUnitPropagate-def
unfolding applicableBacktrack-def
by auto

5.2 Invariants

Invariants that are relevant for the rest of correctness proof.

definition invariantsHoldInState :: State ⇒ Formula ⇒ Variable set ⇒ bool
where
invariantsHoldInState state F0 decisionVars ==
  InvariantImpliedLiterals F0 (getM state) ∧
  InvariantVarsM (getM state) F0 decisionVars ∧
  InvariantConsistent (getM state) ∧
  InvariantUniq (getM state)

Invariants hold in initial states.

lemma invariantsHoldInInitialState:
  fixes state :: State and F0 :: Formula
  assumes isInitialState state F0
  shows invariantsHoldInState state F0 decisionVars
using assms
by (auto simp add:
  isInitialState-def
  invariantsHoldInState-def
  InvariantImpliedLiterals-def
  InvariantVarsM-def
  InvariantConsistent-def
  InvariantUniq-def
)

Valid transitions preserve invariants.

lemma transitionsPreserveInvariants:
  fixes stateA::State and stateB::State
  assumes transition stateA stateB F0 decisionVars and
  invariantsHoldInState stateA F0 decisionVars
  shows invariantsHoldInState stateB F0 decisionVars
proof—
from \langle\text{invariantsHoldInState}\ stateA\ F0\ \text{decisionVars}\rangle

have
  \text{InvariantImpliedLiterals}\ F0\ (\text{getM}\ stateA)\ \text{and}
  \text{InvariantVarsM}\ (\text{getM}\ stateA)\ F0\ \text{decisionVars}\ \text{and}
  \text{InvariantConsistent}\ (\text{getM}\ stateA)\ \text{and}
  \text{InvariantUniq}\ (\text{getM}\ stateA)

unfolding\ \text{invariantsHoldInState-def}
by\ \text{auto}

\{ 

assume\ \text{appliedDecide}\ stateA\ stateB\ \text{decisionVars}
then\ obtain\ l::\text{Literal}\ where
  (\text{var}\ \ l)\in\text{decisionVars}
  \not\ \text{literalTrue}\ l\ (\text{elements}\ (\text{getM}\ stateA))
  \not\ \text{literalFalse}\ l\ (\text{elements}\ (\text{getM}\ stateA))
  \text{getM}\ stateB = \text{getM}\ stateA\ \hat{[}(l,\ \text{True})]\}

unfolding\ \text{appliedDecide-def}
by\ \text{auto}

\{ 

from\ \not\ \text{literalTrue}\ l\ (\text{elements}\ (\text{getM}\ stateA))\} \not\ \text{literalFalse}\ l\ (\text{elements}\ (\text{getM}\ stateA))

have\ \ast:\ \text{var}\ \ l\not\in\text{vars}\ (\text{elements}\ (\text{getM}\ stateA))
  using\ \text{variableDefinedImpliesLiteralDefined}\ [\text{of}\ l\ \text{elements}\ (\text{getM}\ stateA)]
by\ \text{simp}

have\ \text{InvariantImpliedLiterals}\ F0\ (\text{getM}\ stateB)
  using
    \langle\text{getM}\ stateB = \text{getM}\ stateA\ \hat{[}(l,\ \text{True})]\}\rangle
    \langle\text{InvariantImpliedLiterals}\ F0\ (\text{getM}\ stateA)\rangle
    \langle\text{InvariantUniq}\ (\text{getM}\ stateA)\rangle
    \langle\text{var}\ \ l\not\in\text{vars}\ (\text{elements}\ (\text{getM}\ stateA))\rangle
    \text{InvariantImpliedLiteralsAfterDecide}\ [\text{of}\ F0\ \text{getM}\ stateA\ \ l\ \text{getM}\ stateB]
by\ \text{simp}
moreover 

have\ \text{InvariantVarsM}\ (\text{getM}\ stateB)\ F0\ \text{decisionVars}
  using\ \langle\text{getM}\ stateB = \text{getM}\ stateA\ \hat{[}(l,\ \text{True})]\}\rangle
    \langle\text{InvariantVarsM}\ (\text{getM}\ stateA)\rangle
    \langle\text{var}\ \ l\in\text{decisionVars}\rangle
    \text{InvariantVarsMAfterDecide}\ [\text{of}\ \text{getM}\ stateA\ F0\ \text{decisionVars}\ \ l\ \text{getM}\ stateB]
by\ \text{simp}
moreover 

have\ \text{InvariantConsistent}\ (\text{getM}\ stateB)
  using\ \langle\text{getM}\ stateB = \text{getM}\ stateA\ \hat{[}(l,\ \text{True})]\}\rangle
    \langle\text{InvariantConsistent}\ (\text{getM}\ stateA)\rangle
    \langle\text{var}\ \ l\not\in\text{vars}\ (\text{elements}\ (\text{getM}\ stateA))\rangle
    \text{InvariantConsistentAfterDecide}\ [\text{of}\ \text{getM}\ stateA\ \ l\ \text{getM}\ stateB]
by\ \text{simp}

\}
moreover
have InvariantUniq (getM stateB)
  using (getM stateB = getM stateA @ [(l, True)])
  ⟨InvariantUniq (getM stateA)
    ⟨var l \notin vars (elements (getM stateA))
    InvariantUniqAfterDecide[of getM stateA l getM stateB]
    by simp
  ⟩
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto
}
moreover
{
assume appliedUnitPropagate stateA stateB F0
then obtain uc::Clause and ul::Literal where
  uc el F0
  isUnitClause uc ul (elements (getM stateA))
  getM stateB = getM stateA @ [(ul, False)]
  unfolding appliedUnitPropagate-def
  by auto
from ⟨isUnitClause uc ul (elements (getM stateA))⟩
have ul el uc
  unfolding isUnitClause-def
  by simp
from ⟨uc el F0⟩
have formulaEntailsClause F0 uc
  by (simp add: formulaEntailsItsClauses)
have InvariantImpliedLiterals F0 (getM stateB)
  using
    ⟨InvariantImpliedLiterals F0 (getM stateA)
      ⟨formulaEntailsClause F0 uc⟩
      ⟨isUnitClause uc ul (elements (getM stateA))⟩
      ⟨getM stateB = getM stateA @ [(ul, False)]⟩
      InvariantImpliedLiteralsAfterUnitPropagate[of F0 getM stateA uc ul getM stateB]
    ⟩
    by simp
moreover
from ⟨ul el uc⟩ ⟨uc el F0⟩
have ul el F0
  by (auto simp add: literalElFormulaCharacterization)
hence var ul \in vars F0 \cup decisionVars
  using formulaContainsItsLiteralsVariable [of ul F0]
  by auto
have InvariantVarsM (getM stateB) F0 decisionVars
using (InvariantVarsM (getM stateA) F0 decisionVars)
(var ul ∈ vars F0 ∪ decisionVars)
⟨getM stateB = getM stateA @ [(ul, False)]; InvariantVarsMAfterUnitPropagate[of getM stateA F0 decisionVars ul getM stateB]
by simp
moreover
have InvariantConsistent (getM stateB)
using (InvariantConsistent (getM stateA));
⟨isUnitClause uc ul (elements (getM stateA));
⟨getM stateB = getM stateA @ [(ul, False)]; InvariantConsistentAfterUnitPropagate[of getM stateA uc ul getM stateB]
by simp
moreover
have InvariantUniq (getM stateB)
using (InvariantUniq (getM stateA));
⟨isUnitClause uc ul (elements (getM stateA));
⟨getM stateB = getM stateA @ [(ul, False)]; InvariantUniqAfterUnitPropagate[of getM stateA uc ul getM stateB]
by simp
ultimately
have ?thesis
unfolding invariantsHoldInState-def
by auto
}
moreover
{
assume appliedBacktrack stateA stateB F0
hence formulaFalse F0 (elements (getM stateA))
formulaFalse F0 (elements (getM stateA))
decisions (getM stateA) ≠ []
getM stateB = prefixBeforeLastDecision (getM stateA) @ [(opposite (lastDecision (getM stateA)), False)]
unfolding appliedBacktrack-def
by auto
have InvariantImpliedLiterals F0 (getM stateB)
using (InvariantImpliedLiterals F0 (getM stateA));
⟨formulaFalse F0 (elements (getM stateA));
⟨decisions (getM stateA) ≠ []];
⟨getM stateB = prefixBeforeLastDecision (getM stateA) @ [(opposite (lastDecision (getM stateA)), False)];
⟨InvariantUniq (getM stateA));
⟨InvariantConsistent (getM stateA))
InvariantImpliedLiteralsAfterBacktrack[of F0 getM stateA getM stateB]
by simp

207
moreover 
have InvariantVarsM \(\text{getM stateB}\) F0 decisionVars 
using \(\text{InvariantVarsM (getM stateA) F0 decisionVars}\) 
\langle \text{decisions (getM stateA) } \neq \text{[]}\rangle; 
\langle \text{getM stateB } = \text{prefixBeforeLastDecision (getM stateA)} \rangle; 
\langle \text{opposite (lastDecision (getM stateA)), False}\rangle; 
\langle \text{InvariantConsistentAfterBacktrack[of getM stateA F0 decisionVars getM stateB]} \rangle 
by simp

moreover 
have InvariantConsistent (getM stateB) 
using \(\text{InvariantConsistent (getM stateA)}\); 
\langle \text{InvariantUniq (getM stateA)}\rangle; 
\langle \text{decisions (getM stateA) } \neq \text{[]}\rangle; 
\langle \text{getM stateB } = \text{prefixBeforeLastDecision (getM stateA)} \rangle; 
\langle \text{opposite (lastDecision (getM stateA)), False}\rangle; 
\langle \text{InvariantConsistentAfterBacktrack[of getM stateA getM stateB]} \rangle 
by simp

moreover 
have InvariantUniq (getM stateB) 
using \(\text{InvariantConsistent (getM stateA)}\); 
\langle \text{InvariantUniq (getM stateA)}\rangle; 
\langle \text{decisions (getM stateA) } \neq \text{[]}\rangle; 
\langle \text{getM stateB } = \text{prefixBeforeLastDecision (getM stateA)} \rangle; 
\langle \text{opposite (lastDecision (getM stateA)), False}\rangle; 
\langle \text{InvariantUniqAfterBacktrack[of getM stateA getM stateB]} \rangle 
by simp

ultimately 
have \(?\)thesis 
unfolding invariantsHoldInState-def 
by auto

\}

ultimately 
show \(?\)thesis 
using \(\text{transition stateA stateB F0 decisionVars}\) 
unfolding transition-def 
by auto

qed

The consequence is that invariants hold in all valid runs.

lemma invariantsHoldInValidRuns:
  fixes F0 :: Formula and decisionVars :: Variable set
  assumes invariantsHoldInState stateA F0 decisionVars and 
  \((\text{stateA, stateB}) \in \text{transitionRelation F0 decisionVars}\) 
  shows invariantsHoldInState stateB F0 decisionVars
using assms
using transitionsPreserveInvariants
using rtrancl-induct[of stateA stateB] 
\{(\text{stateA, stateB}). transition stateA stateB F0 decisionVars\} \lambda x.
In the following text we will show that there are two kinds of states:

1. **UNSAT** states where \( \text{formulaFalse} \, F_0 \) (elements \((\text{getM state})\)) and \( \text{decisions} \) \((\text{getM state})\) = \[

2. **SAT** states where \( \neg \text{formulaFalse} \, F_0 \) \((\text{elements} \,(\text{getM state}))\) and \( \text{decisionVars} \subseteq \text{vars} \,(\text{elements} \,(\text{getM state}))\).

The soundness theorems claim that if **UNSAT** state is reached the formula is unsatisfiable and if **SAT** state is reached, the formula is satisfiable.

Completeness theorems claim that every final state is either **UNSAT** or **SAT**. A consequence of this and soundness theorems, is that if formula is unsatisfiable the solver will finish in an **UNSAT** state, and if the formula is satisfiable the solver will finish in a **SAT** state.

### 5.3 Soundness

**Theorem** soundnessForUNSAT:

- fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars} :: \text{Variable set} \)
- State and \( \text{state0} :: \text{State} \)
- assumes
  - isInitialState \( \text{state0} \, F_0 \) and
  - \((\text{state0}, \, \text{state}) \in \text{transitionRelation} \, F_0 \, \text{decisionVars}\)

\( \text{formulaFalse} \, F_0 \) \((\text{elements} \,(\text{getM state}))\)

\( \text{decisions} \) \((\text{getM state})\) = \[

\( \neg \) satisfiable \( F_0 \)
proof
from ⟨isInitialState state0 F0, ((state0, state) ∈ transitionRelation F0 decisionVars)⟩ have invariantsHoldInState state F0 decisionVars
using invariantsHoldInValidRunsFromInitialState
by simp
hence InvariantImpliedLiterals F0 (getM state)
unfolding invariantsHoldInState-def
by auto
with ⟨formulaFalse F0 (elements (getM state))⟩
⟨decisions (getM state) = []⟩
show ?thesis
using unsatReport[of F0 getM state F0]
unfolding InvariantEquivalent-def equivalentFormulae-def
by simp
qed

theorem soundnessForSAT:
fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State
assumes vars F0 ⊆ decisionVars and
isInitialState state0 F0 and
(state0, state) ∈ transitionRelation F0 decisionVars
¬ formulaFalse F0 (elements (getM state))
vars (elements (getM state)) ⊇ decisionVars
shows model (elements (getM state)) F0

proof
from ⟨isInitialState state0 F0, ((state0, state) ∈ transitionRelation F0 decisionVars)⟩ have invariantsHoldInState state F0 decisionVars
using invariantsHoldInValidRunsFromInitialState
by simp
hence InvariantConsistent (getM state)
unfolding invariantsHoldInState-def
by auto
with assms
show ?thesis
using satReport[of F0 decisionVars F0 getM state]
unfolding InvariantEquivalent-def equivalentFormulae-def InvariantVarsF-def
5.4 Termination

We now define a termination ordering on the set of states based on the \textit{lexLessRestricted} trail ordering. This ordering will be central in termination proof.

\begin{verbatim}
definition terminationLess (F0::Formula) decisionVars == {{(stateA::State), (stateB::State)}, (getM stateA, getM stateB) ∈ lexLessRestricted (vars F0 ∪ decisionVars)}
\end{verbatim}

We want to show that every valid transition decreases a state with respect to the constructed termination ordering. Therefore, we show that \textit{Decide}, \textit{UnitPropagate} and \textit{Backtrack} rule decrease the trail with respect to the restricted trail ordering. Invariants ensure that trails are indeed \textit{uniq}, \textit{consistent} and with finite variable sets.

\textbf{lemma} trailIsDecreasedByDecidedUnitPropagateAndBacktrack: 
\begin{verbatim}
fixes stateA::State and stateB::State
assumes invariantsHoldInState stateA F0 decisionVars and
appliedDecide stateA stateB decisionVars ⊓ appliedUnitPropagate stateA stateB F0 ⊓ appliedBacktrack stateA stateB F0
shows (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)
\end{verbatim}

\textbf{proof} –

\begin{verbatim}
from ⟨appliedDecide stateA stateB decisionVars ⊓ appliedUnitPropagate stateA stateB F0 ⊓ appliedBacktrack stateA stateB F0⟩
(invariantsHoldInState stateA F0 decisionVars)
have invariantsHoldInState stateB F0 decisionVars
  using transitionsPreserveInvariants
  unfolding transition-def
  by auto
from ⟨invariantsHoldInState stateA F0 decisionVars⟩
have ∗: uniq (elements (getM stateA)) consistent (elements (getM stateA)) vars (elements (getM stateA)) ⊆ vars F0 ∪ decisionVars
  unfolding invariantsHoldInState-def
  unfolding InvariantVarsM-def
  unfolding InvariantConsistent-def
  unfolding InvariantUniq-def
  by auto
from ⟨invariantsHoldInState stateB F0 decisionVars⟩
have ∗∗: uniq (elements (getM stateB)) consistent (elements (getM stateB)) vars (elements (getM stateB)) ⊆ vars F0 ∪ decisionVars
  unfolding invariantsHoldInState-def
  unfolding InvariantVarsM-def
  unfolding InvariantConsistent-def
  by auto
\end{verbatim}
unfolding InvariantUniq-def
by auto
{
 assume appliedDecide stateA stateB decisionVars
 hence (getM stateB, getM stateA) ∈ lexLess
 unfolding appliedDecide-def
 by (auto simp add: lexLessAppend)
 with **
 have ((getM stateB), (getM stateA)) ∈ lexLessRestricted (vars F0 ∪ decisionVars)
 unfolding lexLessRestricted-def
 by auto
}
moreover
{
 assume appliedUnitPropagate stateA stateB F0
 hence (getM stateB, getM stateA) ∈ lexLess
 unfolding appliedUnitPropagate-def
 by (auto simp add: lexLessAppend)
 with **
 have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)
 unfolding lexLessRestricted-def
 by auto
}
moreover
{
 assume appliedBacktrack stateA stateB F0
 hence
 formulaFalse F0 (elements (getM stateA))
 decisions (getM stateA) ≠ []
 getM stateB = prefixBeforeLastDecision (getM stateA) @ [(opposite (lastDecision (getM stateA)), False)]
 unfolding appliedBacktrack-def
 by auto
 hence (getM stateB, getM stateA) ∈ lexLess
 using (decisions (getM stateA) ≠ []):
 (getM stateB = prefixBeforeLastDecision (getM stateA) @ [(opposite (lastDecision (getM stateA)), False)])
 by (simp add: lexLessBacktrack)
 with **
 have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)
 unfolding lexLessRestricted-def
 by auto
}
ultimately
show ?thesis
using assms
Now we can show that every rule application decreases a state with respect to the constructed termination ordering.

**lemma** stateIsDecreasedByValidTransitions:

fixes stateA::State and stateB::State

assumes invariantsHoldInState stateA F0 decisionVars and transition stateA stateB F0 decisionVars

shows (stateB, stateA) ∈ terminationLess F0 decisionVars

**proof**–

from (transition stateA stateB F0 decisionVars)

have appliedDecide stateA stateB decisionVars ∨ appliedUnitPropagate stateA stateB F0 ∨ appliedBacktrack stateA stateB F0

unfolding transition-def

by simp

with (invariantsHoldInState stateA F0 decisionVars)

have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)

using trailIsDecreasedByDeciedUnitPropagateAndBacktrack

by simp

thus ?thesis

unfolding terminationLess-def

by simp

qed

The minimal states with respect to the termination ordering are final i.e., no further transition rules are applicable.

**definition** isMinimalState stateMin F0 decisionVars == (∀ state::State. (state, stateMin) ∉ terminationLess F0 decisionVars)

**lemma** minimalStatesAreFinal:

fixes stateA::State

assumes invariantsHoldInState state F0 decisionVars and isMinimalState state F0 decisionVars

shows isFinalState state F0 decisionVars

**proof**–

{ assume ¬ ?thesis

then obtain state'::State

where transition state state' F0 decisionVars

unfolding isFinalState-def

by auto

with (invariantsHoldInState state F0 decisionVars)

have (state', state) ∈ terminationLess F0 decisionVars

using stateIsDecreasedByValidTransitions[of state F0 decisionVars state']

unfolding transition-def

213
by auto
with (isMinimalState state F0 decisionVars)
have False
unfolding isMinimalState-def
by auto
}
thus ?thesis
by auto
qed

The following key lemma shows that the termination ordering is well founded.

lemma wfTerminationLess:
  fixes decisionVars :: Variable set and F0 :: Formula
  assumes finite decisionVars
  shows wf (terminationLess F0 decisionVars)
unfolding wf-eq-minimal
proof−
  { fix Q :: State set and state :: State
    assume state ∈ Q
    let ?Q1 = {M::LiteralTrail. ∃ state. state ∈ Q ∧ (getM state)
    = M}
    from (state ∈ Q)
    have getM state ∈ ?Q1
      by auto
    from (finite decisionVars)
    have finite (vars F0 ∪ decisionVars)
      using finiteVarsFormula[of F0]
      by simp
    hence wf (lexLessRestricted (vars F0 ∪ decisionVars))
      using wfLexLessRestricted[of vars F0 ∪ decisionVars]
      by simp
    with (getM state ∈ ?Q1)
    obtain Mmin where Mmin ∈ ?Q1 ∀ M′. (M′, Mmin) ∈ lexLess-Restricted (vars F0 ∪ decisionVars) −→ M′ /∈ ?Q1
      unfolding wf-eq-minimal
      apply (erule-tac x=?Q1 in allE)
      apply (erule-tac x=getM state in allE)
      by auto
    from (Mmin ∈ ?Q1) obtain stateMin
      where stateMin ∈ Q (getM stateMin) = Mmin
      by auto
    have ∀ state′. (state′, stateMin) ∈ terminationLess F0 decision-Vars −→ state′ /∈ Q
      proof
      214
Using the termination ordering we show that the transition relation is well founded on states reachable from initial state.

**theorem** `wfTransitionRelation`:

**fixes** `decisionVars :: Variable set` and `F0 :: Formula` and `state0 :: State`

**assumes** `finite decisionVars and isInitialState state0 F0`

**shows** `wf { (stateB, stateA). (state0, stateA) ∈ transitionRelation F0 decisionVars ∧ (transition stateA stateB F0 decisionVars) }`

**proof**

let `?rel = { (stateB, stateA). (state0, stateA) ∈ transitionRelation F0 decisionVars ∧ (transition stateA stateB F0 decisionVars) }`

let `?rel' = terminationLess F0 decisionVars`

have `∀ x y. (x, y) ∈ ?rel → (x, y) ∈ ?rel'`

**proof**

{
fix \( \text{stateA} :: \text{State} \) and \( \text{stateB} :: \text{State} \)
assume \((\text{stateB}, \text{stateA}) \in \mathcal{R}\)
hence \((\text{stateB}, \text{stateA}) \in \mathcal{R}'\)
using \(\text{isInitialState state0 F0}\).
using \(\text{invariantsHoldInValidRunsFromInitialState[of state0 F0 stateA decisionVars]}\)
using \(\text{stateIsDecreasedByValidTransitions[of stateA F0 decisionVars stateB]}\)
by simp 
}
thus \(\mathcal{R}\)
by simp
qed
moreover
have \(\text{wf } \mathcal{R}'\)
using \(\langle \text{finite decisionVars} \rangle\)
by \((\text{rule wfTerminationLess})\)
ultimately
show \(\mathcal{R}\)
using \(\text{wellFoundedEmbed[of } \mathcal{R} \mathcal{R}'\)\)
by simp
qed

We will now give two corollaries of the previous theorem. First is a weak termination result that shows that there is a terminating run from every initial state to the final one.

corollary
fixes \(\text{decisionVars} :: \text{Variable set} \) and \(\text{F0 :: Formula} \) and \(\text{state0 :: State} \)
assumes \(\text{finite decisionVars} \) and \(\text{isInitialState state0 F0} \)
shows \(\exists \text{ state. } (\text{state0}, \text{state}) \in \text{transitionRelation } F0 \text{ decisionVars} \) 
\(\land \text{isFinalState state F0 decisionVars} \)
proof−
{
  assume \(\neg \mathcal{R}\)
  let \(\mathcal{Q} = \{ (\text{state}. (\text{state0}, \text{state}) \in \text{transitionRelation } F0 \text{ decisionVars} ) \}
  let \(\mathcal{R}' = \{ (\text{stateB}, \text{stateA}). (\text{state0}, \text{stateA}) \in \text{transitionRelation } F0 \text{ decisionVars} \) 
  \(\land \text{transition stateA stateB F0 decisionVars} \}
  have \(\text{state0} \in \mathcal{Q}\)
  unfolding \(\text{transitionRelation-def}\)
  by simp
  hence \(\exists \text{ state. state } \in \mathcal{Q}\)
  by auto
  from assms
  have \(\text{wf } \mathcal{R}'\)
  using \(\text{wfTransitionRelation[of decisionVars state0 F0]}\)
by auto
hence $\forall Q. (\exists x. x \in Q) \rightarrow (\exists \text{stateMin} \in Q. \forall \text{state}. (\text{state}, \text{stateMin}) \in \text{?rel} \rightarrow \text{state} \notin Q)

unfolding wf-eq-minimal
by simp
hence $(\exists x. x \in Q) \rightarrow (\exists \text{stateMin} \in Q. \forall \text{state}. (\text{state}, \text{stateMin}) \in \text{?rel} \rightarrow \text{state} \notin Q)
by rule
with $(\exists \text{state}. \text{state} \in Q)$
have $\exists \text{stateMin} \in Q. \forall \text{state}. (\text{state}, \text{stateMin}) \in \text{?rel} \rightarrow \text{state} \notin Q
by simp
then obtain \text{stateMin}
where \text{stateMin} \in Q and $\forall \text{state}. (\text{state}, \text{stateMin}) \in \text{?rel} \rightarrow \text{state} \notin Q
by auto

from $(\text{stateMin} \in Q)$
have $(\text{state0}, \text{stateMin}) \in \text{transitionRelation F0 decisionVars}$
by simp
with $(\neg \text{thesis})$
have $\neg \text{isFinalState stateMin F0 decisionVars}$
by simp
then obtain state'::State
where transition stateMin state' F0 decisionVars
unfolding isFinalState-def
by auto
have $(\text{state'}, \text{stateMin}) \in \text{?rel}$
using $(\text{state0}, \text{stateMin}) \in \text{transitionRelation F0 decisionVars}$
$\langle\text{transition stateMin state'} F0 decisionVars\rangle$
by simp
with $\forall \text{state}. (\text{state}, \text{stateMin}) \in \text{?rel} \rightarrow \text{state} \notin Q$
have $\text{state'} \notin Q$
by force

moreover
from $(\text{state0}, \text{stateMin}) \in \text{transitionRelation F0 decisionVars}$
$\langle\text{transition stateMin state'} F0 decisionVars\rangle$
have $\text{state'} \in Q$
unfolding transitionRelation-def
using rtrancl-into-rtrancl[of state0 stateMin [(stateA, stateB), transition stateA stateB F0 decisionVars] state']
by simp
ultimately
have $\text{False}$
by simp
}
thus $\text{thesis}$
by auto
qed
Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

**corollary noInfiniteTransitionChains:**
- **fixes** $F_0 ::\text{Formula}$ and $\text{decisionVars} ::\text{Variable set}$
- **assumes** finite $\text{decisionVars}$
- **shows** $\neg (\exists Q ::\text{(State set)}. \exists \text{state0} \in Q. \isInitialState \text{state0} F_0 \land$

  \[(\forall \text{state} \in Q. (\exists \text{state'} \in Q. \text{transition state state'} F_0 \text{ decisionVars}))\]

**proof—**

{ 
  assume $\neg \?thesis$
  then obtain $Q ::\text{State set}$ and $\text{state0} ::\text{State}$
  where $\isInitialState \text{state0} F_0 \text{state0} \in Q$

  \[(\forall \text{state} \in Q. (\exists \text{state'} \in Q. \text{transition state state'} F_0 \text{ decisionVars}))\]
  
  by auto

  let $\?rel = \{(\text{stateB}, \text{stateA}). (\text{state0}, \text{stateA}) \in \text{transitionRelation} F_0 \text{ decisionVars} \land$

  \[\text{transition stateA stateB F0 decisionVars}\}\}

  from $\langle \text{finite decisionVars} \rangle \langle \isInitialState \text{state0} F_0 \rangle$

  have $\text{uf} \ ?rel$

  using $\text{ufTransitionRelation}$

  by simp

  hence $\text{wfmin}: \forall Q \ x. x \in Q \rightarrow$

  \[(\exists z \in Q. \forall y. (y, z) \in \?rel \rightarrow y \notin Q)\]

  unfolding $\text{wf-eq-minimal}$

  by simp

  let $\?Q = \{\text{state} \in Q. (\text{state0}, \text{state}) \in \text{transitionRelation} F_0 \text{ decisionVars}\}\}$

  from $\langle \text{state0} \in Q \rangle$

  have $\text{state0} \in \?Q$

  unfolding $\text{transitionRelation-def}$

  by simp

  with $\text{wfmin}$

  obtain $\text{stateMin} ::\text{State}$

  where $\text{stateMin} \in \?Q$ and $\forall y. (y, \text{stateMin}) \in \?rel \rightarrow y \notin \?Q$

  apply (erule-tac $x=\?Q \text{ in allE}$)

  by auto

  from $\langle \text{stateMin} \in \?Q \rangle$

  have $\text{stateMin} \in Q \ (\text{state0}, \text{stateMin}) \in \text{transitionRelation} F_0 \text{ decisionVars}$

218
by auto
with ($\forall \text{ state} \in Q. (\exists \text{ state}' \in Q. \text{ transition state state'} F0 \text{ decisionVars})$)

obtain state':State
where state' \in Q \text{ transition stateMin state'} F0 \text{ decisionVars}
by auto

with (state0, stateMin) \in \text{transitionRelation F0 decisionVars}
have (state', stateMin) \in ?rel
by simp
with $\forall y. (y, stateMin) \in ?rel \rightarrow y \notin ?Q$
have state' \notin ?Q
by force

from (state' \in Q) \wedge (state0, stateMin) \in \text{transitionRelation F0 decisionVars}
(transition stateMin state' F0 decisionVars)
have state' \in ?Q
unfolding transitionRelation-def
using rtrancl-into-rtrancl[of state0 stateMin \{(stateA, stateB). transition stateA stateB F0 decisionVars\} state']
by simp
with (state' \notin ?Q)
have False
by simp
}
thus ?thesis
by force
qed

5.5 Completeness

In this section we will first show that each final state is either SAT or UNSAT state.

lemma finalNonConflictState:
fixes state::State and FO :: Formula
assumes
\neg \text{applicableDecide state decisionVars}
shows \text{vars (elements (getM state))} \supseteq \text{decisionVars}

proof
fix x :: Variable
let ?l = Pos x
assume x \in decisionVars
hence var ?l = x \and var ?l \in decisionVars \and var (\text{opposite ?l}) \in decisionVars
by auto
with ($\neg \text{applicableDecide state decisionVars}$)
have literalTrue ?l (\text{elements (getM state)}) \or literalFalse ?l (\text{elements (getM state)})
unfolding applicableDecideCharacterization by force
with (var ?l = x)
show x ∈ vars (elements (getM state))
  using valuationContainsItsLiteralsVariable[of ?l elements (getM state)]
  using valuationContainsItsLiteralsVariable[of opposite ?l elements (getM state)]
  by auto
qed

lemma finalConflictingState:
  fixes state :: State
  assumes ¬ applicableBacktrack state F0 and
        formulaFalse F0 (elements (getM state))
  shows
    decisions (getM state) = []
  using assms
  using applicableBacktrackCharacterization
  by auto

lemma finalStateCharacterizationLemma:
  fixes state :: State
  assumes ¬ applicableDecide state decisionVars and
        ¬ applicableBacktrack state F0
  shows
    (¬ formulaFalse F0 (elements (getM state)) ∧ vars (elements (getM state)) ⊇ decisionVars) ∨
    (formulaFalse F0 (elements (getM state)) ∧ decisions (getM state) = [])
  proof (cases formulaFalse F0 (elements (getM state)))
    case True
    hence decisions (getM state) = []
    using assms
    using finalConflictingState
    by auto
    with True
    show ?thesis
    by simp
  next
    case False
    hence vars (elements (getM state)) ⊇ decisionVars
    using assms
    using finalNonConflictState
    by auto
    with False

  qed
show \( \textsf{thesis} \)
  by simp
qed

**theorem** finalStateCharacterization:
  fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars} :: \text{Variable set} \) and \( \text{state0} :: \text{State} \) and \( \text{state} :: \text{State} \)
  assumes
  isInitialState \( \text{state0} F_0 \) and
  \( (\text{state0}, \text{state}) \in \text{transitionRelation} F_0 \text{ decisionVars} \) and
  isFinalState \( \text{state} F_0 \text{ decisionVars} \)
  shows
  \( (\neg \text{formulaFalse} F_0 (\text{elements} (\text{getM state}))) \land \text{vars} (\text{elements} (\text{getM state})) \supsete \text{decisionVars}) \lor \)
  \( (\text{formulaFalse} F_0 (\text{elements} (\text{getM state}))) \land \text{decisions} (\text{getM state}) = []) \)

**proof**–
  from \( \langle \text{isFinalState} \text{ state} F_0 \text{ decisionVars} \rangle \)
  have **:
    \( \neg \text{applicableBacktrack} \text{ state} F_0 \)
    \( \neg \text{applicableDecide} \text{ state} \text{ decisionVars} \)
  unfolding finalStateNonApplicable
  by auto

  thus \( \textsf{thesis} \)
  using finalStateCharacterizationLemma[of \text{state} decisionVars]
  by simp
qed

Completeness theorems are easy consequences of this characterization and soundness.

**theorem** completenessForSAT:
  fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars} :: \text{Variable set} \) and \( \text{state0} :: \text{State} \) and \( \text{state} :: \text{State} \)
  assumes
  satisfiable \( F_0 \) and
  isInitialState \( \text{state0} F_0 \) and
  \( (\text{state0}, \text{state}) \in \text{transitionRelation} F_0 \text{ decisionVars} \) and
  isFinalState \( \text{state} F_0 \text{ decisionVars} \)
  shows \( \neg \text{formulaFalse} F_0 (\text{elements} (\text{getM state}))) \land \text{vars} (\text{elements} (\text{getM state})) \supsete \text{decisionVars} \)

**proof**–
  from \( \text{assms} \)
\textbf{theorem} completenessForUNSAT:
\begin{align*}
\text{fixes } & F_0 :: \text{Formula and } \text{decisionVars :: Variable set and } state_0 :: \text{State and } state :: \text{State} \\
\text{assumes } & \text{vars } F_0 \subseteq \text{decisionVars and } \\
\text{isInitialState } & state_0 F_0 \text{ and } \\
\text{isFinalState } & state F_0 \text{ decisionVars} \\
\text{shows } & \text{formulaFalse } F_0 \text{ (elements (getM state)) } \land \text{decisions (getM state)} = [] \\
\text{proof} - & \\
\text{from } & \text{assms} \\
\text{have *: } & (\neg \text{formulaFalse } F_0 \text{ (elements (getM state)) } \land \text{vars (elements (getM state)) } \supseteq \text{decisionVars} ) \lor \\
& (\text{formulaFalse } F_0 \text{ (elements (getM state)) } \land \text{decisions (getM state)} = [] \\
& \text{using } \text{finalStateCharacterization[af state0 } F_0 \text{ state decisionVars]} \\
& \text{by } \text{auto} \\
\end{align*}
{ 
    assume \( \neg \text{formulaFalse} \text{F0 \ (elements (getM \ state))} \) 
    with * 
    have \( \neg \text{formulaFalse} \text{F0 \ (elements (getM \ state)) \ vars \ (elements \ (getM \ state)) \supseteq \ decisionVars} \) 
    by auto 
    with assms 
    have satisfiable \text{F0} 
        using soundnessForSAT[of \text{F0 \ decisionVars \ state0 \ state}] 
        unfolding satisfiable-def 
    by auto 
    with \( \neg \text{satisfiable F0}: \) 
    have False 
    by simp 
} 
with * show \text{thesis} 
by auto 
qed 

\textbf{theorem} partialCorrectness: 
\textbf{fixes} \text{F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State} 
\textbf{assumes} 
vars \text{F0 \ \subseteq \ decisionVars \ and} 

isInitialState state0 \text{F0 and} 
\text{(state0, state) \in transitionRelation \text{F0 \ decisionVars and}} 
isFinalState state \text{F0 \ decisionVars} 

\textbf{shows} 
satisfiable \text{F0} = (\neg \text{formulaFalse} \text{F0 \ (elements (getM \ state))}) 

using assms 
using completenessForUNSAT[of \text{F0 \ decisionVars \ state0 \ state}] 
using completenessForSAT[of \text{F0 \ state0 \ state \ decisionVars}] 
by auto 

end 

6 Transition system of Nieuwenhuis, Oliveras and Tinelli. 

\textbf{theory} NieuwenhuisOliverasTinelli 
\textbf{imports} SatSolverVerification 
\textbf{begin} 

This theory formalizes the transition rule system given by Nieuwen-
huis et al. in [3]

6.1 Specification

record State =
  getF :: Formula
  getM :: LiteralTrail

definition appliedDecide :: State ⇒ State ⇒ Variable set ⇒ bool
where
  appliedDecide stateA stateB decisionVars ==
  ∃ l. (var l) ∈ decisionVars ∧
   ¬ l el (elements (getM stateA)) ∧
   ¬ opposite l el (elements (getM stateA)) ∧

  getF stateB = getF stateA ∧
  getM stateB = getM stateA @ [(l, True)]

definition applicableDecide :: State ⇒ Variable set ⇒ bool
where
  applicableDecide state decisionVars == ∃ state′. appliedDecide state state′ decisionVars

definition applicableUnitPropagate :: State ⇒ State ⇒ bool
where
  applicableUnitPropagate stateA stateB ==
  ∃ (uc::Clause) (ul::Literal).
    uc el (getF stateA) ∧
    isUnitClause uc ul (elements (getM stateA)) ∧

  getF stateB = getF stateA ∧
  getM stateB = getM stateA @ [(ul, False)]

definition applicableUnitPropagate :: State ⇒ bool
where
  applicableUnitPropagate state == ∃ state′. applicableUnitPropagate state state′

definition appliedBackjump :: State ⇒ State ⇒ bool
where
  appliedBackjump stateA stateB ==
  ∃ bc bl level.
    isUnitClause bc bl (elements (prefixToLevel level (getM stateA)))
∧
  \begin{align*}
    \text{formulaEntailsClause} \left( \text{getF stateA} \right) \text{ bc} & \land \\
    \text{var bl} \in \text{vars} \left( \text{getF stateA} \right) \cup \text{vars} \left( \text{elements} \left( \text{getM stateA} \right) \right) & \land \\
    0 \leq \text{level} & \land \text{level} < \left( \text{currentLevel} \left( \text{getM stateA} \right) \right) \land \\
  \end{align*}

  \begin{align*}
    \text{getF stateB} &= \text{getF stateA} \land \\
    \text{getM stateB} &= \text{prefixToLevel level} \left( \text{getM stateA} \right) \oplus \left[ \left( \text{bl}, \text{False} \right) \right]
  \end{align*}

\textbf{definition}
\textit{applicableBackjump} :: \textit{State} \Rightarrow \textit{bool}
\textbf{where}
\textit{applicableBackjump state} == \exists \textit{state}'. \textit{appliedBackjump state state'}

\textbf{definition}
\textit{appliedLearn} :: \textit{State} \Rightarrow \textit{State} \Rightarrow \textit{bool}
\textbf{where}
\textit{appliedLearn stateA stateB} ==
  \exists \textit{c}.
    \begin{align*}
      & \text{formulaEntailsClause} \left( \text{getF stateA} \right) \text{ c} \land \\
      & \text{vars} \textit{c} \subseteq \text{vars} \left( \text{getF stateA} \right) \cup \text{vars} \left( \text{elements} \left( \text{getM stateA} \right) \right) \land \\
      \end{align*}

  \begin{align*}
    \text{getF stateB} &= \text{getF stateA} \oplus \left[ \textit{c} \right] \land \\
    \text{getM stateB} &= \text{getM stateA}
  \end{align*}

\textbf{definition}
\textit{applicableLearn} :: \textit{State} \Rightarrow \textit{bool}
\textbf{where}
\textit{applicableLearn state} == (\exists \textit{state}'. \textit{appliedLearn state state'})

Solving starts with the initial formula and the empty trail.

\textbf{definition}
\textit{isInitialState} :: \textit{State} \Rightarrow \textit{Formula} \Rightarrow \textit{bool}
\textbf{where}
\textit{isInitialState state F0} ==
  \begin{align*}
    \text{getF state} &= \text{F0} \land \\
    \text{getM state} &= []
  \end{align*}

Transitions are performed only by using given rules.

\textbf{definition}
\textit{transition stateA stateB decisionVars} ==
  \begin{align*}
    \text{appliedDecide} \quad & \text{stateA stateB decisionVars} \lor \\
    \text{appliedUnitPropagate} \quad & \text{stateA stateB} \lor \\
    \text{appliedLearn} \quad & \text{stateA stateB} \lor \\
    \text{appliedBackjump} \quad & \text{stateA stateB}
  \end{align*}

Transition relation is obtained by applying transition rules iter-
atatively. It is defined using a reflexive-transitive closure.

\[
\text{definition}
\text{transitionRelation decisionVars == } \{(\text{stateA, stateB}). \text{transition stateA stateB decisionVars}\}^*%
\]

Final state is one in which no rules apply

\[
\text{definition}
isFinalState :: \text{State } \Rightarrow \text{Variable set } \Rightarrow \text{bool}
\]

where

\[
isFinalState \text{ state decisionVars == } \neg (\exists \text{ state'}. \text{transition state state' decisionVars})
\]

The following several lemmas establish conditions for applicability of different rules.

\[
\text{lemma applicableDecideCharacterization:}
\text{fixes stateA::State}
\text{shows applicableDecide stateA decisionVars} =
(\exists l. (\text{var l}) \in \text{decisionVars} \land
\neg l \in \text{elements (getM stateA)} \land
\neg \text{opposite l el (elements (getM stateA))})
\]

\[
(\text{is } ?\text{lhs = } ?\text{rhs})
\]

\[
\text{proof}
\text{assume } ?\text{rhs}
\text{then obtain } l \text{ where}
*: (\text{var l}) \in \text{decisionVars} \neg l \in \text{elements (getM stateA)} \neg \text{opposite l el (elements (getM stateA))}
\text{unfolding applicableDecide-def}
\text{by auto}
\text{let } ?\text{stateB} = \text{stateA}[\text{getM := (getM stateA)} @ [(l, True)] ]
\text{from } * \text{ have applicableDecide stateA } ?\text{stateB decisionVars}
\text{unfolding applicableDecide-def}
\text{by auto}
\text{thus } ?\text{lhs}
\text{unfolding applicableDecide-def}
\text{by auto}
\]

\[
\text{next}
\text{assume } ?\text{lhs}
\text{then obtain } \text{stateB l}
\text{where (var l) \in decisionVars \neg l el (elements (getM stateA))}
\neg \text{opposite l el (elements (getM stateA))}
\text{unfolding applicableDecide-def}
\text{unfolding applicableDecide-def}
\text{by auto}
\text{thus } ?\text{rhs}
\text{by auto}
\]

\text{qed}
lemma applicableUnitPropagateCharacterization:
fixes stateA::State and F0::Formula
shows applicableUnitPropagate stateA =
(∃ uc::Clause ul::Literal.
  uc el (getF stateA) ∧
  isUnitClause uc ul (elements (getM stateA)))
(is ?lhs = ?rhs)
proof
  assume ?rhs
  then obtain ul uc
    where *: uc el (getF stateA) isUnitClause uc ul (elements (getM stateA))
      unfolding applicableUnitPropagate-def
      by auto
  let ?stateB = stateA( getM := getM stateA @ [(ul, False)] )
  from * have applicableUnitPropagate stateA ?stateB
    unfolding applicableUnitPropagate-def
    by auto
  thus ?lhs
    unfolding applicableUnitPropagate-def
    by auto
next
  assume ?lhs
  then obtain stateB uc ul
    where uc el (getF stateA) isUnitClause uc ul (elements (getM stateA))
      unfolding applicableUnitPropagate-def
      unfolding applicableUnitPropagate-def
      by auto
  thus ?rhs
    by auto
qed

lemma applicableBackjumpCharacterization:
fixes stateA::State
shows applicableBackjump stateA =
(∃ bc bl level.
  isUnitClause bc bl (elements (prefixToLevel level (getM stateA)))
  ∧
  formulaEntailsClause (getF stateA) bc ∧
  var bl ∈ vars (getF stateA) ∪ vars (elements (getM stateA)) ∧
  0 ≤ level ∧ level < (currentLevel (getM stateA)) (is ?lhs = ?rhs)
)
proof
  assume ?rhs
  then obtain bc bl level
    where *: isUnitClause bc bl (elements (prefixToLevel level (getM stateA)))
      formulaEntailsClause (getF stateA) bc

227
var bl ∈ vars (getF stateA) ∪ vars (elements (getM stateA))
0 ≤ level level < (currentLevel (getM stateA))
unfolding applicableBackjump-def
by auto
let ?stateB = stateA[ getM := prefixToLevel (getM stateA) @ [(bl, False)]]
from * have appliedBackjump stateA ?stateB
  unfolding appliedBackjump-def
by auto
thus ?lhs
  unfolding applicableBackjump-def
by auto

next
assume ?lhs
then obtain stateB
  where appliedBackjump stateA stateB
  unfolding applicableBackjump-def
by auto
then obtain bc bl level
  where isUnitClause bc bl (elements (prefixToLevel (getM stateA))
formulAEntailsClause (getF stateA) bc
var bl ∈ vars (getF stateA) ∪ vars (elements (getM stateA))
getF stateB = getF stateA
getM stateB = prefixToLevel (getM stateA) @ [(bl, False)]
0 ≤ level level < (currentLevel (getM stateA))
unfolding appliedBackjump-def
by auto
thus ?rhs
by auto
qed

lemma applicableLearnCharacterization:
  fixes stateA::State
  shows applicableLearn stateA =
    (∃ c. formulAEntailsClause (getF stateA) c ∧
     vars c ⊆ vars (getF stateA) ∪ vars (elements (getM stateA)))
(is ?lhs = ?rhs)
proof
  assume ?rhs
  then obtain c where
    *: formulAEntailsClause (getF stateA) c
    vars c ⊆ vars (getF stateA) ∪ vars (elements (getM stateA))
    unfolding applicableLearn-def
    by auto
  let ?stateB = stateA[ getF := getF stateA @ [c]]
  from * have appliedLearn stateA ?stateB
    unfolding appliedLearn-def
    by auto
thus ?lhs
unfolding applicableLearn-def
by auto

next
assume ?lhs
then obtain c stateB
where
  formulaEntailsClause (getF stateA) c
  vars c ⊆ vars (getF stateA) ∪ vars (elements (getM stateA))
unfolding applicableLearn-def
unfolding appliedLearn-def
by auto
thus ?rhs
by auto
qed

Final states are the ones where no rule is applicable.

lemma finalStateNonApplicable:
  fixes state::State
  shows isFinalState state decisionVars =
    (¬ applicableDecide state decisionVars ∧
     ¬ applicableUnitPropagate state ∧
     ¬ applicableBackjump state ∧
     ¬ applicableLearn state)
unfolding isFinalState-def
unfolding transition-def
unfolding applicableDecide-def
unfolding applicableUnitPropagate-def
unfolding applicableBackjump-def
unfolding applicableLearn-def
by auto

6.2 Invariants

Invariants that are relevant for the rest of correctness proof.

definition invariantsHoldInState :: State ⇒ Formula ⇒ Variable set ⇒ bool
  where
  invariantsHoldInState state F0 decisionVars ==
    InvariantImpliedLiterals (getF state) (getM state) ∧
    InvariantVarsM (getM state) F0 decisionVars ∧
    InvariantVarsF (getF state) F0 decisionVars ∧
    InvariantConsistent (getM state) ∧
    InvariantUniq (getM state) ∧
    InvariantEquivalent F0 (getF state)

Invariants hold in initial states.

lemma invariantsHoldInInitialState:
fixes state :: State and F0 :: Formula
assumes isInitialState state F0
shows invariantsHoldInState state F0 decisionVars
using assms
by (auto simp add: isInitialState-def invariantsHoldInState-def InvariantImpliedLiterals-def InvariantVarsM-def InvariantVarsF-def InvariantConsistent-def InvariantUniq-def InvariantEquivalent-def equivalentFormulae-def)

Valid transitions preserve invariants.

lemma transitionsPreserveInvariants:
fixes stateA :: State and stateB :: State
assumes transition stateA stateB decisionVars and invariantsHoldInState stateA F0 decisionVars
shows invariantsHoldInState stateB F0 decisionVars
proof –
from ⟨invariantsHoldInState stateA F0 decisionVars⟩ have
InvariantImpliedLiterals (getF stateA) (getM stateA) and
InvariantVarsM (getM stateA) F0 decisionVars and
InvariantVarsF (getF stateA) F0 decisionVars and
InvariantConsistent (getM stateA) and
InvariantUniq (getM stateA) and
InvariantEquivalent F0 (getF stateA)
unfolding invariantsHoldInState-def
by auto
{
assume appliedDecide stateA stateB decisionVars
then obtain l :: Literal where
(var l) ∈ decisionVars
¬ literalTrue l (elements (getM stateA))
¬ literalFalse l (elements (getM stateA))
geM stateB = getM stateA @ [(l, True)]
geF stateB = getF stateA
unfolding appliedDecide-def
by auto

from (∼ literalTrue l (elements (getM stateA))) (∼ literalFalse l (elements (getM stateA)))
have ∗: var l ∉ vars (elements (getM stateA))
using variableDefinedImpliesLiteralDefined[of l elements (getM stateA)]
by simp

230
have InvariantImpliedLiterals (getF stateB) (getM stateB)
  using (getF stateB = getF stateA)
  ⟨getM stateB = getM stateA @ [(l, True)]⟩
  ⟨InvariantImpliedLiterals (getF stateA) (getM stateA)⟩
  ⟨InvariantUniq (getM stateA)⟩
  ⟨var l /∈ vars (elements (getM stateA))⟩
  InvariantImpliedLiteralsAfterDecide[of getF stateA getM stateA l getM stateB]
  by simp
moreover
have InvariantVarsM (getM stateB) F0 decisionVars
  using (getM stateB = getM stateA @ [(l, True)])
  ⟨InvariantVarsM (getM stateA) F0 decisionVars⟩
  ⟨var l ∈ decisionVars⟩
  InvariantVarsMAfterDecide[of getM stateA F0 decisionVars l getM stateB]
  by simp
moreover
have InvariantVarsF (getF stateB) F0 decisionVars
  using (getF stateB = getF stateA)
  ⟨InvariantVarsF (getF stateA) F0 decisionVars⟩
  by simp
moreover
have InvariantConsistent (getM stateB)
  using (getM stateB = getM stateA @ [(l, True)])
  ⟨InvariantConsistent (getM stateA)⟩
  ⟨var l /∈ vars (elements (getM stateA))⟩
  InvariantConsistentAfterDecide[of getM stateA l getM stateB]
  by simp
moreover
have InvariantUniq (getM stateB)
  using (getM stateB = getM stateA @ [(l, True)])
  ⟨InvariantUniq (getM stateA)⟩
  ⟨var l /∈ vars (elements (getM stateA))⟩
  InvariantUniqAfterDecide[of getM stateA l getM stateB]
  by simp
moreover
have InvariantEquivalent F0 (getF stateB)
  using (getF stateB = getF stateA)
  ⟨InvariantEquivalent F0 (getF stateA)⟩
  by simp
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto
}
moreover
{
assume appliedUnitPropagate stateA stateB
then obtain uc::Clause and ul::Literal where
   uc el (getF stateA)
isUnitClause uc ul (elements (getM stateA))
getF stateB = getF stateA
getM stateB = getM stateA @ [(ul, False)]

unfolding appliedUnitPropagate-def
by auto

from ⟨isUnitClause uc ul (elements (getM stateA))⟩
have ul el uc
   unfolding isUnitClause-def
   by simp

from ⟨uc el (getF stateA)⟩
have formulaEntailsClause (getF stateA) uc
   by (simp add: formulaEntailsItsClauses)

have InvariantImpliedLiterals (getF stateB) (getM stateB)
   using (getF stateB = getF stateA)
   ⟨InvariantImpliedLiterals (getF stateA) (getM stateA)⟩
   ⟨formulaEntailsClause (getF stateA) uc⟩
   ⟨isUnitClause uc ul (elements (getM stateA))⟩
   ⟨getM stateB = getM stateA @ [(ul, False)]⟩
   InvariantImpliedLiteralsAfterUnitPropagate[of getF stateA getM stateA uc ul getM stateB]
   by simp
moreover
from ⟨ul el uc⟩ ⟨uc el (getF stateA)⟩
have ul el (getF stateA)
   by (auto simp add: literalElFormulaCharacterization)
with ⟨InvariantVarsF (getF stateA) F0 decisionVars⟩
have var ul ∈ vars F0 ∪ decisionVars
   unfolding InvariantContainsItsLiteralsVariable [of ul getF stateA]
   by auto

have InvariantVarsM (getM stateB) F0 decisionVars
   using ⟨InvariantVarsM (getM stateA) F0 decisionVars⟩
   ⟨var ul ∈ vars F0 ∪ decisionVars⟩
   ⟨getM stateB = getM stateA @ [(ul, False)]⟩
   InvariantVarsMAfterUnitPropagate[of getM stateA F0 decisionVars ul getM stateB]
   by simp
moreover
have InvariantVarsF (getF stateB) F0 decisionVars
   using (getF stateB = getF stateA)
   ⟨InvariantVarsF (getF stateA) F0 decisionVars⟩

232
by simp
moreover
have InvariantConsistent (getM stateB)
  using InvariantConsistent (getM stateA):
  ⟨InvariantConsistent (getM stateA)⟩
  ⟨isUnitClause uc ul (elements (getM stateA))⟩
  ⟨getM stateB = getM stateA @ [(ul, False)]⟩
  InvariantConsistentAfterUnitPropagate [of getM stateA uc ul]
  ⟨getM stateB = getM stateA⟩
  by simp
moreover
have InvariantUniq (getM stateB)
  using InvariantUniq (getM stateA):
  ⟨isUnitClause uc ul (elements (getM stateA))⟩
  ⟨getM stateB = getM stateA @ [(ul, False)]⟩
  InvariantUniqAfterUnitPropagate [of getM stateA uc ul]
  ⟨getM stateB = getM stateA⟩
  by simp
moreover
have InvariantEquivalent F0 (getF stateB)
  using (getF stateB = getF stateA):
  ⟨InvariantEquivalent F0 (getF stateA)⟩
  by simp
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto
} moreover
{
  assume appliedLearn stateA stateB
  then obtain c::Clause where
  formulaEntailsClause (getF stateA) c
  vars c ⊆ vars (getF stateA) ∪ vars (elements (getM stateA))
  getF stateB = getF stateA @ [c]
  getM stateB = getM stateA
  unfolding appliedLearn-def
  by auto

  have InvariantImpliedLiterals (getF stateB) (getM stateB)
    using
    ⟨InvariantImpliedLiterals (getF stateA) (getM stateA)⟩
    ⟨getF stateB = getF stateA @ [c]⟩
    ⟨getM stateB = getM stateA⟩
    InvariantImpliedLiteralsAfterLearn [of getF stateA getM stateA]
    by simp
  moreover
  have InvariantVarsM (getM stateB) F0 decisionVars
    using

\langle \text{InvariantVarsM} (\text{getM stateA}) \ F0 \ \text{decisionVars} \rangle
\langle \text{getM stateB} = \text{getM stateA} \rangle
\text{by simp}

moreover
from \langle \text{vars c} \subseteq \text{vars} (\text{getF stateA}) \cup \text{vars} (\text{elements (getM stateA)}) \rangle
\langle \text{InvariantVarsM} (\text{getM stateA}) \ F0 \ \text{decisionVars} \rangle
\langle \text{InvariantVarsF} (\text{getF stateA}) \ F0 \ \text{decisionVars} \rangle
have \text{vars c} \subseteq \text{vars} F0 \cup \text{decisionVars}
\text{unfolding InvariantVarsM-def}
\text{unfolding InvariantVarsF-def}
\text{by auto}

hence \text{InvariantVarsF} (\text{getF stateB}) \ F0 \ \text{decisionVars}
using \langle \text{InvariantVarsF} (\text{getF stateA}) \ F0 \ \text{decisionVars} \rangle
\langle \text{getF stateB} = \text{getF stateA} @ \text{[c]} \rangle
using \text{varsAppendFormulae [of getF stateA [c]]}
\text{unfolding InvariantVarsF-def}
\text{by simp}

moreover
have \text{InvariantConsistent} (\text{getM stateB})
using \langle \text{InvariantConsistent} (\text{getM stateA}) \rangle
\langle \text{getM stateB} = \text{getM stateA} \rangle
\text{by simp}

moreover
have \text{InvariantUniq} (\text{getM stateB})
using \langle \text{InvariantUniq} (\text{getM stateA}) \rangle
\langle \text{getM stateB} = \text{getM stateA} \rangle
\text{by simp}

moreover
have \text{InvariantEquivalent} F0 (\text{getF stateB})
using
\langle \text{InvariantEquivalent} F0 (\text{getF stateA}) \rangle
\langle \text{formulaEntailsClause} (\text{getF stateA}) \ c \rangle
\langle \text{getF stateB} = \text{getF stateA} @ \text{[c]} \rangle
\langle \text{InvariantEquivalentAfterLearn}[\text{of F0 getF stateA c getF stateB}] \rangle
\text{by simp}

ultimately
have \text{thesis}
\text{unfolding invariantsHoldInState-def}
\text{by simp}

}

moreover
{
\text{assume appliedBackjump stateA stateB}
\text{then obtain bc::Clause and bl::Literal and level::nat}
\text{where}
\text{isUnitClause bc bl (elements (prefixToLevel level (getM stateA)))}
\text{formulaEntailsClause (getF stateA) bc}
\text{var bl \in vars (getF stateA) \cup vars (elements (getM stateA))}
\text{getF stateB = getF stateA}

}
getM stateB = prefixToLevel level (getM stateA) @ [(bl, False)]
unfolding appliedBackjump-def
by auto

have isPrefix (prefixToLevel level (getM stateA)) (getM stateA)
by (simp add: isPrefixPrefixToLevel)

have InvariantImpliedLiterals (getF stateB) (getM stateB)
   using InvariantImpliedLiterals (getF stateA) (getM stateA)
   (isPrefix (prefixToLevel level (getM stateA)) (getM stateA))
   (isUnitClause bc bl (elements (prefixToLevel level (getM stateA))))
   (formulaEntailsClause (getF stateA) bc)
   (getF stateB = getF stateA)
   (getM stateB = prefixToLevel level (getM stateA) @ [(bl, False)])
   InvariantImpliedLiteralsAfterBackjump[of getF stateA getM stateA prefixToLevel level (getM stateA) bc bl getM stateB]
   by simp
moreover

from InvariantVarsF (getF stateA) F0 decisionVars
   using InvariantVarsM (getM stateA) F0 decisionVars
   (var bl ∈ vars (getF stateA) ∪ vars (elements (getM stateA)))]
have var bl ∈ vars F0 ∪ decisionVars
   unfolding InvariantVarsM-def
   unfolding InvariantVarsF-def
   by auto

have InvariantVarsM (getM stateB) F0 decisionVars
   using InvariantVarsM (getM stateA) F0 decisionVars
   (isPrefix (prefixToLevel level (getM stateA)) (getM stateA))
   (getM stateB = prefixToLevel level (getM stateA) @ [(bl, False)])
   (var bl ∈ vars F0 ∪ decisionVars)
   InvariantVarsMAfterBackjump[of getM stateA F0 decisionVars prefixToLevel level (getM stateA) bl getM stateB]
   by simp
moreover

have InvariantVarsF (getF stateB) F0 decisionVars
   using (getF stateB = getF stateA)
   (InvariantVarsF (getF stateA) F0 decisionVars)
   by simp
moreover

have InvariantConsistent (getM stateB)
   using InvariantConsistent (getM stateA)
   (isPrefix (prefixToLevel level (getM stateA)) (getM stateA))
   (isUnitClause bc bl (elements (prefixToLevel level (getM stateA))))
   (getM stateB = prefixToLevel level (getM stateA) @ [(bl, False)])
   InvariantConsistentAfterBackjump[of getM stateA prefixToLevel level (getM stateA) bc bl getM stateB]
   by simp
moreover
have InvariantUniq (getM stateB)
  using (InvariantUniq (getM stateA))
  ⟨isPrefix (prefixToLevel level (getM stateA)) (getM stateA)⟩
  ⟨isUnitClause bc bl (elements (prefixToLevel level (getM stateA)))⟩
  ⟨getM stateB = prefixToLevel level (getM stateA) @ (bl False)]
  InvariantUniqAfterBackjump (getM stateA) bc bl getM stateB]
  by simp
moreover
have InvariantEquivalent F0 (getF stateB)
  using ⟨InvariantEquivalent F0 (getF stateA)⟩
  ⟨getF stateB = getF stateA⟩
  by simp
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto
}
ultimately
show ?thesis
  using ⟨transition stateA stateB decisionVars⟩
  unfolding transition-def
  by auto
qed

The consequence is that invariants hold in all valid runs.

lemma invariantsHoldInValidRuns:
  fixes F0 :: Formula and decisionVars :: Variable set
  assumes invariantsHoldInState stateA F0 decisionVars and
  (stateA, stateB) ∈ transitionRelation decisionVars
  shows invariantsHoldInState stateB F0 decisionVars
  using assms
  using transitionsPreserveInvariants
  using rtrancl-induct[of stateA stateB
    { (stateA, stateB). transition stateA stateB decisionVars } x. invariantsHoldInState state x F0 decisionVars]
  unfolding transitionRelation-def
  by auto

lemma invariantsHoldInValidRunsFromInitialState:
  fixes F0 :: Formula and decisionVars :: Variable set
  assumesInitialState state0 F0
  and (state0, state) ∈ transitionRelation decisionVars
  shows invariantsHoldInState state F0 decisionVars
proof–
  from ⟨isInitialState state0 F0⟩
  have invariantsHoldInState state0 F0 decisionVars
In the following text we will show that there are two kinds of states:

1. \textit{UNSAT} states where \texttt{formulaFalse\ F0 (elements\ (getM\ state))} and \texttt{decisions\ (getM\ state) = \[]}.  
2. \textit{SAT} states where \(\neg\ \texttt{formulaFalse\ F0 (elements\ (getM\ state))}\) and \texttt{decisionVars \subseteq vars\ (elements\ (getM\ state))}

The soundness theorems claim that if \textit{UNSAT} state is reached the formula is unsatisfiable and if \textit{SAT} state is reached, the formula is satisfiable.

Completeness theorems claim that every final state is either \textit{UNSAT} or \textit{SAT}. A consequence of this and soundness theorems, is that if formula is unsatisfiable the solver will finish in an \textit{UNSAT} state, and if the formula is satisfiable the solver will finish in a \textit{SAT} state.

### 6.3 Soundness

\textbf{theorem soundnessForUNSAT:}

\texttt{fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State}
\texttt{assumes isInitialState state0 F0 and (state0, state) \in transitionRelation decisionVars}
\texttt{formulaFalse (getF state) (elements (getM state)) decisions (getM state) = []}
\texttt{shows \neg\ satisfiable\ F0}

\textbf{proof—}

\texttt{from (isInitialState state0 F0: (state0, state) \in transitionRelation decisionVars)}
\texttt{have invariantsHoldInState state0 F0 decisionVars using invariantsHoldInValidRunsFromInitialState}
\texttt{by simp}
\texttt{hence InvariantImpliedLiterals (getF state) (getM state) InvariantEquivalent F0 (getF state)}
\texttt{unfolding invariantsHoldInState-def}
\texttt{by auto
with ⟨formulaFalse (getF state) (elements (getM state))⟩
   ⟨decisions (getM state) = []⟩
show ⟨thesis⟩
   using unsatReport[of getF state getM state F0]
   by simp
qed

theorem soundnessForSAT:
  fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State
  assumes
    vars F0 ⊆ decisionVars and
    isInitialState state0 F0 and
    (state0, state) ∈ transitionRelation decisionVars
  ¬ formulaFalse (getF state) (elements (getM state))
  vars (elements (getM state)) ⊇ decisionVars
  shows
    model (elements (getM state)) F0
proof −
  from ⟨isInitialState state0 F0⟩ ⟨(state0, state) ∈ transitionRelation decisionVars⟩
  have invariantsHoldInState state F0 decisionVars
     using invariantsHoldInValidRunsFromInitialState
     by simp
  hence
    InvariantConsistent (getM state)
    InvariantEquivalent F0 (getF state)
    InvariantVarsF (getF state) F0 decisionVars
    unfolding invariantsHoldInState-def
    by auto
    with assms
  show ⟨thesis⟩
     using satReport[of F0 decisionVars getF state getM state]
     by simp
qed

6.4 Termination

This system is terminating, but only under assumption that there is no infinite derivation consisting only of applications of rule Learn. We will formalize this condition by requiring that there exists an ordering learnL on the formulae that is well-founded such that the state is decreased with each application of the Learn rule. If such ordering exists, the termination ordering
is built as a lexicographic combination of \textit{lexLessRestricted} trail ordering and the \textit{learnL} ordering.

\textbf{definition} \textit{lexLessState} $F_0$ decision\textit{Vars} == \{(\textit{state}A::State), (\textit{state}B::State)\},

\begin{align*}
(\text{getM \textit{state}A}, \text{getM \textit{state}B}) \in \text{lexLessRestricted} (\text{vars } F_0 \cup \text{decision\textit{Vars}})
\end{align*}

\textbf{definition} \textit{learnLessState} learn\textit{L} == \{(\textit{state}A::State), (\textit{state}B::State)\},

\begin{align*}
\text{getM \textit{state}A} = \text{getM \textit{state}B} \land (\text{getF \textit{state}A}), \text{getF \textit{state}B}) \in \text{learnL}
\end{align*}

\textbf{definition} \textit{terminationLess} $F_0$ decision\textit{Vars} learn\textit{L} == \{(\textit{state}A::State), (\textit{state}B::State)\},

\begin{align*}
(\textit{state}A,\textit{state}B) \in \textit{lexLessState} F_0 \text{ decision\textit{Vars}} \lor
(\textit{state}A,\textit{state}B) \in \textit{learnLessState} \textit{learnL}
\end{align*}

We want to show that every valid transition decreases a state with respect to the constructed termination ordering. Therefore, we show that \textit{Decide}, \textit{UnitPropagate} and \textit{Backjump} rule decrease the trail with respect to the restricted trail ordering \textit{lexLessRestricted}. Invariants ensure that trails are indeed uniq, consistent and with finite variable sets. By assumption, \textit{Learn} rule will decrease the formula component of the state with respect to the \textit{learnL} ordering.

\textbf{lemma} \textit{trailIsDecreasedByDeciedUnitPropagateAndBackjump}: \textbf{fixes} \textit{stateA::State and stateB::State} \textbf{assumes} \textit{invariantsHoldInState stateA F0 decision\textit{Vars} and}
\textit{appliedDecide stateA stateB decision\textit{Vars} \lor \textit{appliedUnitPropagate stateA stateB \lor \textit{appliedBackjump stateA stateB} \textbf{shows} (\text{getM stateB}, \text{getM stateA}) \in \text{lexLessRestricted} (\text{vars } F_0 \cup \text{decision\textit{Vars}})}

\textbf{proof}−
\textbf{from} (\textit{appliedDecide stateA stateB decision\textit{Vars} \lor \textit{appliedUnitPropagate stateA stateB \lor \textit{appliedBackjump stateA stateB})}
\textbf{have} \textit{invariantsHoldInState stateA F0 decision\textit{Vars}} \textbf{using} \textit{transitionsPreserveInvariants}
\textbf{unfolding} \textit{transition-def}
\textbf{by} auto
\textbf{from} (\textit{invariantsHoldInState stateA F0 decision\textit{Vars}})
\textbf{have} *: uniq (\textit{elements (getM stateA)}) consistent (\textit{elements (getM stateA)})\textit{vars} (\textit{elements (getM stateA)}) \subseteq \text{vars } F_0 \cup \text{decision\textit{Vars}}
\textbf{unfolding} \textit{invariantsHoldInState-def}
\textbf{unfolding} \textit{InvariantVarsM-def}
\textbf{unfolding} \textit{InvariantConsistent-def}
\textbf{unfolding} \textit{InvariantUniq-def}
\textbf{by} auto
\textbf{from} (\textit{invariantsHoldInState stateB F0 decision\textit{Vars}})
\textbf{have} **: uniq (\textit{elements (getM stateB)}) consistent (\textit{elements (getM stateB)}}

239
\( \text{stateB} \rangle \) \( \text{vars} \) \( (\text{elements} (\text{getM stateB})) \subseteq \text{vars} F0 \cup \text{decisionVars} \)

unfolding \( \text{invariantsHoldInState-def} \)
unfolding \( \text{InvariantVarsM-def} \)
unfolding \( \text{InvariantConsistent-def} \)
unfolding \( \text{InvariantUniq-def} \)

by auto

\{
assume \( \text{appliedDecide stateA stateB decisionVars} \)
\text{hence} (\text{getM stateB}, \text{getM stateA}) \in \text{lexLess}
unfolding \( \text{appliedDecide-def} \)
by (auto simp add:lexLessAppend)
with * *
have ((\text{getM stateB}), (\text{getM stateA})) \in \text{lexLessRestricted} (\text{vars} F0 \cup \text{decisionVars})
unfolding \( \text{lexLessRestricted-def} \)
by auto
\}

moreover

\{
assume \( \text{appliedUnitPropagate stateA stateB} \)
\text{hence} (\text{getM stateB}, \text{getM stateA}) \in \text{lexLess}
unfolding \( \text{appliedUnitPropagate-def} \)
by (auto simp add:lexLessAppend)
with * *
have (\text{getM stateB}, \text{getM stateA}) \in \text{lexLessRestricted} (\text{vars} F0 \cup \text{decisionVars})
unfolding \( \text{lexLessRestricted-def} \)
by auto
\}

moreover

\{
assume \( \text{appliedBackjump stateA stateB} \)
then obtain bc::\text{Clause} and bl::\text{Literal} and level::\text{nat}
where
\( \text{isUnitClause} bc \text{ bl} (\text{elements} (\text{prefixToLevel level (getM stateA)})) \)
formulaEntailsClause \( \text{getF stateA} \) \( \text{bc} \)
var \( \text{bl} \) \( \in \) \( \text{vars} \) \( \text{getF stateA} \cup \text{vars} \) \( \text{elements} \) \( \text{getM stateA} \)
\( 0 \leq \text{level level} < \text{currentLevel} \) \( \text{getM stateA} \)
\text{getF stateB} = \text{getF stateA}
\text{getM stateB} = \text{prefixToLevel level (getM stateA) @ [(bl, False)]}
unfolding \( \text{appliedBackjump-def} \)
by auto

with (\text{getM stateB} = \text{prefixToLevel level (getM stateA) @ [(bl, False)]})

have (\text{getM stateB}, \text{getM stateA}) \in \text{lexLess}
by (simp add:lexLessBackjump)
with * *
have (\text{getM stateB}, \text{getM stateA}) \in \text{lexLessRestricted} (\text{vars} F0 \cup \text{decisionVars})
Now we can show that, under the assumption for Learn rule, every rule application decreases a state with respect to the constructed termination ordering.

**Theorem stateIsDecreasedByValidTransitions:**

**Facts:**
- `fixes stateA::State and stateB::State`
- `assumes invariantsHoldInState stateA F0 decisionVars and transition stateA stateB decisionVars`
- `appliedLearn stateA stateB → (getF stateB, getF stateA) ∈ learnL`
- `shows (stateB, stateA) ∈ terminationLess F0 decisionVars learnL`

**Proof:**

1. Assume any of the following:
   - `appliedDecide stateA stateB decisionVars`
   - `appliedUnit-Propagate stateA stateB`
   - `appliedBackjump stateA stateB`

   Using the assumption, we have:
   - `have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)`

   Unfolding `lexLessRestricted-def` by `simp`.

2. Assume `appliedLearn stateA stateB` with `appliedLearn stateA stateB → (getF stateB, getF stateA) ∈ learnL`

   Unfolding `appliedLearn-def` by `auto`.

   Ultimately, we have `(stateB, stateA) ∈ learnLessState learnL`
unfolding learnLessState-def
by simp
hence (stateB, stateA) ∈ terminationLess F0 decisionVars learnL
unfolding terminationLess-def
by simp
}
ultimately
show ?thesis
  using ⟨transition stateA stateB decisionVars⟩
unfolding transition-def
by auto
qed

The minimal states with respect to the termination ordering are final i.e., no further transition rules are applicable.

definition isMinimalState stateMin F0 decisionVars learnL == (∀ state::State. (state, stateMin) ∉ terminationLess F0 decisionVars learnL)

lemma minimalStatesAreFinal:
  fixes stateA::State
  assumes *: ∀ (stateA::State) (stateB::State). appliedLearn stateA stateB → (getF stateB, getF stateA) ∈ learnL and invariantsHoldInState state F0 decisionVars and isMinimalState state F0 decisionVars learnL
  shows isFinalState state decisionVars
proof
  { 
    assume ¬ ?thesis
    then obtain state′::State
      where transition state state′ decisionVars
      unfolding isFinalState-def
      by auto
    with ⟨invariantsHoldInState state F0 decisionVars⟩ *
    have (state′, state) ∈ terminationLess F0 decisionVars learnL
      using stateIsDecreasedByValidTransitions[of state F0 decisionVars state′ learnL]
      unfolding transition-def
      by auto
    with ⟨isMinimalState state F0 decisionVars learnL⟩
    have False
      unfolding isMinimalState-def
      by auto
  }
  thus ?thesis
  by auto
qed

We now prove that termination ordering is well founded. We
start with two auxiliary lemmas.

**lemma** `wfLexLessState`:

*fixes* `decisionVars` :: Variable set and `F0` :: Formula

*assumes* finite `decisionVars`

*shows* `wf` (`lexLessState F0 decisionVars`)

*unfolding* `wf-eq-minimal`

*proof* –

show `∀ Q state. state ∈ Q → (∃ stateMin ∈ Q. ∀ state'. (state', stateMin) ∈ lexLessState F0 decisionVars → state' ∉ Q)`

*proof* –

{ 
  fix `Q` :: State set and `state` :: State
  assume `state ∈ Q`
  let `?Q1 = {M :: LiteralTrail. ∃ state. state ∈ Q ∧ (getM state) = M}`
  from `(state ∈ Q)`
  have `getM state ∈ ?Q1` by auto
  from `(finite decisionVars)`
  have finite `(vars F0 ∪ decisionVars)`
    using `finiteVarsFormula[of F0]`
    by simp
  hence `wf (lexLessRestricted (vars F0 ∪ decisionVars))` using `wfLexLessRestricted[of vars F0 ∪ decisionVars]`
  by simp
  with `(getM state ∈ ?Q1)`
  obtain `Mmin where Mmin ∈ ?Q1 ∀ M'. (M', Mmin) ∈ lexLessRestricted (vars F0 ∪ decisionVars) → M' ∉ ?Q1` using `wf-eq-minimal`
  apply (erule_tac `x=:?Q1` in allE)
  apply (erule-tac `x=getM state in allE`)
  by auto
  from `(Mmin ∈ ?Q1)` obtain `stateMin` where `stateMin ∈ Q (getM stateMin) = Mmin` by auto
  have `∀ state'. (state', stateMin) ∈ lexLessState F0 decisionVars → state' ∉ Q`
  proof
    fix `state'`
    show `(state', stateMin) ∈ lexLessState F0 decisionVars → state' ∉ Q`
    proof
      assume `(state', stateMin) ∈ lexLessState F0 decisionVars`
      hence `(getM state', getM stateMin) ∈ lexLessRestricted (vars F0 ∪ decisionVars)`
      unfolding `lexLessState-def`
      by auto
      from `(∀ M'. (M', Mmin) ∈ lexLessRestricted (vars F0 ∪ decisionVars) → M' ∉ ?Q1)`
\[(\text{getM state}', \text{getM stateMin}) \in \text{lexLessRestricted} (\text{vars F0} \cup \text{decisionVars}) ; (\text{getM stateMin} = \text{Mmin})\]

\begin{itemize}
\item have \(\text{getM state}' \notin \ ?Q1\)
  by simp
\item with \(\text{getM stateMin} = \text{Mmin}\)
  show \(\text{state}' \notin Q\)
  by auto
\end{itemize}

qed

qed

with \(\text{stateMin} \in Q\)

have \(\exists \text{stateMin} \in Q, \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{lexLessState} F0 \text{ decisionVars} \rightarrow \text{state}' \notin Q\)
  by auto

\}

thus \(\exists \text{thesis}\)
  by auto

qed

qed

lemma \(\text{wfLearnLessState}\):\n
assumes \(\text{wf learnL}\)

shows \(\text{wf (learnLessState learnL)}\)

unfolding \(\text{wf-eq-minimal}\)

proof–

show \(\forall Q \text{ state. state} \in Q \rightarrow (\exists \text{stateMin} \in Q, \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{learnLessState learnL} \rightarrow \text{state}' \notin Q)\)

proof–

\{
  fix \(Q :: \text{State set} \text{ and state :: State}\)
  assume \(\text{state} \in Q\)
  let \(\text{?M} = (\text{getM state})\)
  let \(\text{?Q1} = \{ f :: \text{Formula}. \exists \text{state. state} \in Q \land (\text{getM state}) = \text{?M} \land (\text{getF state}) = f \}\)
  from \(\text{?state} \in Q\)
  have \(\text{getF state} \in ?Q1\)
  by auto
  with \(\text{wf learnL}\)
  obtain \(\text{FMin where FMin} \in ?Q1 \forall F'. (F', \text{FMin}) \in \text{learnL} \rightarrow F' \notin ?Q1\)

  unfolding \(\text{wf-eq-minimal}\)

  apply \((\text{erule-tac z=?Q1 in allE})\)

apply \((\text{erule-tac x=getF state in allE})\)
  by auto

from \(\text{FMin} \in ?Q1) \ obtain \text{stateMin}
where \(\text{stateMin} \in Q (\text{getM stateMin}) = \text{?M getF stateMin} = \text{FMin}\)
    by auto
  have \(\forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{learnLessState learnL} \rightarrow \text{state}' \notin Q\)

\}

244
proof
fix state'
show (state', stateMin) ∈ learnLessState learnL −→ state' \notin Q
proof
assume (state', stateMin) ∈ learnLessState learnL
with \langle \text{getM stateMin} = ?M \rangle
have getM state' = getM stateMin (getF state', getF stateMin)
∈ learnL
unfolding learnLessState-def
by auto
from \langle \forall F'. (F', FMin) ∈ learnL −→ F' \notin ?Q1 \rangle
\langle (getF state', getF stateMin) ∈ learnL; (getF stateMin = FMin) \rangle
have getF state' \notin ?Q1
by simp
with \langle \text{getM state'} = \text{getM stateMin} \rangle \langle \text{getM stateMin} = ?M \rangle
show state' \notin Q
by auto
qed
qed
with \langle \text{stateMin} \in Q \rangle
have \exists stateMin \in Q. (\forall state'. (state', stateMin) ∈ learnLessState
learnL −→ state' \notin Q)
by auto
}
thus \text{?thesis}
by auto
qed
qed

Now we can prove the following key lemma which shows that the
termination ordering is well founded.

lemma \text{wfTerminationLess}:
fixes F0 :: Formula and decisionVars :: Variable set
assumes finite decisionVars \text{ wf learnL}
sows \text{ wf (terminationLess F0 decisionVars learnL)}
unfolding \text{wf-eq-minimal}
proof−
show \forall Q. \forall state. state ∈ Q −→ (\exists stateMin ∈ Q. \forall state'. (state',
stateMin) ∈ \text{ terminationLess F0 decisionVars learnL −→ state' \notin Q})
proof−
{ 
fix Q::State set
fix state::State
assume state ∈ Q
have \text{ wf (lexLessState F0 decisionVars)}
using \text{wfLexLessState[of decisionVars F0]}
using \text{(finite decisionVars)}
by simp

}
with \( \text{state} \in Q \) obtain \( \text{state0} \)
where \( \text{state0} \in Q \) \( \forall \text{state}' \). (\( \text{state}' \), \( \text{state0} \)) \( \in \text{lexLessState} \) \( F0 \) \( \text{decisionVars} \) \( \rightarrow \) \( \text{state}' \not\in Q \)

unfolding \( \text{wf-eq-minimal} \)
by auto
let \( ?Q0 = \{ \text{state}. \text{state} \in Q \land (\text{getM state}) = (\text{getM state0}) \} \)
from : \( \text{state0} \in Q \)
have \( \text{state0} \in ?Q0 \)
by simp
from \( \text{wf learnL} \)
have \( \text{wf } (\text{learnLessState} \text{ learnL}) \)
using \( \text{wfLearnLessState} \)
by simp
with \( \text{state0} \in ?Q0 \) obtain \( \text{state1} \)
where \( \text{state1} \in ?Q0 \) \( \forall \text{state}' \). (\( \text{state}' \), \( \text{state1} \)) \( \in \text{learnLessState} \) \( \text{learnL} \) \( \rightarrow \) \( \text{state}' \not\in ?Q0 \)

unfolding \( \text{wf-eq-minimal} \)
apply (erule-tac \( x=\text{state0} \) in allE)
apply (erule-tac \( x=\text{state0} \) in allE)
by auto
from : \( \text{state1} \in ?Q0 \)
have \( \text{state1} \in Q \) \( \text{getM state1} = \text{getM state0} \)
by auto
let \( ?\text{stateMin} = \text{state1} \)
have \( \forall \text{state}' . (\text{state}' , ?\text{stateMin} ) \in \text{terminationLess} \ F0 \) \( \text{decisionVars} \) \( \text{learnL} \) \( \rightarrow \) \( \text{state}' \not\in Q \)
proof
fix \( \text{state}' \)
show (\( \text{state}' , ?\text{stateMin} ) \in \text{terminationLess} \ F0 \) \( \text{decisionVars} \) \( \text{learnL} \) \( \rightarrow \) \( \text{state}' \not\in Q \)
proof
assume (\( \text{state}' , ?\text{stateMin} ) \in \text{terminationLess} \ F0 \) \( \text{decisionVars} \) \( \text{learnL} \)

hence
(\( \text{state}' , ?\text{stateMin} ) \in \text{lexLessState} \ F0 \) \( \text{decisionVars} \) \( \lor \)
(\( \text{state}' , ?\text{stateMin} ) \in \text{learnLessState} \text{ learnL} \)
unfolding \( \text{terminationLess-def} \)
by auto
moreover
\{ assume (\( \text{state}' , ?\text{stateMin} ) \in \text{lexLessState} \ F0 \) \( \text{decisionVars} \)
with (\( \text{getM state1} = \text{getM state0} \))
have (\( \text{state}' , \text{state0} ) \in \text{lexLessState} \ F0 \) \( \text{decisionVars} \)
unfolding \( \text{lexLessState-def} \)
by simp
with \( \forall \text{state}' . (\text{state}' , \text{state0} ) \in \text{lexLessState} \ F0 \) \( \text{decisionVars} \)
\( \rightarrow \) \( \text{state}' \not\in Q \)
have \( \text{state}' \not\in Q \)
by simp

246
moreover
{
  assume (state', ?stateMin) ∈ learnLessState learnL
  with ∀ state'. (state', state1) ∈ learnLessState learnL —>
  state' ∉ ?Q0:
  have state' ∉ ?Q0
  by simp
  from ((state', state1) ∈ learnLessState learnL) (getM state1
  = getM state0):
  have getM state' = getM state0
  unfolding learnLessState-def
  by auto
  with (state' ∉ ?Q0)
  have state' ∉ Q
  by simp
}
ultimately
show state' ∉ Q
by auto
qed
qed

Using the termination ordering we show that the transition relation is well founded on states reachable from initial state. The assumption for the Learn rule is necessary.

**Theorem** *wfTransitionRelation*:  
fixes decisionVars :: Variable set and F0 :: Formula  
assumes finite decisionVars and isInitialState state0 F0 and  
*: ∃ learnL::(Formula × Formula) set.
  wf learnL ∧  
  (∀ stateA stateB. appliedLearn stateA stateB —> (getF stateB,
  getF stateA) ∈ learnL)  
shows wf { (state0, stateA).
  (state0, stateA) ∈ transitionRelation decisionVars ∧ 
  (transition stateA stateB decisionVars)}

**Proof**
from * obtain learnL::(Formula × Formula) set
  where
  wf learnL and
∀ stateA stateB. appliedLearn stateA stateB → (getF stateB, getF stateA) ∈ learnL
     by auto
lev ?rel = {(stateB, stateA).
 (state0, stateA) ∈ transitionRelation decisionVars ∧
 (transition stateA stateB decisionVars)}
     let ?rel′ = terminationLess F0 decisionVars learnL
have ∀ x y. (x, y) ∈ ?rel → (x, y) ∈ ?rel′
proof−
 {          
     fix stateA::State and stateB::State
     assume (stateB, stateA) ∈ ?rel
     hence (stateB, stateA) ∈ ?rel'
     using isInitialState state0 F0
     using invariantsHoldInValidRunsFromInitialState[of state0 F0 stateA decisionVars]
     using stateIsDecreasedByValidTransitions[of stateA F0 decisionVars stateB] **
         by simp
     }        
     thus ?thesis
     by simp
qed
moreover
have wf ?rel'
    using (finite decisionVars) (wf learnL)
    by (rule wfTerminationLess)
ultimately
show ?thesis
    using wellFoundedEmbed[of ?rel ?rel']
    by simp
qed

We will now give two corollaries of the previous theorem. First is a weak termination result that shows that there is a terminating run from every initial state to the final one.
corollary
  fixes decisionVars :: Variable set and F0 :: Formula and state0 :: State
  assumes finite decisionVars and isInitialState state0 F0 and
  ∗: ∃ learnL::(Formula × Formula) set.
       wf learnL ∧
       (∀ stateA stateB. appliedLearn stateA stateB → (getF stateB, getF stateA) ∈ learnL)
  shows ∃ state. (state0, state) ∈ transitionRelation decisionVars ∧
              isFinalState state decisionVars
proof−
 {
assume \( \neg \textit{thesis} \)

let \( ?Q = \{ \textit{state}. (\textit{state0}, \textit{state}) \in \text{transitionRelation decisionVars} \} \)

let \( ?rel = \{(\textit{stateB}, \textit{stateA}). (\textit{state0}, \textit{stateA}) \in \text{transitionRelation decisionVars} \land \text{transition stateA stateB decisionVars} \} \)

have \( \textit{state0} \in ?Q \)
  unfolding transitionRelation-def
  by simp
  hence \( \exists \textit{state}. \textit{state} \in ?Q \)
  by auto

from assms
have \( \text{wf } ?rel \)
  using \( \text{wfTransitionRelation[of decisionVars state0 F0} \]
  by auto
  hence \( \forall Q. (\exists x. x \in Q) \longrightarrow (\exists \textit{stateMin} \in Q. \forall \textit{state}. (\textit{state}, \textit{stateMin}) \in ?rel \longrightarrow \textit{state} \not\in Q) \)
  unfolding \( \text{wf-eq-minimal} \)
  by simp
  hence \( (\exists x. x \in ?Q) \longrightarrow (\exists \textit{stateMin} \in ?Q. \forall \textit{state}. (\textit{state}, \textit{stateMin}) \in ?rel \longrightarrow \textit{state} \not\in ?Q) \)
  by rule
  with \( (\exists \textit{state}. \textit{state} \in ?Q) \)
  have \( \exists \textit{stateMin} \in ?Q. \forall \textit{state}. (\textit{state}, \textit{stateMin}) \in ?rel \longrightarrow \textit{state} \not\in ?Q \)
    by simp
  then obtain \( \textit{stateMin} \)
    where \( \textit{stateMin} \in ?Q \land \forall \textit{state}. (\textit{state}, \textit{stateMin}) \in ?rel \longrightarrow \textit{state} \not\in ?Q \)
    by auto

from \( (\textit{stateMin} \in ?Q) \)
have \( (\textit{state0}, \textit{stateMin}) \in \text{transitionRelation decisionVars} \)
  by simp
with \( (\neg \textit{thesis}) \)
have \( \neg \textit{isFinalState stateMin decisionVars} \)
  by simp
then obtain \( \textit{state'}::\text{State} \)
  where \( \text{transition stateMin state'} decisionVars \)
  unfolding \( \text{isFinalState-def} \)
  by auto
have \( (\textit{state'}, \textit{stateMin}) \in ?rel \)
  using \( (\textit{state0}, \textit{stateMin}) \in \text{transitionRelation decisionVars} \)
  \( (\text{transition stateMin state'} decisionVars) \)
  by simp
with \( (\forall \textit{state}. (\textit{state}, \textit{stateMin}) \in ?rel \longrightarrow \textit{state} \not\in ?Q) \)
have \( \textit{state'} \not\in ?Q \)
  by force
moreover

249
from \((\text{state0}, \text{stateMin}) \in \text{transitionRelation} \, \text{decisionVars} : (\text{transition stateMin state'} \, \text{decisionVars})\)

have state' \in ?Q

unfolding \text{transitionRelation-def} using \text{rtrancl-into-rtrancl}[\text{of state0 stateMin} \{ (\text{stateA, stateB}). transition \text{stateA stateB decisionVars} \}, state']

by simp
ultimately
have False
by simp

} thus ?thesis
by auto

qed

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

corollary noInfiniteTransitionChains:
fixes \(F0::\text{Formula} \) and \(\text{decisionVars}::\text{Variable set}\)
assumes finite \(\text{decisionVars} \) and
\(\ast::\exists \text{learnL}::(\text{Formula} \times \text{Formula}) \text{ set}.
\text{wf learnL} \land\)
(\(\forall \text{stateA stateB}. \text{appliedLearn stateA stateB} \rightarrow (\text{getF stateB, getF stateA}) \in \text{learnL}\))

shows \(\neg (\exists Q::(\text{State set}). \exists \text{state0} \in Q. \text{isInitialState state0 F0} \land\)

(\(\forall \text{state} \in Q. (\exists \text{state'} \in Q. \text{transition state state'} \, \text{decisionVars})\))

proof–
{
assume \(\neg ?\text{thesis}\)
then obtain \(Q::\text{State set} \) and \(\text{state0}::\text{State}\)
where \(\text{isInitialState state0 F0 state0} \in Q\)

\(\forall \text{state} \in Q. (\exists \text{state'} \in Q. \text{transition state state'} \, \text{decisionVars})\)

by auto

let \(?\text{rel} = \{ (\text{stateB, stateA}). (\text{state0, stateA}) \in \text{transitionRelation} \, \text{decisionVars} \land\)

\(\text{transition stateA stateB decisionVars} \}\)
from \(\text{finite decisionVars} : \text{isInitialState state0 F0} \ast\)

have \(\text{wf ?rel}\)
using \(\text{wfTransitionRelation}\)
by simp
hence \( \text{wfmin}: \forall x. x \in Q \rightarrow (\exists z \in Q. \forall y. (y, z) \in \text{rel} \rightarrow y \notin Q) \)

unfolding \( \text{wf-eq-minimal} \)
by \( \text{simp} \)
let \( \overline{Q} = \{ \text{state} \in Q. (\text{state0}, \text{state}) \in \text{transitionRelation decision-Vars} \} \)
from \( \langle \text{state0} \in Q \rangle \)
have \( \text{state0} \in \overline{Q} \)
unfolding \( \text{transitionRelation-def} \)
by \( \text{simp} \)
with \( \text{wfmin} \)
obtain \( \text{stateMin::State} \)
where \( \text{stateMin} \in \overline{Q} \text{ and } \forall y. (y, \text{stateMin}) \in \overline{\text{rel}} \rightarrow y \notin \overline{Q} \)
apply \( (\text{erule-tac} x=\overline{Q} \text{ in allE}) \)
by \( \text{auto} \)
from \( \langle \text{stateMin} \in \overline{Q} \rangle \)
have \( \text{stateMin} \in Q \langle (\text{state0}, \text{stateMin}) \in \text{transitionRelation decision-Vars} \rangle \)
by \( \text{auto} \)
with \( (\forall \text{ state} \in Q. (\exists \text{ state}' \in Q. \text{transition state state'} \text{ decision-Vars}) \) \)
obtain \( \text{state'}::\text{State} \)
where \( \text{state'} \in Q \langle \text{transition stateMin state'} \text{ decision-Vars} \rangle \)
by \( \text{auto} \)
with \( (\langle \langle (\text{state0}, \text{stateMin}) \in \text{transitionRelation decision-Vars} \rangle \)
have \( (\text{state'}, \text{stateMin}) \in \overline{\text{rel}} \)
by \( \text{simp} \)
with \( (\forall y. (y, \text{stateMin}) \in \overline{\text{rel}} \rightarrow y \notin \overline{Q}) \)
have \( \text{state'} \notin \overline{Q} \)
by \( \text{force} \)
from \( (\langle \text{state'} \in Q \rangle \langle (\text{state0}, \text{stateMin}) \in \text{transitionRelation decision-Vars} \rangle \)
\langle \text{transition stateMin state'} \text{ decision-Vars} \rangle \)
have \( \text{state'} \in \overline{Q} \)
unfolding \( \text{transitionRelation-def} \)
using \( \text{rtrancl-into-rtrancl} \{ \text{of state0 stateMin} \ (\text{stateA}, \text{stateB}). \text{transition stateA stateB decision-Vars} \} \text{ state'} \)
by \( \text{simp} \)
with \( (\text{state'} \notin \overline{Q}) \)
have \( \text{False} \)
by \( \text{simp} \)
\}
thus \( \text{thesis} \)
by \( \text{force} \)
qed
6.5 Completeness

In this section we will first show that each final state is either SAT or UNSAT state.

**lemma finalNonConflictState:**

```plaintext
fixes state::State and FO :: Formula
assumes 
¬ applicableDecide state decisionVars
shows vars (elements (getM state)) ⊇ decisionVars
```

**proof**

```plaintext
fix x :: Variable
let ?l = Pos x
assume x ∈ decisionVars
hence var ?l = x and var ?l ∈ decisionVars and var (opposite ?l) ∈ decisionVars
by auto
with (∀ applicableDecide state decisionVars)
have literalTrue ?l (elements (getM state)) ∨ literalFalse ?l (elements (getM state))
  unfolding applicableDecideCharacterization
  by force
  with (var ?l = x)
  show x ∈ vars (elements (getM state))
    using valuationContainsItsLiteralsVariable[of ?l elements (getM state)]
    using valuationContainsItsLiteralsVariable[of opposite ?l elements (getM state)]
    by auto
qed
```

**lemma finalConflictingState:**

```plaintext
fixes state :: State
assumes
InactiveUniq (getM state) and
InvariantConsistent (getM state) and
InvariantImpliedLiterals (getF state) (getM state)
¬ applicableBackjump state and
formulaFalse (getF state) (elements (getM state))
shows
decisions (getM state) = []
```

**proof**

```plaintext
from (InactiveUniq (getM state))
have uniq (elements (getM state))
  unfolding InvariantUniq-def
.
from (InvariantConsistent (getM state))
have consistent (elements (getM state))
  unfolding InvariantConsistent-def
```
let \( ?c = \text{oppositeLiteralList}(\text{decisions}(\text{getM state})) \)
{
  \text{assume } \neg \text{thesis}
  \text{hence } ?c \neq []
    \text{using } \text{oppositeLiteralListNonempty}[\text{of decisions}(\text{getM state})]
    \text{by simp}
  \text{moreover}
  \text{have } \text{clauseFalse}(?c, \text{elements}(\text{getM state}))
  \text{proof—}
  \{
    \text{fix } l:\text{Literal}
    \text{assume } l \in ?c
    \text{hence } \text{opposite}(l, \text{el decisions}(\text{getM state}))
      \text{using } \text{literalElListIffOppositeLiteralElOppositeLiteralList}[\text{of l ?c}]
      \text{by simp}
    \text{hence } \text{literalFalse}(l, \text{elements}(\text{getM state}))
      \text{using } \text{markedElementsAreElements}[\text{of opposite}(l, \text{getM state})]
      \text{by simp}
  \}
  \text{thus } \text{thesis}
    \text{using } \text{clauseFalseIffAllLiteralsAreFalse}[\text{of } ?c, \text{elements}(\text{getM state})]
    \text{by simp}
  \text{qed}
  \text{moreover}
  \text{let } ?l = \text{getLastAssertedLiteral}(\text{oppositeLiteralList}(?c) \text{ (elements}(\text{getM state}))
  \text{have } \text{isLastAssertedLiteral}(?l, \text{oppositeLiteralList}(?c) \text{ (elements}(\text{getM state}))
    \text{using } \langle \text{InvariantUniq}(\text{getM state}) \rangle
    \text{using } \text{getLastAssertedLiteralCharacterization}[\text{of } ?c, \text{elements}(\text{getM state})]
    \langle ?c \neq [] \rangle \langle \text{clauseFalse}(?c, \text{elements}(\text{getM state})) \rangle
    \text{unfolding } \text{InvariantUniq-def}
    \text{by simp}
  \text{moreover}
  \text{have } \forall l. l \in ?c \rightarrow (\text{opposite}(l) \text{ el (decisions}(\text{getM state}))
  \text{proof—}
  \{
    \text{fix } l:\text{Literal}
    \text{assume } l \in ?c
    \text{hence } (\text{opposite}(l) \text{ el (oppositeLiteralList}(?c)
      \text{using } \text{literalElListIffOppositeLiteralElOppositeLiteralList}[\text{of l ?c}]
      \text{by simp}
  \}
thus \(?\text{thesis}\)
  by simp

qed

ultimately
have \(\exists\ \text{level}.\ \text{isBackjumpLevel}\ \text{level} (\text{opposite} \ ?l) \ ?c (\text{getM} \ \text{state})\)
  using \(\text{uniq} (\text{elements} (\text{getM} \ \text{state}))\);
  using \(\text{allDecisionsThenExistsBackjumpLevel}[[\text{of getM} \ \text{state} \ ?c \ \text{opposite} \ ?l]]\)
  by simp

then obtain level::nat
  where \(\text{isBackjumpLevel}\ \text{level} (\text{opposite} \ ?l) \ ?c (\text{getM} \ \text{state})\)
  by auto

with \(\text{consistent} (\text{elements} (\text{getM} \ \text{state}));\) \(\text{uniq} (\text{elements} (\text{getM} \ \text{state}));\) \(\text{clauseFalse} \ ?c (\text{elements} (\text{getM} \ \text{state}));\)
have \(\text{isUnitClause} \ ?c (\text{opposite} \ ?l) (\text{elements} (\text{prefixToLevel} \ \text{level} (\text{getM} \ \text{state})))\)
  using \(\text{isBackjumpLevelEnsuresIsUnitInPrefix}[\text{of getM} \ \text{state} \ ?c \ \text{level} \ \text{opposite} \ ?l]\)
  by simp

moreover
have \(\text{formulaEntailsClause} (\text{getF} \ \text{state}) \ ?c\)
proof−
  from \(\text{clauseFalse} \ ?c (\text{elements} (\text{getM} \ \text{state}));\) \(\text{consistent} (\text{elements} (\text{getM} \ \text{state}));\)
  have \(\neg\ \text{clauseTautology} \ ?c\)
    using \(\text{tautologyNotFalse}[\text{of} \ ?c \ \text{elements} (\text{getM} \ \text{state})]\)
    by auto

from \(\text{formulaFalse} (\text{getF} \ \text{state}) (\text{elements} (\text{getM} \ \text{state}));\) \(\text{InvariantImpliedLiterals} (\text{getF} \ \text{state}) (\text{getM} \ \text{state});\)
  have \(\neg\ \text{satisfiable} ((\text{getF} \ \text{state}) \ @ val2form (\text{decisions} (\text{getM} \ \text{state})))\)
  using \(\text{InvariantImpliedLiteralsAndFormulaFalseThenFormulaAndDecisionsAreNotSatisfiable}\)
  by simp
  hence \(\neg\ \text{satisfiable} ((\text{getF} \ \text{state}) \ @ val2form (\text{oppositeLiteralList} \ ?c))\)
  by simp

with \(\neg\ \text{clauseTautology} \ ?c\)
show \(?\text{thesis}\)
  using \(\text{unsatisfiableFormulaWithSingleLiteralClauses}\)
  by simp

qed

moreover
have \(\text{var} \ ?l \ \in\ \text{vars} (\text{getF} \ \text{state}) \cup \text{vars} (\text{elements} (\text{getM} \ \text{state}))\)
proof−
  from \(\text{isLastAssertedLiteral} \ ?l (\text{oppositeLiteralList} \ ?c) (\text{elements} (\text{getM} \ \text{state}))\);
  have \(?l\ \text{el} (\text{oppositeLiteralList} \ ?c)\)
    unfolding \(\text{isLastAssertedLiteral-def}\)

254
by simp
hence literalTrue ?l (elements (getM state))
by (simp add: markedElementsAreElements)

hence var ?l ∈ vars (elements (getM state))
using valuationContainsItsLiteralsVariable[of ?l elements (getM state)]

by simp
thus ?thesis
by simp
qed

moreover
have 0 ≤ level level < (currentLevel (getM state))

proof -
from isBackjumpLevel level (opposite ?l) ?c (getM state)
have 0 ≤ level level < (elementLevel ?l (getM state))
unfolding isBackjumpLevel-def
by auto
thus 0 ≤ level level < (currentLevel (getM state))
using elementLevelLeqCurrentLevel[of ?l getM state]
by auto

qed

ultimately
have applicableBackjump state
unfolding applicableBackjumpCharacterization
by force
with (∀ applicableBackjump state):
have False
by simp

} 
thus ?thesis
by auto

qed

lemma finalStateCharacterizationLemma:
fixes state :: State
assumes
InvariantUniq (getM state) and
InvariantConsistent (getM state) and
InvariantImpliedLiterals (getF state) (getM state)
¬ applicableDecide state decisionVars and
¬ applicableBackjump state

shows
(¬ formulaFalse (getF state) (elements (getM state)) ∧ vars (elements (getM state)) ≥ decisionVars) ∨
(formulaFalse (getF state) (elements (getM state)) ∧ decisions (getM state) = [])
proof (cases formulaFalse (getF state) (elements (getM state)))
case True
hence decisions (getM state) = []
using assms
using finalConflictingState
by auto
with True
show \( ?thesis \)
  by simp
next
case False
hence \( \text{vars} (\text{elements} (\text{getM state})) \supseteq \text{decisionVars} \)
using assms
using finalNonConflictState
by auto
with False
show \( ?thesis \)
  by simp
qed

**Theorem Final State Characterization:**

**Fixes** \( F_0 :: \text{Formula} \) and \( \text{decisionVars} :: \text{Variable set} \) and \( \text{state}_0 :: \text{State} \) and \( \text{state} :: \text{State} \)

**Assumes**

- \( \text{isInitialState} \text{state}_0 F_0 \) and
- \( (\text{state}_0, \text{state}) \in \text{transitionRelation} \text{decisionVars} \) and
- \( \text{isFinalState} \text{state} \text{decisionVars} \)

**Shows**

\[
(\neg \text{formulaFalse} (\text{getF state}) (\text{elements} (\text{getM state})) \land \text{vars} (\text{elements} (\text{getM state})) \supseteq \text{decisionVars}) \lor \\
(\text{formulaFalse} (\text{getF state}) (\text{elements} (\text{getM state})) \land \text{decisions} (\text{getM state}) = [])
\]

**Proof**—

from \( \langle \text{isInitialState} \text{state}_0 F_0; (\text{state}_0, \text{state}) \in \text{transitionRelation} \text{decisionVars} \rangle \)

have \( \text{invariantsHoldInState} \text{state} F_0 \text{decisionVars} \)
  using \( \text{invariantsHoldInValidRunsFromInitialState} \)
  by simp

hence

\[
*: \text{InvariantUniq} (\text{getM state})
\text{InvariantConsistent} (\text{getM state})
\text{InvariantImpliedLiterals} (\text{getF state}) (\text{getM state})
\]

unfolding \( \text{invariantsHoldInState-def} \)
  by auto

from \( \langle \text{isFinalState} \text{state} \text{decisionVars} \rangle \)

have **:

\[ \neg \text{applicableBackjump state} \]
\[ \neg \text{applicableDecide state} \text{decisionVars} \]
unfolding \( \text{finalStateNonApplicable} \)
Completeness theorems are easy consequences of this characterization and soundness.

**Theorem completenessForSAT:**

- **Fixes** $F_0 :: 	ext{Formula}$ and $\text{decisionVars} :: \text{Variable set}$ and $\text{state0} :: \text{State}$
- **Assumes**
  - $\text{satisfiable } F_0$
  - $\text{isInitialState } \text{state0} \ F_0$
- **Shows** $\neg \text{formulaFalse} \ (\text{getF } \text{state}) \ (\text{elements } (\text{getM } \text{state})) \land \text{vars} \ (\text{elements } (\text{getM } \text{state})) \supseteq \text{decisionVars}$

**Proof**

- **From** `assms`
  - **Have** $\ast$: $(\neg \text{formulaFalse} \ (\text{getF } \text{state}) \ (\text{elements } (\text{getM } \text{state})) \land \text{vars} \ (\text{elements } (\text{getM } \text{state})) \supseteq \text{decisionVars}) \lor$
    - $(\text{formulaFalse} \ (\text{getF } \text{state}) \ (\text{elements } (\text{getM } \text{state})) \land \text{decisions} \ (\text{getM state}) = [])$
  - **Using** `finalStateCharacterizationLemma[of state decisionVars]`
  - **By** `auto`

  - **Assume** $\text{formulaFalse} \ (\text{getF state}) \ (\text{elements } (\text{getM state}))$
    - **With** $\ast$
      - **Have** $\text{formulaFalse} \ (\text{getF state}) \ (\text{elements } (\text{getM state})) \land \text{decisions} \ (\text{getM state}) = []$
        - **By** `auto`
    - **With** `assms`
      - **Have** $\neg \text{satisfiable } F_0$
        - **Using** `soundnessForUNSAT`
        - **By** `simp`
    - **Have** $\text{False}$
        - **By** `simp`
  - **With** $\ast$
    - **Show** $\ast$thesis
      - **By** `auto`

**Qed**
theorem completenessForUNSAT:
  fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State
  assumes vars F0 ⊆ decisionVars and
      ¬ satisfiable F0 and
  isInitialState state0 F0 and
  (state0, state) ∈ transitionRelation decisionVars and
  isFinalState state decisionVars
  shows
      formulaFalse (getF state) (elements (getM state)) ∧ decisions (getM state) = []
proof
  from assms have *:
    (¬ formulaFalse (getF state) (elements (getM state)) ∧ vars (elements (getM state)) ⊇ decisionVars) ∨
    (formulaFalse (getF state) (elements (getM state)) ∧ decisions (getM state) = [])
    using finalStateCharacterization[of state0 F0 state decisionVars]
    by auto
  { assume ¬ formulaFalse (getF state) (elements (getM state))
      with *
      have ¬ formulaFalse (getF state) (elements (getM state)) vars (elements (getM state)) ⊇ decisionVars
        by auto
      with assms have satisfiable F0
        using soundnessForSAT[of F0 decisionVars state0 state]
        unfolding satisfiable-def
        by auto
      with (¬ satisfiable F0)
      have False
        by simp
  } with * show ?thesis
  by auto
qed

theorem partialCorrectness:
  fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State
  assumes vars F0 ⊆ decisionVars and
isInitialState state0 F0 and 
(state0, state) ∈ transitionRelation decisionVars and 
isFinalState state decisionVars 
shows 
satisfiable \( F0 = (\neg \text{formulaFalse} (\text{getF state}) (\text{elements} (\text{getM state}))) \)

using assms 
using completenessForUNSAT[of F0 decisionVars state0 state] 
using completenessForSAT[of F0 state0 state decisionVars] 
by auto

end

7 Transition system of Krstić and Goel.

theory KrsticGoel
imports SatSolverVerification
begin

This theory formalizes the transition rule system given by Krstić and Goel in [1]. Some rules of the system are generalized a bit, so that the system can model some more general solvers (e.g., SMT solvers).

7.1 Specification

record State = 
getF :: Formula 
getM :: LiteralTrail 
getConflictFlag :: bool 
getC :: Clause

definition appliedDecide:: State ⇒ State ⇒ Variable set ⇒ bool 
where 
appliedDecide stateA stateB decisionVars ==
  \( \exists \ l. \) (\( \text{var} \ l \) \( \in \) decisionVars \( \land \) 
  \( \neg \ l \ \text{el} \ (\text{elements} (\text{getM stateA})) \) \( \land \) 
  \( \neg \ \text{opposite} \ l \ \text{el} \ (\text{elements} (\text{getM stateA})) \) \( \land \) 
  getF stateB = getF stateA \( \land \) 
  getM stateB = getM stateA @ [(\l, True)] \( \land \) 
  getConflictFlag stateB = getConflictFlag stateA \( \land \) 
  getC stateB = getC stateA)

definition
applicableDecide :: State ⇒ Variable set ⇒ bool
where
applicableDecide state decisionVars == ∃ state'. appliedDecide state state' decisionVars

Notice that the given UnitPropagate description is weaker than in original [1] paper. Namely, propagation can be done over a clause that is not a member of the formula, but is entailed by it. The condition imposed on the variable of the unit literal is necessary to ensure the termination.

definition
appliedUnitPropagate :: State ⇒ State ⇒ Formula ⇒ Variable set ⇒ bool
where
appliedUnitPropagate stateA stateB F0 decisionVars ==
  ∃ (uc::Clause) (ul::Literal).
    formulaEntailsClause (getF stateA) uc ∧
    (var ul) ∈ decisionVars ∪ vars F0 ∧
    isUnitClause uc ul (elements (getM stateA)) ∧
    getF stateB = getF stateA ∧
    getM stateB = getM stateA ⊕ [(ul, False)] ∧
    getConflictFlag stateB = getConflictFlag stateA ∧
    getC stateB = getC stateA

definition
applicableUnitPropagate :: State ⇒ Formula ⇒ Variable set ⇒ bool
where
applicableUnitPropagate state F0 decisionVars == ∃ state'. appliedUnitPropagate state state' F0 decisionVars

Notice, also, that Conflict can be performed for a clause that is not a member of the formula.

definition
appliedConflict :: State ⇒ State ⇒ bool
where
appliedConflict stateA stateB ==
  ∃ clause.
    getConflictFlag stateA = False ∧
    formulaEntailsClause (getF stateA) clause ∧
    clauseFalse clause (elements (getM stateA)) ∧
    getF stateB = getF stateA ∧
    getM stateB = getM stateA ∧
    getConflictFlag stateB = True ∧
    getC stateB = clause

definition
applicableConflict :: State ⇒ bool
where
applicableConflict state == ∃ state'. appliedConflict state state'

Notice, also, that the explanation can be done over a reason clause that is not a member of the formula, but is only entailed by it.

definition
appliedExplain :: State ⇒ State ⇒ bool
where
appliedExplain stateA stateB ==
  ∃ l reason.
  getConflictFlag stateA = True ∧
  l el getC stateA ∧
  formulaEntailsClause (getF stateA) reason ∧
  isReason reason (opposite l) (elements (getM stateA)) ∧
  getF stateB = getF stateA ∧
  getM stateB = getM stateA ∧
  getConflictFlag stateB = True ∧
  getC stateB = resolve (getC stateA) reason l

definition
appliedExplain :: State ⇒ bool
where
appliedExplain state == ∃ state'. appliedExplain state state'

definition
appliedLearn :: State ⇒ State ⇒ bool
where
appliedLearn stateA stateB ==
  getConflictFlag stateA = True ∧
  ¬ getC stateA el getF stateA ∧
  getF stateB = getF stateA @ [getC stateA] ∧
  getM stateB = getM stateA ∧
  getConflictFlag stateB = True ∧
  getC stateB = getC stateA

definition
appliedLearn :: State ⇒ bool
where
appliedLearn state == ∃ state'. appliedLearn state state'

Since unit propagation can be done over non-member clauses, it is not required that the conflict clause is learned before the Backjump is applied.
appliedBackjump :: State → State → bool

where

appliedBackjump stateA stateB ==

∃ l level.
getConflictFlag stateA = True ∧
isBackjumpLevel level l (getC stateA) (getM stateA) ∧

getF stateB = getF stateA ∧
getM stateB = prefixToLevel level (getM stateA) @(l, False) ∧
getConflictFlag stateB = False ∧
ggetC stateB = []

definition
applicableBackjump :: State → bool

where

applicableBackjump state == ∃ state'. appliedBackjump state state'

Solving starts with the initial formula, the empty trail and in non conflicting state.

definition
isInitialState :: State ⇒ Formula ⇒ bool

where

isInitialState state F0 ==

getF state = F0 ∧
ggetM state = [] ∧
ggetConflictFlag state = False ∧
ggetC state = []

Transitions are preformed only by using given rules.

definition
transition :: State ⇒ State ⇒ Formula ⇒ Variable set ⇒ bool

where

transition stateA stateB F0 decisionVars ==

appliedDecide stateA stateB decisionVars ∨
appliedUnitPropagate stateA stateB F0 decisionVars ∨
appliedConflict stateA stateB ∨
appliedExplain stateA stateB ∨
appliedLearn stateA stateB ∨
appliedBackjump stateA stateB

Transition relation is obtained by applying transition rules iteratively. It is defined using a reflexive-transitive closure.

definition
transitionRelation F0 decisionVars == (∪(stateA, stateB). transition stateA stateB F0 decisionVars) ^∗

Final state is one in which no rules apply

definition
isFinalState :: State \rightarrow Formula \rightarrow Variable set \rightarrow bool

where

isFinalState state F_0 decisionVars == \neg (\exists \text{ state}' \text{. transition state state}' F_0 decisionVars)

The following several lemmas establish conditions for applicability of different rules.

lemma applicableDecideCharacterization:
  fixes stateA::State
  shows applicableDecide stateA decisionVars =
  (\exists \ l.
    (\text{var } l) \in \text{decisionVars} \land
    \neg l \in \text{elements (getM stateA)} \land
    \neg \text{opposite } l \in \text{elements (getM stateA)})
  )

proof
  assume \ ?rhs
  then obtain \ l \ where
    *: (\text{var } l) \in \text{decisionVars} \neg l \in \text{elements (getM stateA)} \neg \text{opposite}
    l \in \text{elements (getM stateA)}

  unfolding applicableDecide-def
  by auto

  let \ ?stateB = stateA[ getM := (getM stateA) @ [(l, True)] ]

  from * have applicableDecide stateA \ ?stateB decisionVars

  unfolding appliedDecide-def
  by auto

  thus \ ?lhs

  unfolding applicableDecide-def
  by auto

next
  assume \ ?lhs
  then obtain \ stateB \ l

    where (\text{var } l) \in \text{decisionVars} \neg l \in \text{elements (getM stateA)}
    \neg \text{opposite } l \in \text{elements (getM stateA)}

    unfolding applicableDecide-def

    unfolding appliedDecide-def
    by auto

    thus \ ?rhs

    by auto

qed

lemma applicableUnitPropagateCharacterization:
  fixes stateA::State and \ F_0::Formula
  shows applicableUnitPropagate stateA F_0 decisionVars =
  (\exists \ (uc::Clause) \ (ul::Literal).
    \text{formulaEntailsClause (getF stateA) uc} \land
    (\text{var ul}) \in \text{decisionVars} \cup \text{vars F_0} \land
    \text{isUnitClause uc ul (elements (getM stateA))}
  )

  (is \ ?lhs = \ ?rhs)
proof
assume ?rhs
then obtain ul uc
where *
formulaEntailsClause (getF stateA) uc
(var ul) ∈ decisionVars ∪ vars F0
isUnitClause uc ul (elements (getM stateA))
unfolding applicableUnitPropagate-def
by auto
let ?stateB = stateA[ getM := getM stateA @ [(ul, False)] ]
from * have appliedUnitPropagate stateA ?stateB F0 decisionVars
unfolding appliedUnitPropagate-def
by auto
thus ?lhs
unfolding applicableUnitPropagate-def
by auto
next
assume ?lhs
then obtain stateB uc ul
where
formulaEntailsClause (getF stateA) uc
(var ul) ∈ decisionVars ∪ vars F0
isUnitClause uc ul (elements (getM stateA))
unfolding applicableUnitPropagate-def
unfolding applicableUnitPropagate-def
by auto
thus ?rhs
by auto
qed

lemma applicableBackjumpCharacterization:
fixes stateA::State
shows applicableBackjump stateA =
(∃ l level.
   getConflictFlag stateA = True ∧
   isBackjumpLevel level l (getC stateA) (getM stateA)
) (is ?lhs = ?rhs)
proof
assume ?rhs
then obtain l level
where *
getConflictFlag stateA = True
isBackjumpLevel level l (getC stateA) (getM stateA)
unfolding applicableBackjump-def
by auto
let ?stateB = stateA[ getM := prefixToLevel level (getM stateA) @ [(l, False)],
   getConflictFlag := False,
lemma applicableExplainCharacterization:

```plaintext
defines stateA :: State
shows applicableExplain stateA = 
(∃ l reason.
 getConflictFlag stateA = True ∧
 l el getC stateA ∧
 formulaEntailsClause (getF stateA) reason ∧
 isReason reason (opposite l) (elements (getM stateA))
)
(is ?lhs = ?rhs)

proof
assumes ?lhs
then obtain l reason
  where *
    getConflictFlag stateA = True ∧
    l el (getC stateA) ∧
    formulaEntailsClause (getF stateA) reason ∧
    isReason reason (opposite l) (elements (getM stateA))
  unfolding applicableExplain-def
  by auto
let ?stateB = stateA(getC := resolve (getC stateA) reason l ∅)
from * have applicableExplain stateA ?stateB
  unfolding applicableExplain-def
  by auto
thus ?lhs
  unfolding applicableExplain-def
  by auto
next
assumes ?lhs
then obtain stateB l reason
  where
```
getConflictFlag stateA = True
let getC stateA formulaEntailsClause (getF stateA) reason
isReason reason (opposite l) (elements (getM stateA))
unfolding applicableExplain-def
unfolding appliedExplain-def
by auto
thus ?rhs
by auto
qed

lemma applicableConflictCharacterization:
fixes stateA::State
shows applicableConflict stateA =
(\exists clause.
  getConflictFlag stateA = False \land
  formulaEntailsClause (getF stateA) clause \land
  clauseFalse clause (elements (getM stateA))) (is ? lhs = ? rhs)
proof
assume ?rhs
then obtain clause
where *:
  getConflictFlag stateA = False formulaEntailsClause (getF stateA)
  clause clauseFalse clause (elements (getM stateA))
unfolding applicableConflict-def
by auto
let ?stateB = stateA(\ getC := clause,
                        getConflictFlag := True )
from * have applicableConflict stateA ?stateB
unfolding applicableConflict-def
by auto
thus ?lhs
unfolding applicableConflict-def
by auto
next
assume ?lhs
then obtain stateB clause
where
  getConflictFlag stateA = False
  formulaEntailsClause (getF stateA) clause
  clauseFalse clause (elements (getM stateA))
unfolding applicableConflict-def
unfolding clauseFalse clause (elements (getM stateA))
by auto
thus ?rhs
by auto
qed

lemma applicableLearnCharacterization:
fixes stateA::State
shows applicableLearn stateA =
    (getConflictFlag stateA = True ∧
    ¬ getC stateA el getF stateA) (is ?lhs = ?rhs)

proof
assume ?rhs

hence ∗: getConflictFlag stateA = True ¬ getC stateA el getF stateA
    unfolding applicableLearn-def
    by auto

let ?stateB = stateA( getF := getF stateA @ [getC stateA])

from ∗ have appliedLearn stateA ?stateB
    unfolding appliedLearn-def
    by auto

thus ?lhs
    unfolding applicableLearn-def
    by auto

next
assume ?lhs
then obtain stateB where
    getConflictFlag stateA = True ¬ (getC stateA el (getF stateA)
    unfolding applicableLearn-def
    unfolding applicableLearn-def
    by auto

thus ?rhs
    by auto

qed

Final states are the ones where no rule is applicable.

lemma finalStateNonApplicable:
fixes state::State
shows isFinalState state F0 decisionVars =
    (¬ applicableDecide state decisionVars ∧
    ¬ applicableUnitPropagate state F0 decisionVars ∧
    ¬ applicableBackjump state ∧
    ¬ applicableLearn state ∧
    ¬ applicableConflict state ∧
    ¬ applicableExplain state)

unfolding isFinalState-def
unfolding transition-def
unfolding applicableDecide-def
unfolding applicableUnitPropagate-def
unfolding applicableBackjump-def
unfolding applicableLearn-def
unfolding applicableConflict-def
unfolding applicableExplain-def
by auto
7.2 Invariants

Invariants that are relevant for the rest of correctness proof.

\begin{verbatim}
definition invariantsHoldInState :: State ⇒ Formula ⇒ Variable set ⇒ bool where invariantsHoldInState state F0 decisionVars ==
  InvariantVarsM (getM state) F0 decisionVars ∧
  InvariantVarsF (getF state) F0 decisionVars ∧
  InvariantConsistent (getM state) ∧
  InvariantUniq (getM state) ∧
  InvariantReasonClauses (getF state) (getM state) ∧
  InvariantEquivalent F0 (getF state) ∧
  InvariantCFalse (getConflictFlag state) (getM state) (getC state)
∧
  InvariantCEntailed (getConflictFlag state) (getF state) (getC state)
\end{verbatim}

Invariants hold in initial states

\begin{verbatim}
lemma invariantsHoldInInitialState:
  \textbf{fixes} state :: State \textbf{and} F0 :: Formula
  \textbf{assumes} isInitialState state F0
  \textbf{shows} invariantsHoldInState state F0 decisionVars
  \textbf{using} \textbf{}\textbf{by} \textbf{(}auto simp add: isInitialState-def invariantsHoldInState-def InvariantVarsM-def InvariantVarsF-def InvariantConsistent-def InvariantUniq-def InvariantReasonClauses-def InvariantEquivalent-def equivalentFormulae-def InvariantCFalse-def InvariantCEntailed-def)\textbf{)}
\end{verbatim}

Valid transitions preserve invariants.

\begin{verbatim}
lemma transitionsPreserveInvariants:
  \textbf{fixes} stateA::State \textbf{and} stateB::State
  \textbf{assumes} transition stateA stateB F0 decisionVars \textbf{and}
  invariantsHoldInState stateA F0 decisionVars
  \textbf{shows} invariantsHoldInState stateB F0 decisionVars
  \textbf{proof−}
  \textbf{from} \textbf{(}invariantsHoldInState stateA F0 decisionVars\textbf{)} \textbf{have}
    InvariantVarsM (getM stateA) F0 decisionVars \textbf{and}
    InvariantVarsF (getF stateA) F0 decisionVars \textbf{and}
    InvariantConsistent (getM stateA) \textbf{and}
\end{verbatim}

268
InvariantUniq (getM stateA) and
InvariantReasonClauses (getF stateA) (getM stateA) and
InvariantEquivalent F0 (getF stateA) and
InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA) and
InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)

unfolding invariantsHoldInState-def
by auto
{
assume appliedDecide stateA stateB decisionVars
then obtain l::Literal where
(var l) ∈ decisionVars
¬ literalTrue l (elements (getM stateA))
¬ literalFalse l (elements (getM stateA))
getM stateB = getM stateA @ [(l, True)]
getF stateB = getF stateA
getConflictFlag stateB = getConflictFlag stateA
getC stateB = getC stateA

unfolding appliedDecide-def
by auto

from (¬ literalTrue l (elements (getM stateA))) (¬ literalFalse l (elements (getM stateA)))
have *: var l /∈ vars (elements (getM stateA))
using variableDefinedImpliesLiteralDefined[of l elements (getM stateA)]
by simp

have InvariantVarsM (getM stateB) F0 decisionVars
using (getF stateB = getF stateA)
⟨getM stateB = getM stateA @ [(l, True)]⟩
⟨InvariantVarsM (getM stateA) F0 decisionVars⟩
⟨var l ∈ decisionVars⟩
InvariantVarsMAfterDecide[of getM stateA F0 decisionVars l getM stateB]
by simp
moreover
have InvariantVarsF (getF stateB) F0 decisionVars
using (getF stateB = getF stateA)
⟨InvariantVarsF (getF stateA) F0 decisionVars⟩
by simp
moreover
have InvariantConsistent (getM stateB)
using (getM stateB = getM stateA @ [(l, True)])
⟨InvariantConsistent (getM stateA)⟩
⟨var l /∈ vars (elements (getM stateA))⟩
InvariantConsistentAfterDecide[of getM stateA l getM stateB]
by simp
moreover
have InvariantUniq (getM stateB)
  using (getM stateB = getM stateA @ [(l, True)])
  ⟨InvariantUniq (getM stateA)⟩
  ⟨var l ∉ vars (elements (getM stateA))⟩
  ⟨InvariantUniqAfterDecide[of getM stateA l getM stateB]⟩
  by simp
moreover
have InvariantReasonClauses (getF stateB) (getM stateB)
  using (getF stateB = getF stateA)
  ⟨getM stateB = getM stateA @ [(l, True)]⟩
  ⟨InvariantUniq (getM stateA)⟩
  ⟨InvariantReasonClauses (getF stateA) (getM stateA)⟩
  using InvariantReasonClausesAfterDecide[of getF stateA getM stateA getM stateB l]
  by simp
moreover
have InvariantEquivalent F0 (getF stateB)
  using (getF stateB = getF stateA)
  ⟨InvariantEquivalent F0 (getF stateA)⟩
  by simp
moreover
have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)
  using (getM stateB = getM stateA @ [(l, True)])
  ⟨getConflictFlag stateB = getConflictFlag stateA⟩
  ⟨getC stateB = getC stateA⟩
  ⟨InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)⟩
  ⟨InvariantCFalseAfterDecide[of getConflictFlag stateA getM stateA getC stateA getM stateB l]⟩
  by simp
moreover
have InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)
  using (getF stateB = getF stateA)
  ⟨getConflictFlag stateB = getConflictFlag stateA⟩
  ⟨getC stateB = getC stateA⟩
  ⟨InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)⟩
  by simp
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto
}
moreover
{
  assume appliedUnitPropagate stateA stateB F0 decisionVars
then obtain $uc :: Clause$ and $ul :: Literal$ where

$\text{formulaEntailsClause} (getF \text{ stateA}) \ uc$

$(\text{var} \ ul) \in \text{decisionVars} \cup \text{vars} \ F_0$

$\text{isUnitClause} \ uc \ ul \ (\text{elements} \ (\text{getM} \ \text{stateA}))$

$getF \ \text{stateB} = getF \ \text{stateA}$

$getM \ \text{stateB} = getM \ \text{stateA} @ [(\text{ul}, \text{False})]$

$get\text{ConflictFlag} \ \text{stateB} = get\text{ConflictFlag} \ \text{stateA}$

$getC \ \text{stateB} = getC \ \text{stateA}$

unfolding appliedUnitPropagate-def

by auto

from $(\text{isUnitClause} \ uc \ ul \ (\text{elements} \ (\text{getM} \ \text{stateA})))$

have $ul \in uc$

unfolding isUnitClause-def

by simp

from $(\text{var} \ ul \in \text{decisionVars} \cup \text{vars} \ F_0)$

have $\text{InvariantVarsM} (\text{getM} \ \text{stateB}) \ F_0 \ \text{decisionVars}$

using $(\text{getF} \ \text{stateB} = \text{getF} \ \text{stateA})$

$(\text{InvariantVarsM} \ (\text{getM} \ \text{stateA}) \ F_0 \ \text{decisionVars})$

$(\text{getM} \ \text{stateB} = \text{getM} \ \text{stateA} @ [(\text{ul}, \text{False})])$

InvariantVarsMAfterUnitPropagate[of \ (\text{getM} \ \text{stateA}) \ F_0 \ \text{decisionVars} \ \text{ul} \ \text{getM} \ \text{stateB}]$

by auto

moreover

have $\text{InvariantVarsF} (\text{getF} \ \text{stateB}) \ F_0 \ \text{decisionVars}$

using $(\text{getF} \ \text{stateB} = \text{getF} \ \text{stateA})$

$(\text{InvariantVarsF} \ (\text{getF} \ \text{stateA}) \ F_0 \ \text{decisionVars})$

by simp

moreover

have $\text{InvariantConsistent} \ (\text{getM} \ \text{stateB})$

using $(\text{InvariantConsistent} \ (\text{getM} \ \text{stateA}))$

$(\text{isUnitClause} \ uc \ ul \ (\text{elements} \ (\text{getM} \ \text{stateA})))$

$(\text{getM} \ \text{stateB} = \text{getM} \ \text{stateA} @ [(\text{ul}, \text{False})])$

InvariantConsistentAfterUnitPropagate[of \ (\text{getM} \ \text{stateA}) \ F_0 \ \text{decisionVars} \ \text{ul} \ \text{getM} \ \text{stateB}]$

by simp

moreover

have $\text{InvariantUniq} \ (\text{getM} \ \text{stateB})$

using $(\text{InvariantUniq} \ (\text{getM} \ \text{stateA}))$

$(\text{isUnitClause} \ uc \ ul \ (\text{elements} \ (\text{getM} \ \text{stateA})))$

$(\text{getM} \ \text{stateB} = \text{getM} \ \text{stateA} @ [(\text{ul}, \text{False})])$

InvariantUniqAfterUnitPropagate[of \ (\text{getM} \ \text{stateA}) \ F_0 \ \text{decisionVars} \ \text{ul} \ \text{getM} \ \text{stateB}]$

by simp

moreover

have $\text{InvariantReasonClauses} (\text{getF} \ \text{stateB}) \ (\text{getM} \ \text{stateB})$

using $(\text{getF} \ \text{stateB} = \text{getF} \ \text{stateA})$

$(\text{InvariantReasonClauses} \ (\text{getF} \ \text{stateA}) \ (\text{getM} \ \text{stateA}))$
\(\langle\text{isUnitClause utc ul (elements (getM stateA))}\rangle\)
\(\langle\text{getM stateB = getM stateA @ [(ul, False)]}\rangle\)
\(\langle\text{formulaEntailsClause (getF stateA) utc}\rangle\)
\(\text{InvariantReasonClausesAfterUnitPropagate[of getF stateA getM stateA utc ul getM stateB]}\)
\(\text{by simp}\)

\text{moreover}\n\text{have}\(\text{InvariantEquivalent F0 (getF stateB)}\)
\text{using}\(\langle\text{getF stateB = getF stateA}\rangle\)
\(\langle\text{InvariantEquivalent F0 (getF stateA)}\rangle\)
\text{by simp}\n
\text{moreover}\n\text{have}\(\text{InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)}\)
\text{using}\(\langle\text{getM stateB = getM stateA @ [(ul, False)]}\rangle\)
\(\langle\text{getConflictFlag stateB = getConflictFlag stateA}\rangle\)
\(\langle\text{getC stateB = getC stateA}\rangle\)
\(\langle\text{InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)}\rangle\)
\(\text{InvariantCFalseAfterUnitPropagate[of getConflictFlag stateA getM stateA getC stateA getM stateB ul]}\)
\text{by simp}\n
\text{moreover}\n\text{have}\(\text{InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)}\)
\text{using}\(\langle\text{getF stateB = getF stateA}\rangle\)
\(\langle\text{getConflictFlag stateB = getConflictFlag stateA}\rangle\)
\(\langle\text{getC stateB = getC stateA}\rangle\)
\(\langle\text{InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)}\rangle\)
\text{by simp}\n
\text{ultimately}\n\text{have}\(\text{?thesis}\)
\text{unfolding}\(\text{invariantsHoldInState-def}\)
\text{by auto}\n
\text{moreover}\n\{\n\text{assume}\(\text{appliedConflict stateA stateB}\)
\text{then obtain}\(\text{clause::Clause where}\)
\text{getConflictFlag stateA = False}\n\text{formulaEntailsClause (getF stateA) clause}\n\text{clauseFalse clause (elements (getM stateA))}\n\text{getF stateB = getF stateA}\n\text{getM stateB = getM stateA}\n\text{getConflictFlag stateB = True}\n\text{getC stateB = clause}\n\text{unfolding}\(\text{appliedConflict-def}\)
\text{by auto}\n\}
have InvariantVarsM (getM stateB) F0 decisionVars
  using (InvariantVarsM (getM stateA) F0 decisionVars)
  (getM stateB = getM stateA)
  by simp
moreover
have InvariantVarsF (getF stateB) F0 decisionVars
  using (InvariantVarsF (getF stateA) F0 decisionVars)
  (getF stateB = getF stateA)
  by simp
moreover
have InvariantConsistent (getM stateB)
  using (InvariantConsistent (getM stateA))
  (getM stateB = getM stateA)
  by simp
moreover
have InvariantUniq (getM stateB)
  using (InvariantUniq (getM stateA))
  (getM stateB = getM stateA)
  by simp
moreover
have InvariantReasonClauses (getF stateB) (getM stateB)
  using (InvariantReasonClauses (getF stateA) (getM stateA))
  (getF stateB = getF stateA)
  (getM stateB = getM stateA)
  by simp
moreover
have InvariantEquivalent F0 (getF stateB)
  using (InvariantEquivalent F0 (getF stateA))
  (getF stateB = getF stateA)
  by simp
moreover
have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)
  using
  (clauseFalse clause (elements (getM stateA)))
  (getM stateB = getM stateA)
  (getConflictFlag stateB = True)
  (getC stateB = clause)
  unfolding InvariantCFalse-def
  by simp
moreover
have InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)
  unfolding InvariantCEntailed-def
  using
  (getConflictFlag stateB = True)
  (formulaEntailsClause (getF stateA) clause)
  (getF stateB = getF stateA)
\( \langle \text{getC stateB} = \text{clause} \rangle \)
by simp
ultimately
have \( ?\text{thesis} \)
  unfolding \text{invariantsHoldInState-def} 
  by auto
}\nmoreover
{
  assume \text{appliedExplain stateA stateB} 
  then obtain \text{l::Literal and reason::Clause where} 
  \( \text{getConflictFlag stateA} = \text{True} \)
  \( \text{l el getC stateA} \)
  formulaEntailsClause (\text{getF stateA}) reason 
  isReason reason (\text{opposite l}) (\text{elements (getM stateA)}) 
  \text{getF stateB} = \text{getF stateA} 
  \text{getM stateB} = \text{getM stateA} 
  \text{getConflictFlag stateB} = \text{True} 
  \text{getC stateB} = \text{resolve (getC stateA) reason l} 
  unfolding \text{appliedExplain-def} 
  by auto

have \text{InvariantVarsM (getM stateB) F0 decisionVars} 
  using \( \langle \text{InvariantVarsM (getM stateA) F0 decisionVars} \rangle \)
  (\text{getM stateB} = \text{getM stateA}) 
  by simp
moreover
have \text{InvariantVarsF (getF stateB) F0 decisionVars} 
  using \( \langle \text{InvariantVarsF (getF stateA) F0 decisionVars} \rangle \)
  (\text{getF stateB} = \text{getF stateA}) 
  by simp
moreover
have \text{InvariantConsistent (getM stateB)} 
  using 
  (\text{getM stateB} = \text{getM stateA}) 
  (\text{InvariantConsistent (getM stateA)}) 
  by simp
moreover
have \text{InvariantUniq (getM stateB)} 
  using 
  (\text{getM stateB} = \text{getM stateA}) 
  (\text{InvariantUniq (getM stateA)}) 
  by simp
moreover
have \text{InvariantReasonClauses (getF stateB) (getM stateB)} 
  using 
  (\text{getF stateB} = \text{getF stateA}) 
  (\text{getM stateB} = \text{getM stateA}) 
  (\text{InvariantReasonClauses (getF stateA) (getM stateA)})
by simp
moreover
have InvariantEquivalent F0 (getF stateB)
  using
  ⟨getF stateB = getF stateA⟩
  ⟨InvariantEquivalent F0 (getF stateA)⟩
by simp
moreover
have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)
  using
  ⟨InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)⟩
  ⟨l el getC stateA⟩
  ⟨isReason reason (opposite l) (elements (getM stateA))⟩
  ⟨getM stateB = getM stateA⟩
  ⟨getC stateB = resolve (getC stateA) reason l⟩
  ⟨getConflictFlag stateA = True⟩
  ⟨getConflictFlag stateB = True⟩
  ⟨InvariantCFalseAfterExplain[of getConflictFlag stateA getM stateA getC stateA opposite l reason getC stateB]⟩
by simp
moreover
have InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)
  using
  ⟨InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)⟩
  ⟨l el getC stateA⟩
  ⟨isReason reason (opposite l) (elements (getM stateA))⟩
  ⟨getF stateB = getF stateA⟩
  ⟨getC stateB = resolve (getC stateA) reason l⟩
  ⟨getConflictFlag stateA = True⟩
  ⟨getConflictFlag stateB = True⟩
  ⟨formulaEntailsClause (getF stateA) reason⟩
  ⟨InvariantCEntailedAfterExplain[of getConflictFlag stateA getF stateA getC stateA reason getC stateB opposite l]⟩
by simp
moreover
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto
}
moreover
{
  assume appliedLearn stateA stateB
  hence
    getConflictFlag stateA = True

275
\[ \neg \text{getC stateA} \implies \text{getF stateA} \]
\[ \text{getF stateB} = \text{getF stateA} \oplus \text{getC stateA} \]
\[ \text{getM stateB} = \text{getM stateA} \]
\[ \text{getConflictFlag stateB} = \text{True} \]
\[ \text{getC stateB} = \text{getC stateA} \]
\[ \text{unfolding appliedLearn-def} \]
\[ \text{by auto} \]

\[ \text{from (getConflictFlag stateA = True \implies \text{InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA))}} \]
\[ \text{have formulaEntailsClause (getF stateA) (getC stateA)} \]
\[ \text{unfolding InvariantCEntailed-def} \]
\[ \text{by simp} \]

\[ \text{have InvariantVarsM (getM stateB) F0 decisionVars} \]
\[ \text{using (InvariantVarsM (getM stateA) F0 decisionVars)} \]
\[ \langle \text{getM stateB} = \text{getM stateA} \rangle \]
\[ \text{by simp} \]
\[ \text{moreover} \]
\[ \text{from (InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)) (getConflictFlag stateA = True)} \]
\[ \text{have clauseFalse (getC stateA) (elements (getM stateA))} \]
\[ \text{unfolding InvariantCFalse-def} \]
\[ \text{by simp} \]
\[ \text{with (InvariantVarsM (getM stateA) F0 decisionVars)} \]
\[ \text{have (vars (getC stateA)) \subseteq vars F0 \cup decisionVars} \]
\[ \text{unfolding InvariantVarsM-def} \]
\[ \text{using valuationContainsItsFalseClausesVariables[of getC stateA elements (getM stateA)]} \]
\[ \text{by simp} \]
\[ \text{hence InvariantVarsF (getF stateB) F0 decisionVars} \]
\[ \text{using (getF stateB = getF stateA \oplus \text{getC stateA})} \]
\[ \langle \text{InvariantVarsF (getF stateA) F0 decisionVars} \rangle \]
\[ \text{InvariantVarsFAfterLearn [of getF stateA F0 decisionVars getC stateA getF stateB]} \]
\[ \text{by simp} \]
\[ \text{moreover} \]
\[ \text{have InvariantConsistent (getM stateB)} \]
\[ \text{using (InvariantConsistent (getM stateA))} \]
\[ \langle \text{getM stateB} = \text{getM stateA} \rangle \]
\[ \text{by simp} \]
\[ \text{moreover} \]
\[ \text{have InvariantUniq (getM stateB)} \]
\[ \text{using (InvariantUniq (getM stateA))} \]
\[ \langle \text{getM stateB} = \text{getM stateA} \rangle \]
\[ \text{by simp} \]
\[ \text{moreover} \]
\[ \text{have InvariantReasonClauses (getF stateB) (getM stateB)} \]
\[ \text{using} \]

276
\textbf{(InvariantReasonClauses (getF stateA) (getM stateA))}

\textbf{(formulaEntailsClause (getF stateA) (getC stateA))}

\textbf{(getF stateB = getF stateA @ [getC stateA])}

\textbf{(getM stateB = getM stateA)}

\textbf{InvariantReasonClausesAfterLearn[of getF stateA getM stateA getC stateA getF stateB]}

\textbf{by simp}

\textbf{moreover}

\textbf{have InvariantEquivalent F0 (getF stateB)}

\textbf{using}

\textbf{(InvariantEquivalent F0 (getF stateA))}

\textbf{(formulaEntailsClause (getF stateA) (getC stateA))}

\textbf{(getF stateB = getF stateA @ [getC stateA])}

\textbf{InvariantEquivalentAfterLearn[of F0 getF stateA getC stateA getF stateB]}

\textbf{by simp}

\textbf{moreover}

\textbf{have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)}

\textbf{using (InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA))}

\textbf{(getM stateB = getM stateA)}

\textbf{(getConflictFlag stateA = True)}

\textbf{(getConflictFlag stateB = True)}

\textbf{(getM stateB = getM stateA)}

\textbf{(getC stateB = getC stateA)}

\textbf{by simp}

\textbf{moreover}

\textbf{have InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)}

\textbf{using (InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA))}

\textbf{(formulaEntailsClause (getF stateA) (getC stateA))}

\textbf{(getF stateB = getF stateA @ [getC stateA])}

\textbf{(getConflictFlag stateA = True)}

\textbf{(getConflictFlag stateB = True)}

\textbf{(getC stateB = getC stateA)}

\textbf{InvariantCEntailedAfterLearn[of getConflictFlag stateA getF stateA getC stateA getF stateB]}

\textbf{by simp}

\textbf{ultimately}

\textbf{have ?thesis}

\textbf{unfolding invariantsHoldInState-def}

\textbf{by auto}

}\}

\textbf{moreover}

\{\textbf{assume appliedBackjump stateA stateB}\}
then obtain $l::\text{Literal}$ and $\text{level}::\text{nat}$

where

$\text{getConflictFlag stateA} = \text{True}$

$\text{isBackjumpLevel level \ (getC stateA) (getM stateA)}$

$\text{getF stateB} = \text{getF stateA}$

$\text{getM stateB} = \text{prefixToLevel level (getM stateA)} @ [(l, \text{False})]$  

$\text{getConflictFlag stateB} = \text{False}$

$\text{getC stateB} = []$

**unfolding** appliedBackjump-def

by auto

with ($\text{InvariantConsistent (getM stateA)}$) ($\text{InvariantUniq (getM stateA)}$)

($\text{InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)}$)

have $\text{isUnitClause (getC stateA) l (elements (prefixToLevel level (getM stateA)))}$

**unfolding** InvariantUniq-def

**unfolding** InvariantConsistent-def

**unfolding** InvariantCFalse-def

using $\text{isBackjumpLevelEnsuresIsUnitInPrefix[of getM stateA getC stateA level l]}$

by simp

from ($\text{getConflictFlag stateA} = \text{True}$) ($\text{InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)}$)

have $\text{formulaEntailsClause (getF stateA) (getC stateA)}$

**unfolding** InvariantCEntailed-def

by simp

from ($\text{isBackjumpLevel level l (getC stateA) (getM stateA)}$)

have $\text{isLastAssertedLiteral (opposite l) (oppositeLiteralList (getC stateA)) (elements (getM stateA))}$

**unfolding** isBackjumpLevel-def

by simp

hence $l \ \text{el getC stateA}$

**unfolding** isLastAssertedLiteral-def

using literalElListIffOppositeLiteralElOppositeLiteralList[of l getC stateA]

by simp

have $\text{isPrefix (prefixToLevel level (getM stateA)) (getM stateA)}$

by (simp add:isPrefixPrefixToLevel)

from ($\text{getConflictFlag stateA} = \text{True}$) ($\text{InvariantCEntailed (getConflictFlag stateA) (getF stateA) (getC stateA)}$)

have $\text{formulaEntailsClause (getF stateA) (getC stateA)}$

**unfolding** InvariantCEntailed-def

by simp

278
from ⟨getConflictFlag stateA = True⟩ ⟨InvariantCFalse (getConflictFlag stateA) (getM stateA) (getC stateA)⟩
  have clauseFalse (getC stateA) (elements (getM stateA))
    unfolding InvariantCFalse-def
    by simp
  hence vars (getC stateA) ⊆ vars (elements (getM stateA))
  using valuationContainsItsFalseClausesVariables[of getM stateA elements (getM stateA)]
  by simp
moreover
from ⟨el getC stateA⟩
  have var l ∈ vars (getC stateA)
    unfolding InvariantVars-def
    by auto
have InvariantVarsM (getM stateB) F0 decisionVars
  using ⟨InvariantVarsM (getM stateA) F0 decisionVars⟩
  ⟨isUnitClause (getC stateA) l elements (prefixToLevel level (getM stateA))⟩
  ⟨isPrefix (prefixToLevel level (getM stateA)) (getM stateA)⟩
  ⟨var l ∈ vars F0 ∪ decisionVars⟩
  ⟨formulaEntailsClause (getF stateA) (getC stateA)⟩
  ⟨getF stateB = getF stateA⟩
  ⟨getM stateB = prefixToLevel level (getM stateA) @ [(l, False)]⟩
  unfolding InvariantVarsMAfterBackjump[of getM stateA F0 decisionVars prefixToLevel level (getM stateA) l getM stateB]
  by simp
moreover
have InvariantVarsF (getF stateB) F0 decisionVars
  using ⟨InvariantVarsF (getF stateA) F0 decisionVars⟩
  ⟨getF stateB = getF stateA⟩
  by simp
moreover
have InvariantConsistent (getM stateB)
  using ⟨InvariantConsistent (getM stateA)⟩
  ⟨isUnitClause (getC stateA) l elements (prefixToLevel level (getM stateA))⟩
  ⟨isPrefix (prefixToLevel level (getM stateA)) (getM stateA)⟩
  ⟨getM stateB = prefixToLevel level (getM stateA) @ [(l, False)]⟩
  unfolding InvariantConsistentAfterBackjump[of getM stateA prefixToLevel level (getM stateA) getC stateA l getM stateB]
  by simp
moreover
have InvariantUniq (getM stateB)
using ⟨InvariantUniq (getM stateA))
  ⟨isUnitClause (getC stateA) l (elements (prefixToLevel level (getM stateA)))⟩
  ⟨isPrefix (prefixToLevel level (getM stateA)) (getM stateA)⟩
  (getM stateB = prefixToLevel level (getM stateA) @ [l, False])
  InvariantUniqAfterBackjump[of getM stateA prefixToLevel level (getM stateA) getC stateA l getM stateB]
  by simp

moreover
have InvariantReasonClauses (getF stateB) (getM stateB)
  using ⟨InvariantUniq (getM stateA)⟩ ⟨InvariantReasonClauses (getF stateA) (getM stateA)⟩
  ⟨isUnitClause (getC stateA) l (elements (prefixToLevel level (getM stateA)))⟩
  ⟨isPrefix (prefixToLevel level (getM stateA)) (getM stateA)⟩
  ⟨formulaEntailsClause (getF stateA) (getC stateA)⟩
  ⟨getF stateB = getF stateA⟩
  (getM stateB = prefixToLevel level (getM stateA) @ [l, False])
  InvariantReasonClausesAfterBackjump[of getF stateA getM stateA]
  prefixToLevel level (getM stateA) getC stateA l getM stateB]
  by simp

moreover
have InvariantEquivalent F0 (getF stateB)
  using ⟨InvariantEquivalent F0 (getF stateA)⟩
  ⟨getF stateB = getF stateA⟩
  by simp

moreover
have InvariantCFalse (getConflictFlag stateB) (getM stateB) (getC stateB)
  using ⟨getConflictFlag stateB = False⟩
  unfolding InvariantCFalse-def
  by simp

moreover
have InvariantCEntailed (getConflictFlag stateB) (getF stateB) (getC stateB)
  using ⟨getConflictFlag stateB = False⟩
  unfolding InvariantCEntailed-def
  by simp

moreover
ultimately
have ?thesis
  unfolding invariantsHoldInState-def
  by auto

} ultimately

show ?thesis
using ⟨transition stateA stateB F0 decisionVars⟩

280
The consequence is that invariants hold in all valid runs.

**lemma** invariantsHoldInValidRuns:

fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars} :: \text{Variable set} \)

assumes invariantsHoldInState stateA \( F_0 \) decisionVars and

\((stateA, stateB) \in \text{transitionRelation} \ F_0 \ \text{decisionVars}\)

shows invariantsHoldInState stateB \( F_0 \) decisionVars

using assms

using transitionsPreserveInvariants

using rtrancl-induct[of stateA stateB

\{(stateA, stateB), \text{transition stateA stateB} \ F_0 \ \text{decisionVars}\} \ \lambda \ x. \ invariantsHoldInState x \ F_0 \ \text{decisionVars}\]

**unfolding** transitionRelation-def

by auto

**lemma** invariantsHoldInValidRunsFromInitialState:

fixes \( F_0 :: \text{Formula} \) and \( \text{decisionVars} :: \text{Variable set} \)

assumes isInitialState state0 \( F_0 \)

and \((state0, state) \in \text{transitionRelation} \ \ F_0 \ \text{decisionVars}\)

shows invariantsHoldInState state \( F_0 \) decisionVars

proof–

from \(\text{isInitialState state0} \ F_0\)

have invariantsHoldInState state0 \( F_0 \) decisionVars

by (simp add: invariantsHoldInInitialState)

with assms

show ?thesis

using invariantsHoldInValidRuns [of state0 \( F_0 \) decisionVars state]

by simp

qed

In the following text we will show that there are two kinds of states:

1. **UNSAT** states where \(\text{getConflictFlag} \ \text{state} = \text{True} \) and \(\text{getC} \ \text{state} = \text{[]}\).

2. **SAT** states where \(\text{getConflictFlag} \ \text{state} = \text{False} \), \(\neg \) formula\( F_0 \) (elements (getM state)) and decisionVars \(\subseteq\) vars (elements (getM state)).

The soundness theorems claim that if **UNSAT** state is reached the formula is unsatisfiable and if **SAT** state is reached, the formula is satisfiable.

Completeness theorems claim that every final state is either **UNSAT** or **SAT**. A consequence of this and soundness theorems, is that if formula is unsatisfiable the solver will finish in an **UNSAT**
state, and if the formula is satisfiable the solver will finish in a SAT state.

7.3 Soundness

**Theorem soundnessForUNSAT:**

fixes $F_0 :: \text{Formula}$ and $\text{decisionVars :: Variable set}$ and $\text{state0 :: State}$ and $\text{state :: State}$

assumes

`isInitialState state0 F0` and

`(state0, state) \in \text{transitionRelation F0 decisionVars`}

`getConflictFlag state = True` and

`getC state = []`

shows $\neg \text{satisfiable F0}$

**Proof**

- from `(\text{isInitialState state0 F0: (state0, state) \in transitionRelation F0 decisionVars})`

  have `\text{invariantsHoldInState state F0 decisionVars}`

  using `\text{invariantsHoldInValidRunsFromInitialState}`

  by simp

  hence

  `\text{InvariantEquivalence F0 (getF state)}`

  `\text{InvariantCEntailed (getConflictFlag state) (getF state) (getC state)}`

  unfolding `\text{invariantsHoldInState-def}`

  by auto

  with `(getConflictFlag state = True: (getC state = []))`

  `\text{show ?thesis}`

  by `(simp add: unsatReportExtensiveExplain)`

**Qed**

**Theorem soundnessForSAT:**

fixes $F_0 :: \text{Formula}$ and $\text{decisionVars :: Variable set}$ and $\text{state0 :: State}$ and $\text{state :: State}$

assumes

`\text{vars F0 \subseteq decisionVars}` and

`isInitialState state0 F0` and

`(state0, state) \in \text{transitionRelation F0 decisionVars}` and

`\text{getConflictFlag state = False}`

`\neg \text{formulaFalse (getF state) (elements (getM state))}`

`\text{vars (elements (getM state)) \supseteq decisionVars}`

shows

`\text{model (elements (getM state)) F0}`

**Proof**

- from `(\text{isInitialState state0 F0: (state0, state) \in transitionRelation F0 decisionVars})`

  have `\text{invariantsHoldInState state F0 decisionVars}`

282
using invariantsHoldInValidRunsFromInitialState
by simp

hence
InvariantConsistent (getM state)
InvariantEquivalent F0 (getF state)
InvariantVarsF (getF state) F0 decisionVars

unfolding invariantsHoldInState-def
by auto
with assms
show ?thesis
using satReport[of F0 decisionVars getF state getM state]
by simp
qed

7.4 Termination

We now define a termination ordering which is a lexicographic combination of lexLessRestricted trail ordering, boolLess conflict flag ordering, multLess conflict clause ordering and learnLess formula ordering. This ordering will be central in termination proof.

definition lexLessState (F0::Formula) decisionVars == {((stateA::State), (stateB::State)).
  (getM stateA, getM stateB) ∈ lexLessRestricted (vars F0 ∪ decisionVars)}
definition boolLessState == {((stateA::State), (stateB::State)).
  getM stateA = getM stateB ∧
  (getConflictFlag stateA, getConflictFlag stateB) ∈ boolLess}
definition multLessState == {((stateA::State), (stateB::State)).
  getM stateA = getM stateB ∧
  getConflictFlag stateA = getConflictFlag stateB ∧
  (getC stateA, getC stateB) ∈ multLess (getM stateA)}
definition learnLessState == {((stateA::State), (stateB::State)).
  getM stateA = getM stateB ∧
  getConflictFlag stateA = getConflictFlag stateB ∧
  getC stateA = getC stateB ∧
  (getF stateA, getF stateB) ∈ learnLess (getC stateA)}

definition terminationLess F0 decisionVars == {((stateA::State), (stateB::State)).
  (stateA, stateB) ∈ lexLessState F0 decisionVars ∨
  (stateA, stateB) ∈ boolLessState ∨
  (stateA, stateB) ∈ multLessState ∨
  (stateA, stateB) ∈ learnLessState}

We want to show that every valid transition decreases a state with respect to the constructed termination ordering.

First we show that Decide, UnitPropagate and Backjump rule decrease the trail with respect to the restricted trail ordering.

283
lexLessRestricted. Invariants ensure that trails are indeed uniq, consistent and with finite variable sets.

**lemma** trailsDecreasedByDecidedUnitPropagateAndBackjump:
  *fixes* stateA::State and stateB::State
  *assumes* invariantsHoldInState stateA F0 decisionVars and
  appliedDecide stateA stateB decisionVars ∨ appliedUnitPropagate stateA stateB F0 decisionVars ∨ appliedBackjump stateA stateB
  *shows* (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)

**proof**
  *from* appliedDecide stateA stateB decisionVars ∨ appliedUnitPropagate stateA stateB F0 decisionVars ∨ appliedBackjump stateA stateB
  *have* invariantsHoldInState stateB F0 decisionVars
    using transitionsPreserveInvariants
    unfolding transition-def
    by auto
  *from* invariantsHoldInState stateA F0 decisionVars
  have *: uniq (elements (getM stateA)) consistent (elements (getM stateA)) vars (elements (getM stateA)) ⊆ vars F0 ∪ decisionVars
    unfolding invariantsHoldInState-def
    unfolding InvariantVarsM-def
    unfolding InvariantConsistent-def
    unfolding InvariantUniq-def
    by auto
  *from* invariantsHoldInState stateB F0 decisionVars
  have **: uniq (elements (getM stateB)) consistent (elements (getM stateB)) vars (elements (getM stateB)) ⊆ vars F0 ∪ decisionVars
    unfolding invariantsHoldInState-def
    unfolding InvariantVarsM-def
    unfolding InvariantConsistent-def
    unfolding InvariantUniq-def
    by auto

  {
    *assume* appliedDecide stateA stateB decisionVars
    hence (getM stateB, getM stateA) ∈ lexLess
      unfolding appliedDecide-def
      by (auto simp add:lexLessAppend)
    with * **
    *have* (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)
      unfolding lexLessRestricted-def
      by auto
  }

  moreover

  {
    *assume* appliedUnitPropagate stateA stateB F0 decisionVars
    hence (getM stateB, getM stateA) ∈ lexLess
      unfolding appliedUnitPropagate-def
  }

  284
by (auto simp add: lexLessAppend)
with * **
have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪
decisionVars)
  unfolding lexLessRestricted-def
by auto
}

moreover
{
assume appliedBackjump stateA stateB
then obtain l::Literal and level::nat
  where
  getConflictFlag stateA = True
  isBackjumpLevel level l (getC stateA) (getM stateA)
  getF stateB = getF stateA
  getM stateB = prefixToLevel level (getM stateA) @ [(l, False)]
  getConflictFlag stateB = False
  getC stateB = []
  unfolding appliedBackjump-def
  by auto

  from (isBackjumpLevel level l (getC stateA) (getM stateA))
  have isLastAssertedLiteral (opposite l) (oppositeLiteralList (getC
  stateA)) (elements (getM stateA))
    unfolding isBackjumpLevel-def
    by simp
  hence (opposite l) el elements (getM stateA)
  unfolding isLastAssertedLiteral-def
  by simp
  hence elementLevel (opposite l) (getM stateA) <= currentLevel
  (getM stateA)
    by (simp add: elementLevelLeqCurrentLevel)
  moreover
  from (isBackjumpLevel level l (getC stateA) (getM stateA))
  have θ ≤ level and level < elementLevel (opposite l) (getM stateA)
    unfolding isBackjumpLevel-def
    using (isLastAssertedLiteral (opposite l) (oppositeLiteralList (getC
    stateA)) (elements (getM stateA))):
    by auto
  ultimately
  have level < currentLevel (getM stateA)
    by simp
  with 0 ≤ level: (getM stateB = prefixToLevel level (getM stateA)
  @(l, False))
  have (getM stateB, getM stateA) ∈ lexLess
    by (simp add: lexLessBackjump)
  with * **
  have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪
\texttt{decisionVars})

\begin{verbatim}
    unfolding \texttt{lexLessRestricted-def}
    by \texttt{auto}
\end{verbatim}

\{ ultimately
  show \texttt{thesis}
  using \texttt{assms}
  by \texttt{auto}
\}

\texttt{qed}

Next we show that \texttt{Conflict} decreases the conflict flag in the \texttt{boolLess} ordering.

\textbf{lemma} \texttt{conflictFlagIsDecreasedByConflict}:
\begin{verbatim}
  fixes \texttt{stateA::State and stateB::State}
  assumes \texttt{appliedConflict stateA stateB}
  shows \texttt{getM stateA = getM stateB and (getConflictFlag stateB, getConflictFlag stateA) ∈ boolLess}
  using \texttt{assms}
  unfolding \texttt{appliedConflict-def}
  unfolding \texttt{boolLess-def}
  by \texttt{auto}
\end{verbatim}

Next we show that \texttt{Explain} decreases the conflict clause with respect to the \texttt{multLess} clause ordering.

\textbf{lemma} \texttt{conflictClauseIsDecreasedByExplain}:
\begin{verbatim}
  fixes \texttt{stateA::State and stateB::State}
  assumes \texttt{appliedExplain stateA stateB}
  shows \texttt{getM stateA = getM stateB and getConflictFlag stateA = getConflictFlag stateB and (getC stateB, getC stateA) ∈ multLess (getM stateA)}
  proof
    from \texttt{(appliedExplain stateA stateB)}
    obtain \texttt{l::Literal and reason::Clause where}
      \texttt{getConflictFlag stateA = True}
      \texttt{l ∈ (getC stateA)}
      \texttt{isReason reason (opposite l) (elements (getM stateA))}
      \texttt{getF stateB = getF stateA}
      \texttt{getM stateB = getM stateA}
      \texttt{getConflictFlag stateB = True}
      \texttt{getC stateB = resolve (getC stateA) reason l}
      unfolding \texttt{appliedExplain-def}
    by \texttt{auto}
    thus \texttt{getM stateA = getM stateB getConflictFlag stateA = getConflictFlag stateB (getC stateB, getC stateA) ∈ multLess (getM stateA)}
    using \texttt{multLessResolve[of opposite l getC stateA reason getM stateA]}
    by \texttt{auto}
  qed
\end{verbatim}
Finally, we show that \textsl{Learn} decreases the formula in the learn-Less formula ordering.

\textbf{lemma \textit{formulaIsDecreasedByLearn}:}
\begin{itemize}
\item \textit{fixes stateA::State and stateB::State}
\item \textit{assumes \textit{appliedLearn stateA stateB}}
\item \textit{shows}
\begin{itemize}
\item getM stateA = getM stateB and
\item getConflictFlag stateA = getConflictFlag stateB and
\item getC stateA = getC stateB and
\item (getF stateB, getF stateA) ∈ learnLess (getC stateA)
\end{itemize}
\end{itemize}

\textit{proof}–
\begin{itemize}
\item \textit{from (appliedLearn stateA stateB)}
\item \textit{have}
\begin{itemize}
\item getConflictFlag stateA = True
\item − getC stateA el getF stateA
\item getF stateB = getF stateA @ [getC stateA]
\item getM stateB = getM stateA
\item getConflictFlag stateB = True
\item getC stateB = getC stateA
\end{itemize}
\item \textit{unfolding \textit{appliedLearn-def}}
\item \textit{by auto}
\item \textit{thus}
\begin{itemize}
\item getM stateA = getM stateB
\item getConflictFlag stateA = getConflictFlag stateB
\item getC stateA = getC stateB
\item (getF stateB, getF stateA) ∈ learnLess (getC stateA)
\end{itemize}
\item \textit{unfolding \textit{learnLess-def}}
\item \textit{by auto}
\item \textit{qed}
\end{itemize}

Now we can prove that every rule application decreases a state with respect to the constructed termination ordering.

\textbf{lemma \textit{stateIsDecreasedByValidTransitions}:}
\begin{itemize}
\item \textit{fixes stateA::State and stateB::State}
\item \textit{assumes \textit{invariantsHoldInState stateA F0 decisionVars and transition stateA stateB F0 decisionVars}}
\item \textit{shows (stateB, stateA) ∈ \textit{terminationLess F0 decisionVars}}
\end{itemize}

\textit{proof}–
\begin{itemize}
\item \textit{assume \textit{appliedDecide stateA stateB decisionVars} ∨ \textit{appliedUnitPropagate stateA stateB F0 decisionVars} ∨ \textit{appliedBackjump stateA stateB}}
\item \textit{with (invariantsHoldInState stateA F0 decisionVars)}
\item \textit{have (getM stateB, getM stateA) ∈ lexLessRestricted (vars F0 ∪ decisionVars)}
\item \textit{using \textit{trailIsDecreasedByDeciedUnitPropagateAndBackjump}}
\item \textit{by simp}
\item \textit{hence (stateB, stateA) ∈ lexLessState F0 decisionVars}
\item \textit{unfolding \textit{lexLessState-def}}
\end{itemize}
by simp
hence \((stateB, stateA) \in \text{terminationLess } F0 \text{ decisionVars}\)
  unfolding \text{terminationLess-def}
  by simp\}
}
moreover
\{
  assume appliedConflict stateA stateB
  hence getM stateA = getM stateB (getConflictFlag stateB, getConflictFlag stateA) ∈ boolLess
    using conflictFlagIsDecreasedByConflict
    by auto
  hence \((stateB, stateA) \in \text{boolLessState}\)
    unfolding \text{boolLessState-def}
    by simp
  hence \((stateB, stateA) \in \text{terminationLess } F0 \text{ decisionVars}\)
    unfolding \text{terminationLess-def}
    by simp\}
}
moreover
\{
  assume appliedExplain stateA stateB
  hence getM stateA = getM stateB
    getConflictFlag stateA = getConflictFlag stateB
    (getC stateB, getC stateA) ∈ multLess (getM stateA)
    using conflictClauseIsDecreasedByExplain
    by auto
  hence \((stateB, stateA) \in \text{multLessState}\)
    unfolding \text{multLessState-def}
    unfolding \text{multLess-def}
    by simp
  hence \((stateB, stateA) \in \text{terminationLess } F0 \text{ decisionVars}\)
    unfolding \text{terminationLess-def}
    by simp\}
}
moreover
\{
  assume appliedLearn stateA stateB
  hence getM stateA = getM stateB
    getConflictFlag stateA = getConflictFlag stateB
    getC stateA = getC stateB
    (getF stateB, getF stateA) ∈ learnLess (getC stateA)
    using formulaIsDecreasedByLearn
    by auto
  hence \((stateB, stateA) \in \text{learnLessState}\)
    unfolding \text{learnLessState-def}
    by simp
  hence \((stateB, stateA) \in \text{terminationLess } F0 \text{ decisionVars}\)
The minimal states with respect to the termination ordering are final i.e., no further transition rules are applicable.

**Definition**

\[
\text{isMinimalState} \; \text{stateMin} \; F0 \; \text{decisionVars} \equiv (\forall \; \text{state}::\text{State}. \; (\text{state}, \text{stateMin}) \notin \text{terminationLess} \; F0 \; \text{decisionVars})
\]

**Lemma** minimalStatesAreFinal:

fixes stateA::State

assumes invariantsHoldInState state F0 decisionVars and isMinimalState state F0 decisionVars

shows isFinalState state F0 decisionVars

proof

\{  
assume ¬ thesis  
then obtain state'::State  
where transition state state' F0 decisionVars  
unfolding isFinalState-def  
by auto  
with (invariantsHoldInState state F0 decisionVars)  
have (state', state) ∈ terminationLess F0 decisionVars  
using stateIsDecreasedByValidTransitions[of state F0 decisionVars state']  
unfolding transition-def  
by auto  
with (isMinimalState state F0 decisionVars)  
have False  
unfolding isMinimalState-def  
by auto  
\}  
thus thesis  
by auto

qed

We now prove that termination ordering is well founded. We start with several auxiliary lemmas, one for each component of the termination ordering.

**Lemma** wfLexLessState:

fixes decisionVars :: Variable set and F0 :: Formula
assumes finite decisionVars
shows wf (lexLessState F0 decisionVars)
unfolding wf-eq-minimal
proof –
  show ∀ Q state. state ∈ Q → (∃ stateMin ∈ Q. ∀ state'. (state', stateMin) ∈ lexLessState F0 decisionVars → state' ∉ Q)
  proof –
  {  
    fix Q :: State set and state :: State  
    assume state ∈ Q  
    let ?Q1 = {M::LiteralTrail. ∃ state. state ∈ Q ∧ (getM state)
      = M}  
    from (state ∈ Q)
    have getM state ∈ ?Q1  
      by auto  
    from (finite decisionVars)
    have finite {vars F0 ∪ decisionVars}
      using finiteVarsFormula[of F0]
      by simp
    hence wf (lexLessRestricted {vars F0 ∪ decisionVars})
      using wfLexLessRestricted[of vars F0 ∪ decisionVars]
      by simp
    with {getM state ∈ ?Q1}
    obtain Mmin where Mmin ∈ ?Q1 ∀ M'. (M', Mmin) ∈ lexLessRestricted {vars F0 ∪ decisionVars} → M' ∉ ?Q1
      unfolding wf-eq-minimal
      apply (erule-tac x=?Q1 in allE)
      apply (erule-tac x=getM state in allE)
      by auto
    from (Mmin ∈ ?Q1) obtain stateMin
      where stateMin ∈ Q (getM stateMin) = Mmin
      by auto
    have ∀ state'. (state', stateMin) ∈ lexLessState F0 decisionVars  
      → state' ∉ Q
    proof
      fix state'
      show (state', stateMin) ∈ lexLessState F0 decisionVars → state' ∉ Q
    proof
      assume (state', stateMin) ∈ lexLessState F0 decisionVars
      hence (getM state', getM stateMin) ∈ lexLessRestricted {vars F0 ∪ decisionVars}
        unfolding lexLessState-def
        by auto
      from ∀ M'. (M', Mmin) ∈ lexLessRestricted {vars F0 ∪ decisionVars} → M' ∉ ?Q1
        (getM state', getM stateMin) ∈ lexLessRestricted {vars F0 ∪ decisionVars} (getM stateMin = Mmin)
      have getM state' ∉ ?Q1
      by auto
  }

290
by simp
with \( \text{getM stateMin} = M_{\text{min}} \)
show \( \text{state'} \notin Q \)
by auto
qed

with \( \text{stateMin} \in Q \)

have \( \exists \text{stateMin} \in Q. \ (\forall \text{state}'. \ (\text{state}', \text{stateMin}) \in \text{lexLessState} \rightarrow \text{state'} \notin Q) \)
by auto
\}
thus \(?\text{thesis}\)
by auto
qed

lemma \( \text{wfBoolLessState} \):
shows \( \text{wf boolLessState} \)
unfolding \( \text{wf-eq-minimal} \)
proof—
show \( \forall Q. \ \text{state} \in Q \rightarrow (\exists \text{stateMin} \in Q. \ (\forall \text{state}'. \ (\text{state}', \text{stateMin}) \in \text{boolLessState} \rightarrow \text{state'} \notin Q) \)
proof—
{ 
fix \( Q :: \text{State set} \) and \( \text{state} :: \text{State} \)
assume \( \text{state} \in Q \)
let \( ?M = (\text{getM state}) \)
let \( ?Q1 = \{ b::\text{bool}. \ \exists \ \text{state} \in Q. \ (\text{getM state}) = ?M \land (\text{getConflictFlag state}) = b \} \)
from \( \text{state} \in Q \)
have \( \text{getConflictFlag state} \in ?Q1 \)
by auto
with \( \text{wfBoolLess} \)
obtain \( \text{bMin} \) where \( \text{bMin} \in ?Q1 \land \forall b'. \ (b', bMin) \in \text{boolLess} \rightarrow b' \notin ?Q1 \)
unfolding \( \text{wf-eq-minimal} \)
apply \( \text{erule-tac x=?Q1 in allE} \)
apply \( \text{erule-tac x=\text{getConflictFlag state} in allE} \)
by auto
from \( \text{bMin} \in ?Q1 \) obtain \( \text{stateMin} \)
where \( \text{stateMin} \in Q. \ (\text{getM stateMin}) = ?M \land \text{getConflictFlag stateMin} = \text{bMin} \)
by auto
have \( \forall \text{state}'. \ (\text{state}', \text{stateMin}) \in \text{boolLessState} \rightarrow \text{state'} \notin Q \)
proof
fix \( \text{state'} \)
show \( \text{state}', \text{stateMin}) \in \text{boolLessState} \rightarrow \text{state'} \notin Q \)
proof
assume \( \text{state}', \text{stateMin}) \in \text{boolLessState} \)
with ⟨getM stateMin = ?M⟩
have getM state' = getM stateMin (getConflictFlag state',
getConflictFlag stateMin) ∈ boolLess
  unfolding boolLessState-def
  by auto
from ∀ b'. (b', bMin) ∈ boolLess → b' ∉ ?Q1
⟨(getConflictFlag state', getConflictFlag stateMin) ∈ boolLess⟩
⟨getConflictFlag stateMin = bMin⟩
have getConflictFlag state' ∉ ?Q1
  by simp
with ⟨getM state' = getM stateMin ⟩ ⟨getM stateMin = ?M⟩
show state' ∉ Q
  by auto
qed
qed
with ⟨stateMin ∈ Q⟩
have ∃ stateMin ∈ Q. (∀ state'. (state', stateMin) ∈ boolLessState
  → state' ∉ Q)
  by auto
}
thus ?thesis
  by auto
qed
qed

lemma wfMultLessState:
  shows wf multLessState
  unfolding wf-eq-minimal
proof –
  show ∀ Q state. state ∈ Q → (∃ stateMin ∈ Q. ∀ state'. (state', stateMin) ∈ multLessState
                           → state' ∉ Q)
    by auto
  }
  thus ?thesis
    by auto
qed

wfMultLessState
where \(\text{stateMin} \in Q\) 
\( (\text{getM stateMin}) = ?M \) 
\( \text{getC stateMin} = \text{Cmin} \)

by auto

have \(\forall \text{state}'. \ (\text{state}', \text{stateMin}) \in \text{multLessState} \rightarrow \text{state}' \notin Q\)

proof

fix \(\text{state}'\)

show \((\text{state}', \text{stateMin}) \in \text{multLessState} \rightarrow \text{state}' \notin Q\)

proof

assume \((\text{state}', \text{stateMin}) \in \text{multLessState} \)

with \((\text{getM stateMin} = ?M)\)

have \((\text{getM state'} = \text{getM stateMin})\) 
\((\text{getC state'} = \text{getC stateMin})\)

\(\in \text{multLess } ?M\)

unfolding \(\text{multLessState-def}\)

by auto

from \(\forall C', (C', \text{Cmin}) \in \text{multLess } ?M \rightarrow C' \notin ?Q1\)

\((\text{getC state'}, \text{getC stateMin}) \in \text{multLess } ?M\) 
\((\text{getC stateMin})\) 

\(= \text{Cmin}\)

have \((\text{getC state'} \notin ?Q1)\)

by simp

with \((\text{getM state'} = \text{getM stateMin})\)

\((\text{getM stateMin} = ?M)\)

show \((\text{state'} \notin Q)\)

by auto

qed

qed

with \((\text{stateMin} \in Q)\)

have \(\exists \text{stateMin} \in Q. \ (\forall \text{state}'. \ (\text{state}', \text{stateMin}) \in \text{multLessState} \rightarrow \text{state}' \notin Q)\)

by auto

}\n
thus \(\text{?thesis}\)

by auto

qed

qed

lemma \(\text{wfLearnLessState}:\)

shows \(\text{wf learnLessState}\)

unfolding \(\text{wf-eq-minimal}\)

proof−

show \(\forall Q \text{ state}. \ \text{state} \in Q \rightarrow (\exists \text{stateMin} \in Q. \ (\forall \text{state}'. \ (\text{state}', \text{stateMin}) \in \text{learnLessState} \rightarrow \text{state}' \notin Q)\)

proof−

\{ 

fix \(Q :: \text{State set and state :: State}\)

assume \(\text{state} \in Q\)

let \(?M = (\text{getM state})\)

let \(?C = (\text{getC state})\)

let \(?\text{conflictFlag} = (\text{getConflictFlag state})\)

let \(?Q1 = (F::Formula. \exists \text{state}. \ \text{state} \in Q \land (\text{getM state}) = ?M \land (\text{getConflictFlag state}) = ?\text{conflictFlag})\)

\(\)
∧ (getC state) = ?C ∧ (getF state) = F \}
from \state ∈ Q
have getF state ∈ ?Q1
by auto
with wfLearnLess[of ?C]
obtain Fmin where Fmin ∈ ?Q1 ∀ F'. (F', Fmin) ∈ learnLess
?C → F' ∉ ?Q1
unfolding wf-eq-minimal
apply (erule-tac x = ?Q1 in allE)
apply (erule-tac x = getF state in allE)
by auto
from \Fmin ∈ ?Q1\ obtain stateMin
where stateMin ∈ Q (getM stateMin) = ?M (getC stateMin) = ?C (getConflictFlag stateMin) = ?conflictFlag getF stateMin = Fmin
by auto
have ∀ state'. (state', stateMin) ∈ learnLessState → state' ∉ Q
proof
fix state'
show (state', stateMin) ∈ learnLessState → state' ∉ Q
proof
assume (state', stateMin) ∈ learnLessState
with \getM stateMin = ?M \getC stateMin = ?C \getConflictFlag stateMin = ?conflictFlag\ have getM state' = getM stateMin getC state' = getC stateMin
getConflictFlag state' = getConflictFlag stateMin (getF state', getF stateMin) ∈ learnLess ?C
unfolding learnLessState-def
by auto
from \∀ F'. (F', Fmin) ∈ learnLess ?C → F' ∉ ?Q1\ \getF state' ∈ learnLess ?C \getF stateMin ∈ learnLess ?C\ have getF state' ∉ ?Q1
by simp
with \getM state' = getM stateMin \getC state' = getC stateMin \getConflictFlag state' = getConflictFlag stateMin\ have getM stateMin = ?M \getC stateMin = ?C \getConflictFlag stateMin = ?conflictFlag\ show state' ∉ Q
by auto
qed
qed
have ∃ stateMin ∈ Q
have ∀ stateMin ∈ Q. (∀ state'. (state', stateMin) ∈ learnLessState → state' ∉ Q)
by auto
\}
thus ?thesis
by auto
Now we can prove the following key lemma which shows that the termination ordering is well founded.

**lemma** \(\text{wfTerminationLess}:
\)

**fixes** \(\text{decisionVars}::\text{Variable set}\) and \(F0::\text{Formula}\)

**assumes** finite \(\text{decisionVars}\)

**shows** \(\text{wf} (\text{terminationLess } F0 \text{ decisionVars})\)

**unfolding** \(\text{wf-eq-minimal}\)

**proof**

\[
\forall \ Q \ state. \ state \in Q \implies (\exists \ stateMin \in Q. \forall \ state', (state', stateMin) \in \text{terminationLess } F0 \text{ decisionVars} \implies state' \not\in Q)
\]

**proof**

\[
\begin{align*}
&\{ \\
&\text{fix } Q::\text{State set} \\
&\text{fix } state::\text{State} \\
&\text{assume } state \in Q
\end{align*}
\]

\[
\begin{align*}
&\text{from } \langle \text{finite decisionVars} \rangle \\
&\text{have } \text{wf } (\text{lexLessState } F0 \text{ decisionVars}) \\
&\quad \text{using } \text{wfLexLessState}[\text{of decisionVars } F0] \\
&\quad \text{by simp}
\end{align*}
\]

**with** \((state \in Q)\) **obtain** state0

**where** state0 \(\in Q \forall state'. (state', state0) \in \text{lexLessState } F0 \text{ decisionVars} \implies state' \not\in Q\)

**unfolding** \(\text{wf-eq-minimal}\)

**by** auto

**let** \(?Q0 = \{state. state \in Q \land (getM state) = (getM state0)\}\) from \(\langle state0 \in Q \rangle\)

**have** state0 \(\in ?Q0\)

**by simp

**have** \(\text{wf boolLessState}\)

**using** \(\text{wfBoolLessState}\)

**with** \(\langle state0 \in Q \rangle\) **obtain** state1

**where** state1 \(\in ?Q0 \forall state'. (state', state1) \in \text{boolLessState} \implies state' \not\in ?Q0\)

**unfolding** \(\text{wf-eq-minimal}\)

**apply** (erule-tac \(x=?Q0 \text{ in allE}\))

**apply** (erule-tac \(x=state0 \text{ in allE}\))

**by auto

**let** \(?Q1 = \{state. state \in Q \land getM state = getM state0 \land getConflictFlag state = getConflictFlag state1\}\) from \(\langle state1 \in ?Q0 \rangle\)

**have** state1 \(\in ?Q1\)

**by simp

**have** \(\text{wf multLessState}\)
using wfMultLessState

with \( \text{state1} \in ?Q1 \) obtain \( \text{state2} \)

where \( \text{state2} \in ?Q1 \land \text{state}'. (\text{state}', \text{state2}) \in \text{multLessState} \)

\( \rightarrow \text{state}' \notin ?Q1 \)

unfolding wf-eq-minimal
apply (erule-tac \( x=\text{Q1} \) in allE)
apply (erule-tac \( x=\text{state1} \) in allE)
by auto

let \( ?Q2 = \{ \text{state}, \text{state} \in Q \land \text{getM state} = \text{getM state0} \land \text{getConflictFlag state} = \text{getConflictFlag state1} \land \text{getC state} = \text{getC state2} \} \)

from \( \text{state2} \in ?Q1 \)
have \( \text{state2} \in ?Q2 \)
by simp

have wf learnLessState
using wfLearnLessState

\( \rightarrow \text{state}' \notin ?Q2 \)

unfolding wf-eq-minimal
apply (erule-tac \( x=\text{Q2} \) in allE)
apply (erule-tac \( x=\text{state2} \) in allE)
by auto

from \( \text{state3} \in ?Q2 \)
have \( \text{state3} \in Q \)
by simp

from \( \text{state1} \in ?Q0 \)
have \( \text{getM state1} = \text{getM state0} \)
by simp

from \( \text{state2} \in ?Q1 \)
have \( \text{getM state2} = \text{getM state0} \land \text{getConflictFlag state2} = \text{getConflictFlag state1} \land \text{getC state2} = \text{getC state2} \)
by auto

let \( \text{stateMin} = \text{state3} \)

have \( \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{terminationLess F0 decision-Vars} \rightarrow \text{state}' \notin Q \)
proof
fix \text{state}'
show \((\text{state}', \text{stateMin}) \in \text{terminationLess F0 decision-Vars} \rightarrow \text{state}' \notin Q \)
proof
assume \((\text{state}', \text{stateMin}) \in \text{terminationLess F0 decision-Vars} \)
hence
(state', ?stateMin) ∈ lexLessState F0 decisionVars ∨
(state', ?stateMin) ∈ boolLessState ∨
(state', ?stateMin) ∈ multLessState ∨
(state', ?stateMin) ∈ learnLessState

unfolding terminationLess-def
by auto

moreover
{
  assume (state', ?stateMin) ∈ lexLessState F0 decisionVars
  with (getM state3 = getM state0)
  have (state', state0) ∈ lexLessState F0 decisionVars
  unfolding lexLessState-def
  by simp
  with (∀ state'. (state', state0) ∈ lexLessState F0 decisionVars
to state' ∈ Q)
  have state' ∉ Q
    by simp
}

moreover
{
  assume (state', ?stateMin) ∈ boolLessState
  from (?stateMin ∈ ?Q2)
  (getM state1 = getM state0)
  have getConflictFlag state3 = getConflictFlag state1 getM
  state3 = getM state1
    by auto
  with ((state', ?stateMin) ∈ boolLessState)
  have (state', state1) ∈ boolLessState
  unfolding boolLessState-def
  by simp
  with (∀ state'. (state', state1) ∈ boolLessState → state' ∉ ?Q0)
  have state' ∉ ?Q0
    by simp
  from ((state', state1) ∈ boolLessState) (getM state1 = getM
  state0)
  have getM state' = getM state0
  unfolding boolLessState-def
  by auto
  with (state' ∉ ?Q0)
  have state' ∉ Q
    by simp
}

moreover
{
  assume (state', ?stateMin) ∈ multLessState
  from (?stateMin ∈ ?Q2)
  (getM state1 = getM state0) (getM state2 = getM state0)
  (getConflictFlag state2 = getConflictFlag state1)

  state2 = getM state1
    by auto
  with ((state', ?stateMin) ∈ multLessState)
  have (state', state1) ∈ multLessState
  unfolding multLessState-def
  by simp
  with (∀ state'. (state', state1) ∈ multLessState → state' ∉ ?Q0)
  have state' ∉ ?Q0
    by simp
  from ((state', state1) ∈ multLessState) (getM state1 = getM
  state0)
  have getM state' = getM state0
  unfolding multLessState-def
  by auto
  with (state' ∉ ?Q0)
  have state' ∉ Q
    by simp


have getC state3 = getC state2
getConflictFlag state2 getM state3 = getM state2
by auto
with ⟨(state', ?stateMin) ∈ multLessState⟩
have (state', state2) ∈ multLessState
unfolding multLessState-def
by auto
with ∀state'. (state', state2) ∈ multLessState → state' /∈ ?Q1

have state' /∈ ?Q1
by simp
from ⟨(state', state2) ∈ multLessState⟩ ⟨getM state2 = getM state0⟩ ⟨getConflictFlag state2 = getConflictFlag state1⟩
have getM state' = getM state0 getConflictFlag state' =
getConflictFlag state1
unfolding multLessState-def
by auto
with ⟨state' /∈ ?Q1⟩
have state' /∈ Q
by simp
 }
moreover
{
assume (state', ?stateMin) ∈ learnLessState
with ∀state'. (state', ?stateMin) ∈ learnLessState → state'
/∈ ?Q2

have state' /∈ ?Q2
by simp
from ⟨(state', ?stateMin) ∈ learnLessState⟩
:getM state3 = getM state0⟩ ⟨getConflictFlag state3 = getConflictFlag state1⟩ ⟨getC state3 = getC state2⟩
have getM state' = getM state0 getConflictFlag state' =
getConflictFlag state1 getC state' = getC state2
unfolding learnLessState-def
by auto
with ⟨state' /∈ ?Q2⟩
have state' /∈ Q
by simp
 }
ultimately
show state' /∈ Q
by auto
qed

with ⟨?stateMin ∈ Q⟩ have (∃ stateMin ∈ Q. ∀ state'. (state', stateMin) ∈ terminationLess F0 decisionVars → state' /∈ Q)
by auto
}
thus ?thesis
Using the termination ordering we show that the transition relation is well founded on states reachable from initial state.

**Theorem ufTransitionRelation:**

**Fixes** decisionVars :: Variable set and F0 :: Formula

** Assumes** finite decisionVars and isInitialState state0 F0

**Shows** wf { (stateB, stateA).

(transition state0, stateA) ∈ transitionRelation F0 decisionVars ∧
(transition stateA stateB F0 decisionVars) }

**Proof:**

1. Let ?rel = { (stateB, stateA).

(transition state0, stateA) ∈ transitionRelation F0 decisionVars ∧
(transition stateA stateB F0 decisionVars) }

2. Let ?rel' = terminationLess F0 decisionVars

3. Have ∀ x y. (x, y) ∈ ?rel → (x, y) ∈ ?rel'

**Proof:**

1. Fix stateA::State and stateB::State

2. Assume (stateB, stateA) ∈ ?rel

3. Hence (stateB, stateA) ∈ ?rel'

4. Using isInitialState state0 F0

5. Using invariantsHoldInValidRunsFromInitialState[of state0 F0 stateA decisionVars]

6. Using stateIsDecreasedByValidTransitions[of stateA F0 decisionVars stateB]

7. By simp

8. Thus ?thesis

9. By simp

**Qed**

Moreover

1. Have wf ?rel'

2. Using (finite decisionVars)

3. By (rule wfTerminationLess)

**Ultimately**

1. Show ?thesis

2. Using wellFoundedEmbed[of ?rel ?rel']

3. By simp

**Qed**

We will now give two corollaries of the previous theorem. First is a weak termination result that shows that there is a terminating run from every initial state to the final one.

**Corollary**
fixes decisionVars :: Variable set and F0 :: Formula and state0 :: State
assumes finite decisionVars and isInitialState state0 F0
shows \( \exists \text{ state}. \ (\text{state0, state}) \in \text{transitionRelation} \ F0 \ \text{decisionVars} \wedge \text{isFinalState} \ \text{state} \ F0 \ \text{decisionVars} \)
proof–
\{
  assume \( \neg \ ?\text{thesis} \)
  let \(?Q = \{\text{state}. \ (\text{state0, state}) \in \text{transitionRelation} \ F0 \ \text{decisionVars}\}\)
  let \(?rel = \{\text{stateB, stateA}. \ (\text{state0, stateA}) \in \text{transitionRelation} \ F0 \ \text{decisionVars} \wedge \text{transition} \text{stateA} \text{stateB} \ F0 \ \text{decisionVars}\}\)
  have state0 \in ?Q
    unfolding transitionRelation-def
    by simp
    hence \( \exists \text{ state}. \ \text{state} \in ?Q \)
    by auto
from assms
have wf ?rel
  using wfTransitionRelation[of decisionVars state0 F0]
  by auto
hence \( \forall Q. (\exists x. \ x \in Q) \rightarrow (\exists \text{stateMin} \in Q. \forall \text{state}. \ (\text{state, stateMin}) \in ?rel \rightarrow \text{state} \notin Q) \)
  unfolding wf-eq-minimal
  by simp
hence \( (\exists x. \ x \in ?Q) \rightarrow (\exists \text{stateMin} \in ?Q. \forall \text{state}. \ (\text{state, stateMin}) \in ?rel \rightarrow \text{state} \notin ?Q) \)
  by rule
with \( (\exists \text{state}. \ \text{state} \in ?Q) \)
have \( \exists \text{stateMin} \in ?Q. \forall \text{state}. \ (\text{state, stateMin}) \in ?rel \rightarrow \text{state} \notin ?Q \)
  by simp
then obtain stateMin
where stateMin \in ?Q and \( \forall \text{state}. \ (\text{state, stateMin}) \in ?rel \rightarrow \text{state} \notin ?Q \)
  by auto
from (stateMin \in ?Q)
have (state0, stateMin) \in transitionRelation F0 decisionVars
  by simp
with (\( \neg ?\text{thesis} \))
have \( \neg \text{isFinalState} \text{stateMin} \ F0 \ \text{decisionVars} \)
  by simp
then obtain state'::State
where transition stateMin state' F0 decisionVars
unfolding isFinalState-def
by auto
have \((\text{state}', \text{stateMin}) \in \text{?rel}\)
using \((\text{state}0, \text{stateMin}) \in \text{transitionRelation } F0 \text{ decisionVars}\)
\(<\text{transition } \text{stateMin} \text{ state'} F0 \text{ decisionVars}>\)
by simp
with \(? \text{state. } (\text{state}, \text{stateMin}) \in \text{?rel } \rightarrow \text{state } \notin \text{?Q}\)
have \(\text{state}' \notin \text{?Q}\)
by force
moreover
from \((\text{state}0, \text{stateMin}) \in \text{transitionRelation } F0 \text{ decisionVars}\)
\(<\text{transition } \text{stateMin} \text{ state'} F0 \text{ decisionVars}>\)
have \(\text{state}' \in \text{?Q}\)
unfolding \text{transitionRelation-def}
using \text{rtrancl-into-rtrancl[of state0 stateMin } \{(\text{stateA}, \text{stateB})\}.\text{transition stateA stateB F0 decisionVars} \text{ state'}\]
by simp
ultimately
have \(\text{False}\)
by simp

\}
thus \(?\text{thesis}\)
by auto
qed

Now we prove the final strong termination result which states that there cannot be infinite chains of transitions. If there is an infinite transition chain that starts from an initial state, its elements would for a set that would contain initial state and for every element of that set there would be another element of that set that is directly reachable from it. We show that no such set exists.

\text{corollary} \ no\text{InfiniteTransitionChains:}
\text{fixes } F0::\text{Formula and decisionVars::Variable set}
\text{assumes finite decisionVars}
\text{shows } \neg (\exists \ Q::(\text{State set}). \exists \text{state0 } \in \ Q. \text{isInitialState state0 F0 } \land

\(\forall \text{ state } \in \ Q. (\exists \text{ state'} \in \ Q. \text{transition state state'} F0 \text{ decisionVars})\))

\text{proof—}
\{
assume \(\neg \text{?thesis}\)
then obtain \(Q::\text{State set and state0::State}\)
where \text{isInitialState state0 F0 state0 } \in \ Q
\(\forall \text{ state } \in \ Q. (\exists \text{ state'} \in \ Q. \text{transition state state'} F0 \text{ decisionVars})\)
by auto
let \(?\text{rel} = \{(\text{stateB}, \text{stateA}). (\text{state0}, \text{stateA}) \in \text{transitionRelation } F0 \text{ decisionVars} \land\)
transition stateA stateB F0 decisionVars

from ⟨finite decisionVars⟩ ⟨isInitialState state0 F0⟩
have wf ?rel
  using wfTransitionRelation
by simp

hence wfmin: ∀ Q. x ∈ Q →
  (∃ z ∈ Q. ∀ y. (y, z) ∈ ?rel → y /∈ Q)
unfolding wf-eq-minimal
by simp

let ?Q = { state ∈ Q. (state0, state) ∈ transitionRelation F0 decisionVars }

from ⟨state0 ∈ Q⟩
have state0 ∈ ?Q
unfolding transitionRelation-def
by simp
with wfmin
obtain stateMin::State
  where stateMin ∈ ?Q and ∀ y. (y, stateMin) ∈ ?rel → y /∈ ?Q
apply (erule-tac x=?Q in allE)
by auto

from ⟨stateMin ∈ ?Q⟩
have stateMin ∈ Q (state0, stateMin) ∈ transitionRelation F0 decisionVars
by auto
with (∀ state ∈ Q. (∃ state' ∈ Q. transition state state' F0 decisionVars))
obtain state'::State
  where state' ∈ Q transition stateMin state' F0 decisionVars
by auto

with ⟨(state0, stateMin) ∈ transitionRelation F0 decisionVars⟩
have ⟨state', stateMin⟩ ∈ ?rel
by simp
with (∀ y. (y, stateMin) ∈ ?rel → y /∈ ?Q)
have state' /∈ ?Q
by force

from ⟨state' ∈ ?Q⟩ ⟨(state0, stateMin) ∈ transitionRelation F0 decisionVars⟩
  ⟨transition stateMin state' F0 decisionVars⟩
have state' ∈ ?Q
unfolding transitionRelation-def
  using rtrancl-into-rtrancl[of state0 stateMin { (stateA, stateB). transition stateA stateB F0 decisionVars } state']
by simp
with ⟨state' /∈ ?Q⟩
have False
by simp

302
thus thesis
by force
qed

7.5 Completeness

In this section we will first show that each final state is either SAT or UNSAT state.

lemma finalNonConflictState:
  fixes state::State and FO :: Formula
  assumes
  getConflictFlag state = False and
  ¬ applicableDecide state decisionVars and
  ¬ applicableConflict state
  shows ¬ formulaFalse (getF state) (elements (getM state)) and
  vars (elements (getM state)) ⊇ decisionVars
proof−
  from (¬ applicableConflict state) (getConflictFlag state = False)
  show ¬ formulaFalse (getF state) (elements (getM state))
    unfolding applicableConflictCharacterization
    by (auto simp add: formulaFalseIfFalseClause formulaEntailsItsClauses)
  show vars (elements (getM state)) ⊇ decisionVars
proof
  fix x :: Variable
  let ?l = Pos x
  assume x ∈ decisionVars
  hence var ?l = x and var ?l ∈ decisionVars and var (opposite ?l) ∈ decisionVars
    by auto
  with (¬ applicableDecide state decisionVars)
  have literalTrue ?l (elements (getM state)) ∨ literalFalse ?l (elements (getM state))
    unfolding applicableDecideCharacterization
    by force
  with (var ?l = x)
  show x ∈ vars (elements (getM state))
    using valuationContainsItsLiteralsVariable[of ?l elements (getM state)]
    using valuationContainsItsLiteralsVariable[of opposite ?l elements (getM state)]
    by auto
qed

lemma finalConflictingState:
  fixes state :: State
  assumes
  InvariantUniq (getM state) and
InvariantReasonClauses (getF state) (getM state) and
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
and
¬ applicableExplain state and
¬ applicableBackjump state and
getConflictFlag state
shows
getC state = []
proof (cases ∀ l. l el getC state → opposite l el decisions (getM state))
case True
{
assume getC state ≠ []
let ℓ = getLastAssertedLiteral (oppositeLiteralList (getC state))
(elements (getM state))

from ⟨InvariantUniq (getM state)⟩
have uniq (elements (getM state))
  unfolding InvariantUniq-def
.

from ⟨getConflictFlag state⟩ ⟨InvariantCFalse (getConflictFlag state)
  (getM state) (getC state)⟩
  have clauseFalse (getC state) (elements (getM state))
  unfolding InvariantCFalse-def
  by simp

with ⟨getC state ≠ []⟩ ⟨InvariantUniq (getM state)⟩
  have isLastAssertedLiteral ℓ (oppositeLiteralList (getC state))
  (elements (getM state))
  unfolding InvariantUniq-def
  using getLastAssertedLiteralCharacterization
  by simp

with True ⟨uniq (elements (getM state))⟩
  have ∃ level. (isBackjumpLevel level (opposite ℓ) (getC state)
  (getM state))
  using allDecisionsThenExistsBackjumpLevel [of getM state getC state opposite ℓ]
  by simp
then
obtain level::nat where
  isBackjumpLevel level (opposite ℓ) (getC state) (getM state)
  by auto
with ⟨getConflictFlag state⟩
have applicableBackjump state
  unfolding applicableBackjumpCharacterization
  by auto
with (¬ applicableBackjump state)
  have False
  by simp
}
thus ?thesis
by auto

next
case False
  then obtain literal::Literal where literal el getC state ¬ opposite
  literal el decisions (getM state)
  by auto
  with (InvariantReasonClauses (getF state) (getM state)) (InvariantCFalse
  (getConflictFlag state) (getM state) (getC state)) (getConflictFlag state)
  have ∃ c. formulaEntailsClause (getF state) c ∧ isReason c (opposite
  literal) (elements (getM state))
    using explainApplicableToEachNonDecision[of getF state getM
  state getConflictFlag state getC state opposite literal]
  by auto
  then obtain c::Clause
    where formulaEntailsClause (getF state) c isReason c (opposite
    literal) (elements (getM state))
    by auto
    with (¬ applicableExplain state) (getConflictFlag state) (literal el
    (getC state))
    have False
      unfolding applicableExplainCharacterization
    by auto
    thus ?thesis
    by simp
qed

lemma finalStateCharacterizationLemma:
  fixes state :: State
  assumes
    InvariantUniq (getM state) and
    InvariantReasonClauses (getF state) (getM state) and
    InvariantCFalse (getConflictFlag state) (getM state) (getC state)
  and
    ¬ applicableDecide state decisionVars and
    ¬ applicableConflict state
    ¬ applicableExplain state and
    ¬ applicableBackjump state
  shows
    (getConflictFlag state = False ∧
    ¬ formulaFalse (getF state) (elements (getM state)) ∧
    vars (elements (getM state)) ⊇ decisionVars) ∨
    (getConflictFlag state = True ∧
    getC state = [])
proof (cases getConflictFlag state)
case True
  hence getC state = []
    using assms
    using finalConflictingState
    by auto
  with True
  show ?thesis
    by simp
next
  case False
  hence ¬ formulaFalse (getF state) (elements (getM state)) and vars
    (elements (getM state)) ⊇ decisionVars
    using assms
    using finalNonConflictState
    by auto
  with False
  show ?thesis
    by simp
qed

theorem finalStateCharacterization:
  fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State
  assumes
    isInitialState state0 F0 and
    (state0, state) ∈ transitionRelation F0 decisionVars and
    isFinalState state F0 decisionVars
  shows
    (getConflictFlag state = False ∧
     ¬ formulaFalse (getF state) (elements (getM state)) ∧
     vars (elements (getM state)) ⊇ decisionVars) ∨
    (getConflictFlag state = True ∧
     getC state = [])

proof
  from ⟨isInitialState state0 F0; (state0, state) ∈ transitionRelation F0 decisionVars⟩
  have invariantsHoldInState state F0 decisionVars
    using invariantsHoldInValidRunsFromInitialState
    by simp
  hence
    ⋆: InvariantUniq (getM state)
    InvariantReasonClauses (getF state) (getM state)
    InvariantCFalse (getConflictFlag state) (getM state) (getC state)
    unfolding invariantsHoldInState-def
    by auto
  from ⟨isFinalState state F0 decisionVars⟩
have **:
¬ applicableDecide state decisionVars
¬ applicableConflict state
¬ applicableExplain state
¬ applicableLearn state
¬ applicableBackjump state

unfolding finalStateNonApplicable
by auto

from **
show ?thesis
using finalStateCharacterizationLemma[of state decisionVars]
by simp

qed

Completeness theorems are easy consequences of this characterization and soundness.

theorem completenessForSAT:
fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State
assumes
isInitialState state0 F0 and
(is state0, state) ∈ transitionRelation F0 decisionVars and
isFinalState state F0 decisionVars

shows getConflictFlag state = False ∧ ¬formulaFalse (getF state)
(elements (getM state)) ∧
vars (elements (getM state)) ⊇ decisionVars

proof –
from assms
have *: (getConflictFlag state = False ∧
¬formulaFalse (getF state) (elements (getM state)) ∧
vars (elements (getM state)) ⊇ decisionVars) ∨
(getConflictFlag state = True ∧
getC state = [])
using finalStateCharacterization[of state0 F0 state decisionVars]
by auto
{
assume ¬ (getConflictFlag state = False)
with *
have getConflictFlag state = True getC state = []
by auto
with assms
have ¬ satisfiable F0
using soundnessForUNSAT
by simp
with  \langle \text{satisfiable } F_0 \rangle
have \ False
  by simp
\}
with * show \ ?thesis
  by auto
qed

\textbf{theorem} completenessForUNSAT:
\textbf{fixes} F_0 :: \text{Formula} \textbf{and} decisionVars :: \text{Variable set} \textbf{and} state0 :: \text{State}
\textbf{and} state :: \text{State}
\textbf{assumes}
vars F_0 \subseteq decisionVars \textbf{and}
\neg \text{satisfiable } F_0 \textbf{and}
isInitialState state0 F_0 \textbf{and}
(state0, state) \in \text{transitionRelation } F_0 \text{ decisionVars} \textbf{and}
isFinalState state F_0 decisionVars
\textbf{shows}
\text{getConflictFlag} state = \text{True} \land \text{getC state} = []
\textbf{proof}–
\textbf{from} \text{assms}
\textbf{have} *: (\text{getConflictFlag} state = False \land
  \neg \text{formulaFalse} (getF state) (\text{elements} (getM state)) \land
  \text{vars} (\text{elements} (getM state)) \supseteq \text{decisionVars} \lor
  (\text{getConflictFlag} state = \text{True} \land
  \text{getC state} = []))
  \textbf{using} finalStateCharacterization[of state0 F0 state decisionVars]
  \textbf{by} auto
\{
\textbf{assume} \neg \text{getConflictFlag} state = \text{True}
\textbf{with} *
\textbf{have} \text{getConflictFlag} state = False \land \neg \text{formulaFalse} (getF state)
  (\text{elements} (getM state)) \land \text{vars} (\text{elements} (getM state)) \supseteq \text{decisionVars}
  \textbf{by} simp
\textbf{with} \text{assms}
\textbf{have} \text{satisfiable} F_0
  \textbf{using} soundnessForSAT[of F0 decisionVars state0 state]
  \textbf{unfolding} \text{satisfiable-def}
  \textbf{by} auto
\textbf{with} \langle \neg \text{satisfiable } F_0 \rangle
\textbf{have} False
  \textbf{by} simp
\}
\textbf{with} * show \ ?thesis
by auto

qed

theorem partialCorrectness:
  fixes F0 :: Formula and decisionVars :: Variable set and state0 :: State and state :: State
  assumes vars F0 ⊆ decisionVars and
  isInitialState state0 F0 and
  (state0, state) ∈ transitionRelation F0 decisionVars and
  isFinalState state F0 decisionVars
  shows satisfiable F0 = (¬ getConflictFlag state)

using assms
using completenessForUNSAT[of F0 decisionVars state0 state]
using completenessForSAT[of F0 state0 state decisionVars]
by auto

end

8 Functional implementation of a SAT solver with Two Watch literal propagation.

theory SatSolverCode
imports SatSolverVerification ~/src/HOL/Library/Code-Target-Numeral
begin

8.1 Specification

lemma [code-unfold]:
  fixes literal :: Literal and clause :: Clause
  shows literal el clause = List.member clause literal
  by (auto simp add: member-def)

datatype ExtendedBool = TRUE | FALSE | UNDEF

record State =
  — Satisfiability flag: UNDEF, TRUE or FALSE
getSATFlag :: ExtendedBool
  — Formula
getF :: Formula
  — Assertion Trail
getM :: LiteralTrail
  — Conflict flag
getConflictFlag :: bool — raised iff M falsifies F
— Conflict clause index
getConflictClause :: nat — corresponding clause from F is false in M
— Unit propagation graph
getQ :: Literal list — Unit propagation queue
— Two-watch literal scheme
— clause indices instead of clauses are used
getReason :: Literal ⇒ nat option — index of a clause that is a reason for propagation of a literal
getWatch1 :: nat ⇒ Literal option — First watch of a clause
getWatch2 :: nat ⇒ Literal option — Second watch of a clause
getWatchList :: Literal ⇒ nat list — Watch list of a given literal
— Conflict analysis data structures
getC :: Clause — Conflict analysis clause - always false in M
getCl :: Literal — Last asserted literal in (opposite getC)
getCll :: Literal — Second last asserted literal in (opposite getC)
getCn :: nat — Number of literals of (opposite getC) on the (currentLevel M)

definition
setWatch1 :: nat ⇒ Literal ⇒ State ⇒ State
where
setWatch1 clause literal state =
  state{ getWatch1 := (getWatch state)(clause := Some literal),
            getWatchList := (getWatchList state)(literal := clause ≠ (getWatchList state literal))
  }
define setWatch1-def[code-unfold]
definition
setWatch2 :: nat ⇒ Literal ⇒ State ⇒ State
where
setWatch2 clause literal state =
  state{ getWatch2 := (getWatch state)(clause := Some literal),
            getWatchList := (getWatchList state)(literal := clause ≠ (getWatchList state literal))
  }
define setWatch2-def[code-unfold]
definition
swapWatches :: nat ⇒ State ⇒ State
where

swapWatches clause state ==
  state(\ getWatch1 := (getWatch1 state)(clause := (getWatch2 state clause)),
  getWatch2 := (getWatch2 state)(clause := (getWatch1 state clause))
)

declare swapWatches-def[code-unfold]

primrec getNonWatchedUnfalsifiedLiteral :: Clause ⇒ Literal ⇒ Literal ⇒ LiteralTrail ⇒ Literal option
where
  getNonWatchedUnfalsifiedLiteral [] w1 w2 M = None |
  getNonWatchedUnfalsifiedLiteral (literal # clause) w1 w2 M =
    (if literal ≠ w1 ∧
      literal ≠ w2 ∧
      ¬ (literalFalse literal (elements M)) then
        Some literal
      else
        getNonWatchedUnfalsifiedLiteral clause w1 w2 M
    )

definition
  setReason :: Literal ⇒ nat ⇒ State ⇒ State
where
  setReason literal clause state =
    state(\ getReason := (getReason state)(literal := Some clause) )

declare setReason-def[code-unfold]

primrec notifyWatches-loop::Literal ⇒ nat list ⇒ nat list ⇒ State ⇒ State
where
  notifyWatches-loop literal [] newWl state = state(\ getWatchList :=
    (getWatchList state)(literal := newWl) ) |
  notifyWatches-loop literal (clause ≠ list') newWl state =
    (let state' = (if Some literal = (getWatch1 state clause) then
      (swapWatches clause state)
    else
      state) in
      case (getWatch1 state' clause) of
        None ⇒ state
      | Some w1 ⇒ (case (getWatch2 state' clause) of
        None ⇒ state
      | Some w2 ⇒
        (if (literalTrue w1 (elements (getM state')))) then
          notifyWatches-loop literal list' (clause ≠ newWl) state'
      ))
else
  (case (getNonWatchedUnfalsifiedLiteral (nth (getF state') clause) w1 w2 (getM state')) of
    Some l' ⇒
      notifyWatches-loop literal list' newWl (setWatch2 clause l' state')
    | None ⇒
      (if (literalFalse w1 (elements (getM state'))) then
        let state'' = (state'' getConflictFlag := True, getConflictClause := clause []) in
        notifyWatches-loop literal list' (clause # newWl) state''
      else
        let state'' = state'' getQ := (if w1 el (getQ state')
          then (getQ state')
          else (getQ state') @ [w1]
        ) in
        let state''' = (setReason w1 clause state'') in
        notifyWatches-loop literal list' (clause # newWl) state'''
      )
  )
)
)
)


definition

notifyWatches :: Literal ⇒ State ⇒ State
where

notifyWatches literal state ==
    notifyWatches-loop literal (getWatchList state literal) [] state

declare notifyWatches-def [code-unfold]

definition

assertLiteral :: Literal ⇒ bool ⇒ State ⇒ State
where

assertLiteral literal decision state ==
    let state' = (state(state getM := (getM state) @ [(literal, decision)]) [])
    in
    notifyWatches (opposite literal) state'

definition

applyUnitPropagate :: State ⇒ State

312
where
applyUnitPropagate state =
  (let state' = (assertLiteral (hd (getQ state)) False state) in
   state'(| getQ := tl (getQ state')))

partial-function (tailrec)
exhaustiveUnitPropagate :: State ⇒ State
where
exhaustiveUnitPropagate-unfold[code]:
exhaustiveUnitPropagate state =
  (if (getConflictFlag state) ∨ (getQ state) = [] then
    state
  else
    exhaustiveUnitPropagate (applyUnitPropagate state)
)

inductive
exhaustiveUnitPropagate-dom :: State ⇒ bool
where
step: (¬ getConflictFlag state ⇒ getQ state ≠ []
  ⇒ exhaustiveUnitPropagate-dom (applyUnitPropagate state))
  ⇒ exhaustiveUnitPropagate-dom state

definition
addClause :: Clause ⇒ State ⇒ State
where
addClause clause state =
  (let clause' = (remdups (removeFalseLiterals clause (elements (getM state)))) in
   (if (clauseTrue clause' (elements (getM state))) then
    state
  else (if clause'=[] then
    state(| getSATFlag := FALSE )
  else (if (length clause' = 1) then
    let state' = (assertLiteral (hd clause') False state) in
    exhaustiveUnitPropagate state'
  else (if (clauseTautology clause') then
    state
  else
    let clauseIndex = length (getF state) in
    let state' = state(| getF := (getF state) @ [clause'] ) in
    let state'' = setWatch1 clauseIndex (nth clause' 0) state' in
    let state''' = setWatch2 clauseIndex (nth clause' 1) state'' in
    state'''))
  )))
)
definition
initialState :: State
where
initialState =
  ({ getSATFlag = UNDEF,
      getF = [],
      getM = [],
      getConflictFlag = False,
      getConflictClause = 0,
      getQ = [],
      getReason = λ l. None,
      getWatch1 = λ c. None,
      getWatch2 = λ c. None,
      getWatchList = λ l. [],
      getC = [],
      getCl = (Pos 0),
      getC1 = (Pos 0),
      getCn = 0
  } )

primrec initialize :: Formula ⇒ State ⇒ State
where
initialize [] state = state |
initialize (clause # formula) state = initialize formula (addClause clause state)

definition
findFirstAssertedLiteral :: State ⇒ State
where
findFirstAssertedLiteral state =
  state ( getCl := getLastAssertedLiteral (oppositeLiteralList (getC state)) (elements (getM state)) )

definition
countCurrentLevelLiterals :: State ⇒ State
where
countCurrentLevelLiterals state =
  (let cl = currentLevel (getM state) in
   state ( getCn := length (filter (λ l. elementLevel (opposite l) (getM state)) = cl) (getC state)) )

definition
setConflictAnalysisClause :: Clause ⇒ State ⇒ State
where
setConflictAnalysisClause clause state =
  (let oppM0 = oppositeLiteralList (elements (prefixToLevel 0 (getM state))) in
   state ( setC = oppM0, setCn = length (filter (λ l. elementLevel (opposite l) (getM state))) ) )

314
let state′ = state (let C := remdups (list-diff clause oppM0) in countCurrentLevelLiterals (findLastAssertedLiteral state))

definition applyConflict :: State ⇒ State
where
applyConflict state =
(let conflictClause = (nth (getF state) (getConflictClause state)) in setConflictAnalysisClause conflictClause state)

definition applyExplain :: Literal ⇒ State ⇒ State
where
applyExplain literal state =
(case (getReason state literal) of
None ⇒ state
| Some reason ⇒
let res = resolve (getC state) (nth (getF state) reason)
(opposite literal) in
setConflictAnalysisClause res state)

partial-function (tailrec)
applyExplainUIP :: State ⇒ State
where
applyExplainUIP-unfold:
applyExplainUIP state =
(if (getCn state = 1) then state
else
applyExplainUIP (applyExplain (getCl state) state)
)

inductive applyExplainUIP-dom :: State ⇒ bool
where
step:
(getCn state ≠ 1
⇒ applyExplainUIP-dom (applyExplain (getCl state) state))
⇒⇒ applyExplainUIP-dom state

definition applyLearn :: State ⇒ State
where
applyLearn state =
  (if getC state = [opposite (getCl state)] then
   state
   else
   let state' = state[] getF := (getF state) @ [getC state] [] in
   let l = (getCl state) in
   let ll = (getLastAssertedLiteral (removeAll l (oppositeLiteralList
   (getC state))) (elements (getM state))) in
   let clauseIndex = length (getF state) in
   let state'' = setWatch1 clauseIndex (opposite l) state' in
   let state''' = setWatch2 clauseIndex (opposite ll) state'' in
   state'''[ getCll := ll ]
)

definition
getc BACKjumpLevel :: State ⇒ nat
where
getc BACKjumpLevel state ==
  (if getC state = [opposite (getCl state)] then
   0
   else
   elementLevel (getCll state) (getM state)
  )

definition
applyBackjump :: State ⇒ State
where
applyBackjump state =
  (let l = (getCl state) in
   let level = getBackjumpLevel state in
   let state' = state[ getConflictFlag := False, getQ := [], getM :=
   (prefixToLevel level (getM state))[] in
   let state'' = (if level > 0 then setReason (opposite l) (length (getF
   state) - 1) state' else state') in
   assertLiteral (opposite l) False state''
  )

axiomatization selectLiteral :: State ⇒ Variable set ⇒ Literal
where
selectLiteral-def:
Vbl − vars (elements (getM state)) ≠ {} —→
  var (selectLiteral state Vbl) ∈ (Vbl − vars (elements (getM state)))

definition
applyDecide :: State ⇒ Variable set ⇒ State
where

applyDecide state Vbl =
  assertLiteral (selectLiteral state Vbl) True state

definition

solve-loop-body :: State ⇒ Variable set ⇒ State

where

solve-loop-body state Vbl =
  (let state′ = exhaustiveUnitPropagate state in
   (if (getConflictFlag state′) then
    (if (currentLevel (getM state′)) = 0 then
     state′[\ getSATFlag := FALSE ]
    else
     (applyBackjump
      (applyLearn
       (applyExplainUIP
        (applyConflict
         state′
        )
      )))
    else
    (if (vars (elements (getM state′)) ⊇ Vbl) then
     state′[\ getSATFlag := TRUE ]
    else
     applyDecide state′ Vbl
    ))
  )
)

partial-function (tailrec)

solve-loop :: State ⇒ Variable set ⇒ State

where

solve-loop-unfold:

solve-loop state Vbl =
  (if (getSATFlag state) ≠ UNDEF then
   state
  else
   let state′ = solve-loop-body state Vbl in
   solve-loop state′ Vbl
  )

inductive
solve-loop-dom :: State ⇒ Variable set ⇒ bool
where
step:
(getSATFlag state = UNDEF
  ⇒ solve-loop-dom (solve-loop-body state Vbl) Vbl)
  ⇒ solve-loop-dom state Vbl

definition solve::Formula ⇒ ExtendedBool
where
solve F0 =
  (getSATFlag
    (solve-loop
      (initialize F0 initialState) (vars F0)
    )
  )


definition
InvariantWatchListsContainOnlyClausesFromF :: (Literal ⇒ nat list) ⇒ Formula ⇒ bool
where
InvariantWatchListsContainOnlyClausesFromF WI F =
  (∀ (l::Literal) (c::nat). c ∈ set (WI l) → 0 ≤ c ∧ c < length F)


definition
InvariantWatchListsUniq :: (Literal ⇒ nat list) ⇒ bool
where
InvariantWatchListsUniq WI =
  (∀ l. uniq (WI l))


definition
InvariantWatchListsCharacterization :: (Literal ⇒ nat list) ⇒ (nat ⇒ Literal option) ⇒ (nat ⇒ Literal option) ⇒ bool
where
InvariantWatchListsCharacterization WI w1 w2 =
  (∀ (c::nat) (l::Literal). c ∈ set (WI l) = (Some l = (w1 c) ∨ Some l = (w2 c)))


definition
InvariantWatchesEl :: Formula ⇒ (nat ⇒ Literal option) ⇒ (nat ⇒
\[\text{Literal option} \Rightarrow \text{bool}\]

**where**

\[\text{InvariantWatchesEl formula watch1 watch2} =\]
\[
\forall (\text{clause}::\text{nat}). \ 0 \leq \text{clause} \land \text{clause} < \text{length formula} \rightarrow \\
(\exists (w1::\text{Literal}) (w2::\text{Literal}). \ \text{watch1 clause} = \text{Some w1} \land \\
\text{watch2 clause} = \text{Some w2} \land \\
w1 \ \text{el (nth formula clause)} \land w2 \ \text{el (nth formula clause)})\]

**definition**

\[\text{InvariantWatchesDiffer :: Formula} \Rightarrow (\text{nat} \Rightarrow \text{Literal option}) \Rightarrow (\text{nat} \Rightarrow \text{Literal option}) \Rightarrow \text{bool}\]

**where**

\[\text{InvariantWatchesDiffer formula watch1 watch2} =\]
\[
\forall (\text{clause}::\text{nat}). \ 0 \leq \text{clause} \land \text{clause} < \text{length formula} \rightarrow \text{watch1 clause} \neq \text{watch2 clause}\]

**definition**

\[\text{watchCharacterizationCondition::Literal} \Rightarrow \text{Literal} \Rightarrow \text{LiteralTrail} \Rightarrow \text{Clause} \Rightarrow \text{bool}\]

**where**

\[\text{watchCharacterizationCondition w1 w2 M clause} =\]
\[
(\text{literalFalse w1 (elements M) } \rightarrow \\
(\exists l. l \ \text{el clause} \land \text{literalTrue l (elements M) } \land \text{elementLevel l M} \leq \text{elementLevel (opposite w1) M}) \lor \\
(\forall l. l \ \text{el clause} \land l \neq w1 \land l \neq w2 \rightarrow \\
\text{literalFalse l (elements M) } \land \text{elementLevel (opposite l) M} \leq \text{elementLevel (opposite w1) M}) \]
\[
)\]

**definition**

\[\text{InvariantWatchCharacterization::Formula} \Rightarrow (\text{nat} \Rightarrow \text{Literal option}) \Rightarrow (\text{nat} \Rightarrow \text{Literal option}) \Rightarrow \text{LiteralTrail} \Rightarrow \text{bool}\]

**where**

\[\text{InvariantWatchCharacterization F watch1 watch2 M} =\]
\[
(\forall c w1 w2. (0 \leq c \land c < \text{length F} \land \text{Some w1} = \text{watch1 c} \land \\
\text{Some w2} = \text{watch2 c} ) \rightarrow \\
\text{watchCharacterizationCondition w1 w2 M (nth F c)} \land \\
\text{watchCharacterizationCondition w2 w1 M (nth F c)}\]
\[
)\]

**definition**

\[\text{InvariantQCharacterization :: bool} \Rightarrow \text{Literal list} \Rightarrow \text{Formula} \Rightarrow \text{LiteralTrail} \Rightarrow \text{bool}\]

**where**

319
\[
\text{InvariantQCharacterization } \text{conflictFlag } Q F M \equiv \\
\neg \text{conflictFlag} \rightarrow (\forall (l::\text{Literal}). \ l \in Q = (\exists (c::\text{Clause}). \ c \in F \land \\
isUnitClause c \ l (\text{elements } M)))
\]

\textbf{definition}

\texttt{InvariantUniqQ :: Literal list } \Rightarrow \texttt{ bool}

\texttt{where}

\texttt{InvariantUniqQ } Q =

\texttt{uniq } Q

\textbf{definition}

\texttt{InvariantConflictFlagCharacterization :: bool } \Rightarrow \texttt{ Formula } \Rightarrow \texttt{ Literal-Trail } \Rightarrow \texttt{ bool}

\texttt{where}

\texttt{InvariantConflictFlagCharacterization } \text{conflictFlag } F M \equiv \\
\text{conflictFlag} = \text{formulaFalse } F (\text{elements } M)

\textbf{definition}

\texttt{InvariantNoDecisionsWhenConflict :: Formula } \Rightarrow \texttt{ LiteralTrail } \Rightarrow \texttt{ nat } \Rightarrow \\
\texttt{ bool}

\texttt{where}

\texttt{InvariantNoDecisionsWhenConflict } F M \text{ level } =

(\forall \text{ level’}. \text{ level’} < \text{ level } \rightarrow \\
\neg \text{formulaFalse } F (\text{elements } (\text{prefixToLevel } \text{ level’ } M)))

\textbf{definition}

\texttt{InvariantNoDecisionsWhenUnit :: Formula } \Rightarrow \texttt{ LiteralTrail } \Rightarrow \texttt{ nat } \Rightarrow \\
\texttt{ bool}

\texttt{where}

\texttt{InvariantNoDecisionsWhenUnit } F M \text{ level } =

(\forall \text{ level’}. \text{ level’} < \text{ level } \rightarrow \\
\neg (\exists \text{ clause literal } . \text{ clause } \in F \land \\
isUnitClause \text{ clause literal } (\text{elements } (\text{prefixToLevel } \text{ level’ } M)))

\textbf{definition} \texttt{InvariantEquivalentZL :: Formula } \Rightarrow \texttt{ LiteralTrail } \Rightarrow \texttt{ Formula } \Rightarrow \\
\texttt{ bool}

\texttt{where}

\texttt{InvariantEquivalentZL } F M \text{ F0 } =

\texttt{equivalentFormulae } (F \otimes \text{val2form } (\text{elements } (\text{prefixToLevel } 0 M)))

\texttt{F0
definition
InvariantGetReasonIsReason :: Literal → nat option → Formula → LiteralTrail → Literal set ⇒ bool
where
InvariantGetReasonIsReason GetReason F M Q ==
  ∀ literal. (literal el (elements M) ∧ ¬ literal el (decisions M) ∧
elementLevel literal M > 0 ➝
    (∃ (reason::nat). (GetReason literal) = Some reason ∧
     0 ≤ reason ∧ reason < length F ∧
     isReason (nth F reason) literal (elements M))
  ) ∧
    (currentLevel M > 0 ∧ literal ∈ Q ➝
     (∃ (reason::nat). (GetReason literal) = Some reason ∧
     0 ≤ reason ∧ reason < length F ∧
     isUnitClause (nth F reason) literal (elements M)
  ∨ clauseFalse (nth F reason) (elements M))

definition
InvariantConflictClauseCharacterization :: bool ⇒ nat ⇒ Formula ⇒ LiteralTrail ⇒ bool
where
InvariantConflictClauseCharacterization conflictFlag conflictClause F M ==
  conflictFlag ➝ (conflictClause < length F ∧
  clauseFalse (nth F conflictClause) (elements M))

definition
InvariantClCharacterization :: Literal ⇒ Clause ⇒ LiteralTrail ⇒ bool
where
InvariantClCharacterization Cl C M ==
  isLastAssertedLiteral Cl (oppositeLiteralList C) (elements M)

definition
InvariantClCharacterization :: Literal ⇒ Literal ⇒ Clause ⇒ LiteralTrail ⇒ bool
where
InvariantClCharacterization Cl Cl̸ C M ==
  set C ≠ {opposite Cl} ➝
    isLastAssertedLiteral Cl̸ (removeAll Cl (oppositeLiteralList C))
  (elements M)

definition
InvariantClCurrentLevel :: Literal ⇒ LiteralTrail ⇒ bool
where
InvariantClCurrentLevel \( Cl \ M \)
\[ \equiv \]
\[ \text{elementLevel} \ Cl \ M \ = \ \text{currentLevel} \ M \]

definition
InvariantCnCharacterization :: \( \text{nat} \ \Rightarrow \ \text{Clause} \ \Rightarrow \ \text{LiteralTrail} \ \Rightarrow \ \text{bool} \)
\where
InvariantCnCharacterization \( Cn \ C \ M \)
\[ \equiv \]
\[ Cn = \text{length} \ (\text{filter} \ (\lambda \ l. \ \text{elementLevel} \ (\text{opposite} \ l) \ M \ = \ \text{currentLevel} \ M) \ (\text{remdup} \ C)) \]

definition
InvariantUniqC :: \( \text{Clause} \ \Rightarrow \ \text{bool} \)
\where
InvariantUniqC \( \text{clause} \)
\[ = \]
\[ \text{uniq} \ \text{clause} \]

definition
InvariantVarsQ :: \( \text{Literal list} \ \Rightarrow \ \text{Formula} \ \Rightarrow \ \text{Variable set} \ \Rightarrow \ \text{bool} \)
\where
InvariantVarsQ \( Q \ F0 \ Vbl \)
\[ = \]
\[ \text{vars} \ Q \ \subseteq \ \text{vars} \ F0 \ \cup \ Vbl \]

end

theory AssertLiteral
imports SatSolverCode
begin

lemma getNonWatchedUnfalsifiedLiteralSomeCharacterization:
\textbf{fixes} \( \text{clause} :: \ \text{Clause} \ \text{and} \ w1 :: \ \text{Literal} \ \text{and} \ w2 :: \ \text{Literal} \ \text{and} \ M :: \ \text{LiteralTrail} \ \text{and} \ l :: \ \text{Literal} \)
\textbf{assumes}
\( \text{getNonWatchedUnfalsifiedLiteral} \ \text{clause} \ w1 \ w2 \ M = \ \text{Some} \ l \)
\textbf{shows}
\( l \ \text{el} \ \text{clause} \ l \neq w1 \ l \neq w2 \ \neg \ \text{literalFalse} \ l \ (\text{elements} \ M) \)
\textbf{using} \ \text{assms}
\textbf{by} \ (\text{induct \ clause}) \ (\text{auto: \ split-if-asm})

lemma getNonWatchedUnfalsifiedLiteralNoneCharacterization:
\textbf{fixes} \( \text{clause} :: \ \text{Clause} \ \text{and} \ w1 :: \ \text{Literal} \ \text{and} \ w2 :: \ \text{Literal} \ \text{and} \ M :: \ \text{LiteralTrail} \)

end
assumes
getNonWatchedUnfalsifiedLiteral clause w1 w2 M = None
shows
∀ l. l ∈ clause ∧ l ≠ w1 ∧ l ≠ w2 −→ literalFalse l (elements M)
using assms
by (induct clause) (auto split: split-if-asm)

lemma swapWatchesEffect:
fixes clause::nat and state::State and clause′::nat
shows
getWatch1 (swapWatches clause state) clause′ = (if clause = clause′
then getWatch2 state clause′ else getWatch1 state clause′) and
getWatch2 (swapWatches clause state) clause′ = (if clause = clause′
then getWatch1 state clause′ else getWatch2 state clause′)
unfolding swapWatches-def
by auto

lemma notifyWatchesLoopPreservedVariables:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and
state :: State
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
∀ (c::nat). c ∈ set Wl −→ 0 ≤ c ∧ c < length (getF state)
shows
let state′ = (notifyWatches-loop literal Wl newWl state) in
(getM state′) = (getM state) ∧
(getF state′) = (getF state) ∧
(getSATFlag state′) = (getSATFlag state) ∧
isPrefix (getQ state) (getQ state′)
unfolding isPrefix-def
by simp

proof (induct Wl arbitrary: newWl state)
case Nil
thus ?case
by simp
next
case (Cons clause Wl)
from ∀ (c::nat). c ∈ set (clause # Wl) −→ 0 ≤ c ∧ c < length
(getF state)
have $0 \leq \text{clause} \land \text{clause} < \text{length} \left( \text{getF state} \right)$
by auto
then obtain \text{wa}::\text{Literal} and \text{wb}::\text{Literal}
where \text{getWatch1 state clause} = \text{Some wa} and \text{getWatch2 state clause} = \text{Some wb}
using Cons
unfolding InvariantWatchesEl-def
by auto
show \text{?case}
proof \(\text{(cases Some literal} = \text{getWatch1 state clause})\)
\hspace{1em} case True
let \text{?state'} = \text{swapWatches clause state}
let \text{?w1} = \text{wb}
have \text{getWatch1 ?state' clause} = \text{Some ?w1}
using \text{getWatch2 state clause} = \text{Some wb}
unfolding \text{swapWatches-def}
by auto
let \text{?w2} = \text{wa}
have \text{getWatch2 ?state' clause} = \text{Some ?w2}
using \text{getWatch1 state clause} = \text{Some wa}
unfolding \text{swapWatches-def}
by auto
show \text{?thesis}
proof \(\text{(cases literalTrue \?w1 (elements (getM ?state'))})\)
\hspace{1em} case True

from Cons(2)
\hspace{1em} have InvariantWatchesEl \(\text{getF ?state'} \) \text{(getWatch1 ?state')}
\hspace{1em} (getWatch2 ?state')
unfolding InvariantWatchesEl-def
unfolding \text{swapWatches-def}
by auto
moreover
have \text{getM ?state'} = \text{getM state} \land
\text{getF ?state'} = \text{getF state} \land
\text{getSATFlag ?state'} = \text{getSATFlag state} \land
\text{getQ ?state'} = \text{getQ state}
unfolding \text{swapWatches-def}
by simp
ultimately
show \text{?thesis}
using Cons(1)[of \text{?state' clause} \# newWl]
using Cons(3)
using \text{getWatch1 ?state' clause} = \text{Some ?w1};
using \text{getWatch2 ?state' clause} = \text{Some w2};
using \text{literalTrue \?w1 (elements (getM ?state'))};
by (simp add: Let-def)

324
next
  case False
  show ?thesis
  proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
    case (Some l')
    hence l' el (nth (getF ?state') clause)
    using getNonWatchedUnfalsifiedLiteralSomeCharacterization
    by simp

    let ?state'' = setWatch2 clause l' ?state'

    from Cons(2)
    have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
    (getWatch2 ?state'')
    using l' el (nth (getF ?state') clause)
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    by auto
    moreover
    have getM ?state'' = getM state ∧
      getF ?state'' = getF state ∧
      getSATFlag ?state'' = getSATFlag state ∧
      getQ ?state'' = getQ state
    unfolding swapWatches-def
    unfolding setWatch2-def
    by simp
    ultimately
    show ?thesis
    using Cons(1)[of ?state'' newWl]
    using Cons(3)
    using (getWatch1 ?state' clause = Some ?w1)
    using (getWatch2 ?state' clause = Some ?w2)
    using (Some literal = getWatch1 state clause)
    using (~ literalTrue ?w1 (elements (getM ?state')))
    using Some
    by (simp add: Let-def)

next
  case None
  show ?thesis
  proof (cases literalFalse ?w1 (elements (getM ?state')))
    case True
    let ?state'' = ?state''[getConflictFlag := True, getConflict-Clause := clause]

    from Cons(2)
    have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
    (getWatch2 ?state'')
unfolding $\text{InvariantWatchesEl-def}$
unfolding $\text{swapWatches-def}$
by auto
moreover
have $\text{getM ?state''} = \text{getM state} \land$
$\text{getF ?state''} = \text{getF state} \land$
$\text{getSATFlag ?state''} = \text{getSATFlag state} \land$
$\text{getQ ?state''} = \text{getQ state}$
unfolding $\text{swapWatches-def}$
by simp
ultimately
show $\text{?thesis}$
using $\text{Cons(1)[of ?state'' clause # newWI]}$
using $\text{Cons(3)}$
using $(\text{getWatch1 ?state' clause = Some ?w1})$
using $(\text{getWatch2 ?state' clause = Some ?w2})$
using $(\text{Some literal = getWatch1 state clause})$
using $(\neg \text{literalTrue ?w1 (elements (getM ?state'))})$
using None
using $(\text{literalFalse ?w1 (elements (getM ?state'))})$
by (simp add: \text{Let-def})
next
case False
let $\text{?state''} = \text{setReason ?w1 clause (?state'(getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1]))})$
from $\text{Cons(2)}$
have $\text{InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')}$
$(\text{getWatch2 ?state'''})$
unfolding $\text{InvariantWatchesEl-def}$
unfolding $\text{swapWatches-def}$
unfolding $\text{setReason-def}$
by auto
moreover
have $\text{getM ?state''} = \text{getM state} \land$
$\text{getF ?state''} = \text{getF state} \land$
$\text{getSATFlag ?state''} = \text{getSATFlag state} \land$
$\text{getQ ?state''} = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])$
unfolding $\text{swapWatches-def}$
unfolding $\text{setReason-def}$
by auto
ultimately
show $\text{?thesis}$
using $\text{Cons(1)[of ?state'' clause # newWI]}$
using $\text{Cons(3)}$
using $(\text{getWatch1 ?state' clause = Some ?w1})$
using $(\text{getWatch2 ?state' clause = Some ?w2})$
using $(\text{Some literal = getWatch1 state clause})$
using $(\neg \text{literalTrue ?w1 (elements (getM ?state'))})$
using None
using (¬ literalFalse ?w1 (elements (getM ?state′)))
unfolding isPrefix-def
by (auto simp add: Let-def split: split-if-asm)
qed
qed
qed
next
case False
let ?state′ = state
let ?w1 = wa
have getWatch1 ?state′ clause = Some ?w1
  using (getWatch1 state clause = Some wa)
unfolding swapWatches-def
by auto
let ?w2 = wb
have getWatch2 ?state′ clause = Some ?w2
  using (getWatch2 state clause = Some wb)
unfolding swapWatches-def
by auto
show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state′)))
case True
thus ?thesis
  using Cons
  using (¬ Some literal = getWatch1 state clause)
  using (getWatch1 ?state′ clause = Some ?w1);
  using (getWatch2 ?state′ clause = Some ?w2);
  using (literalTrue ?w1 (elements (getM ?state′)))
  by (simp add:Let-def)
next
case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state′) clause) ?w1 ?w2 (getM ?state′))
case (Some l′)
hence l′ el (nth (getF ?state′)) clause
  using getNonWatchedUnfalsifiedLiteralSomeCharacterization
  by simp
let ?state′′ = setWatch2 clause l′ ?state′

from Cons(2)
  have InvariantWatchesEl (getF ?state′′) (getWatch1 ?state′′)
  (getWatch2 ?state′)
  using (l′ el (nth (getF ?state′)) clause)
  unfolding InvariantWatchesEl-def
  unfolding setWatch2-def
  by auto
moreover
have \(\text{getM} \ ?\text{state}'' = \text{getM} \ \text{state} \land\)
    \(\text{getF} \ ?\text{state}'' = \text{getF} \ \text{state} \land\)
    \(\text{getSATFlag} \ ?\text{state}'' = \text{getSATFlag} \ \text{state} \land\)
    \(\text{getQ} \ ?\text{state}'' = \text{getQ} \ \text{state}\)
  unfolding setWatch2-def
  by simp
ultimately
show \(?\text{thesis}\)
  using Cons(1)[of \(?\text{state}'\)]
  using Cons(3)
  using (getWatch1 ?state' clause = Some \(?w1\))
  using (getWatch2 ?state' clause = Some \(?w2\))
  using (\(\sim\) Some literal = getWatch1 state clause)
  using (\(\sim\) literalTrue \(?w1\) (elements \(\text{getM} \ ?\text{state}'\))):
    using Some
    by (simp add: Let-def)

next
case None
show \(?\text{thesis}\)
proof (cases literalFalse \(?w1\) (elements \(\text{getM} \ ?\text{state}'\)))
case True
  let \(?\text{state}'' = ?\text{state}'[\text{getConflictFlag} := \text{True}, \text{getConflict-Clause} := \text{clause}]\)

from Cons(2)
have InvariantWatchesEl \(\text{getF} \ ?\text{state}'\) \(\text{getWatch1} \ ?\text{state}'\)
  (getWatch2 ?state'')
    unfolding InvariantWatchesEl-def
    by auto
moreover
have \(\text{getM} \ ?\text{state}'' = \text{getM} \ \text{state} \land\)
    \(\text{getF} \ ?\text{state}'' = \text{getF} \ \text{state} \land\)
    \(\text{getSATFlag} \ ?\text{state}'' = \text{getSATFlag} \ \text{state} \land\)
    \(\text{getQ} \ ?\text{state}'' = \text{getQ} \ \text{state}\)
    by simp
ultimately
show \(?\text{thesis}\)
  using Cons(1)[of \(?\text{state}'\)]
  using Cons(3)
  using (getWatch1 ?state' clause = Some \(?w1\))
  using (getWatch2 ?state' clause = Some \(?w2\))
  using (\(\sim\) Some literal = getWatch1 state clause)
  using (\(\sim\) literalTrue \(?w1\) (elements \(\text{getM} \ ?\text{state}'\))):
    using None
    using literalFalse \(?w1\) (elements \(\text{getM} \ ?\text{state}'\))
    by (simp add: Let-def)

next
case False
let \( ?\text{state}' = \text{setReason} (?,?w1 \text{ clause}(?,?w1 \text{ el}(?,?\text{getQ} ?\text{state}' \text{ then}(?,?\text{getQ} ?\text{state}' \text{ else}(?,?\text{getQ} ?\text{state}' @ ?w1)))) \)

from \( \text{Cons}(2) \)

have \( \text{InvariantWatchesEl} (\text{getF} ?\text{state}'') (\text{getWatch1} ?\text{state}'') \)
  unfolding \( \text{InvariantWatchesEl-def} \)
  unfolding \( \text{setReason-def} \)
  by \( \text{auto} \)

moreover

have \( \text{getM} ?\text{state}'' = \text{getM} \text{ state} \land \)
\( \text{getF} ?\text{state}'' = \text{getF} \text{ state} \land \)
\( \text{getSATFlag} ?\text{state}'' = \text{getSATFlag} \text{ state} \land \)
\( \text{getQ} ?\text{state}'' = (\text{if} ?w1 \text{ el}(?,?\text{getQ} \text{ state} \text{ then}(?,?\text{getQ} \text{ state} \text{ else}(?,?\text{getQ} \text{ state} @ ?w1)))) \)

  unfolding \( \text{setReason-def} \)
  by \( \text{simp} \)

ultimately

show \( ?\text{thesis} \)
  using \( \text{Cons}(1)[of ?\text{state}''] \)
  using \( \text{Cons}(3) \)
  using \( \text{(getWatch1} ?\text{state}' \text{ clause} = \text{Some} ?w1) \)
  using \( \text{(getWatch2} ?\text{state}' \text{ clause} = \text{Some} ?w2) \)
  using \( \text{(\neg \text{Some literal} = \text{getWatch1} \text{ state clause})} \)
  using \( \text{(\neg \text{literalTrue} ?w1 \text{ (elements (getM} ?\text{state}')})} \)
  using \( \text{None} \)
  using \( \text{(\neg \text{literalFalse} ?w1 \text{ (elements (getM} ?\text{state}')})} \)
  unfolding \( \text{isPrefix-def} \)
  by \( \text{(auto simp add: Let-def split: split-if-asm)} \)

qed
qed
qed
qed
qed

lemma notifyWatchesStartQIrelevent:

fixes \( \text{literal} :: \text{Literal} \text{ and} \ Wl :: \text{nat list} \text{ and} \ newWl :: \text{nat list} \text{ and} \ state :: \text{State} \)

assumes
  \( \text{InvariantWatchesEl} (\text{getF} \text{ stateA}) (\text{getWatch1} \text{ stateA}) (\text{getWatch2} \text{ stateA}) \text{ and} \)
  \( \forall \ (c::\text{nat}), \ c \in \text{set} \ Wl \longrightarrow \emptyset \leq c \land \ c < \text{length (getF} \text{ stateA}) \text{ and} \)
  getM \text{ stateA} = \text{getM} \text{ stateB} \text{ and} \)
  getF \text{ stateA} = \text{getF} \text{ stateB} \text{ and} \)
  getWatch1 \text{ stateA} = \text{getWatch1} \text{ stateB} \text{ and} \)
  getWatch2 \text{ stateA} = \text{getWatch2} \text{ stateB} \text{ and} \)
  getConflictFlag \text{ stateA} = \text{getConflictFlag} \text{ stateB} \text{ and} \)
  getSATFlag \text{ stateA} = \text{getSATFlag} \text{ stateB} \text{ and} \)

shows
let state' = (notifyWatches-loop literal Wl newWl stateA) in
let state'' = (notifyWatches-loop literal Wl newWl stateB) in
  (getM state') = (getM state'') ∧
  (getF state') = (getF state'') ∧
  (getSATFlag state') = (getSATFlag state'') ∧
  (getConflictFlag state') = (getConflictFlag state'')

using assms
proof (induct Wl arbitrary: newWl stateA stateB)
  case Nil
  thus ?case
  by simp
next
  case (Cons clause Wl)
  from ∀ (c::nat). c ∈ set (clause # Wl') → 0 ≤ c ∧ c < length (getF stateA)
  have 0 ≤ clause ∧ clause < length (getF stateA)
  by auto
  then obtain wa::Literal and wb::Literal
  where getWatch1 stateA clause = Some wa and getWatch2 stateA clause = Some wb
  using Cons
  unfolding InvariantWatchesEl-def
  by auto
  show ?case
proof (cases Some literal = getWatch1 stateA clause)
  case True
  hence Some literal = getWatch1 stateB clause
  using (getWatch1 stateA = getWatch1 stateB)
  by simp

let ?state'A = swapWatches clause stateA
let ?state'B = swapWatches clause stateB

have
  getM ?state'A = getM ?state'B
  getF ?state'A = getF ?state'B
  getWatch1 ?state'A = getWatch1 ?state'B
  getWatch2 ?state'A = getWatch2 ?state'B
  getConflictFlag ?state'A = getConflictFlag ?state'B
  getSATFlag ?state'A = getSATFlag ?state'B
  using Cons
  unfolding swapWatches-def
  by auto

let ?w1 = wb
have getWatch1 ?state'A clause = Some ?w1
  using (getWatch2 stateA clause = Some wb)
  unfolding swapWatches-def
by auto
hence getWatch1 ?state'B clause = Some ?w1
  using (getWatch1 ?state'A = getWatch1 ?state'B);
by simp
let ?w2 = wa
have getWatch2 ?state'A clause = Some ?w2
  using (getWatch1 stateA clause = Some wa);
unfolding swapWatches-def
by auto
hence getWatch2 ?state'B clause = Some ?w2
  using (getWatch2 ?state'A = getWatch2 ?state'B);
by simp

show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state'A)))
case True
  hence literalTrue ?w1 (elements (getM ?state'B))
    using (getM ?state'A = getM ?state'B);
by simp

from Cons(2)
  have InvariantWatchesEl (getF ?state'A) (getWatch1 ?state'A)
    (getWatch2 ?state'A)
      unfolding InvariantWatchesEl-def
      unfolding swapWatches-def
      by auto
moreover
  have getM ?state'A = getM stateA ∧
    getF ?state'A = getF stateA ∧
    getSATFlag ?state'A = getSATFlag stateA ∧
    getQ ?state'A = getQ stateA
      unfolding swapWatches-def
      by simp
moreover
  have getM ?state'B = getM stateB ∧
    getF ?state'B = getF stateB ∧
    getSATFlag ?state'B = getSATFlag stateB ∧
    getQ ?state'B = getQ stateB
      unfolding swapWatches-def
      by simp
ultimately
show ?thesis
  using Cons(1)[of ?state'A ?state'B clause ≠ newWl]
  using (getM ?state'A = getM ?state'B);
  using (getF ?state'A = getF ?state'B);
  using (getWatch1 ?state'A = getWatch1 ?state'B);
  using (getWatch2 ?state'A = getWatch2 ?state'B);
using \( \langle \text{getConflictFlag } ?\text{state}'A = \text{getConflictFlag } ?\text{state}'B \rangle \)
using \( \langle \text{getSATFlag } ?\text{state}'A = \text{getSATFlag } ?\text{state}'B \rangle \)
using \( \text{Cons}(3) \)
using \( \langle \text{getWatch1 } ?\text{state}'A \text{ clause } = \text{Some } ?w1 \rangle \)
using \( \langle \text{getWatch2 } ?\text{state}'A \text{ clause } = \text{Some } ?w2 \rangle \)
using \( \langle \text{getWatch1 } ?\text{state}'B \text{ clause } = \text{Some } ?w1 \rangle \)
using \( \langle \text{getWatch2 } ?\text{state}'B \text{ clause } = \text{Some } ?w2 \rangle \)
using \( \langle \text{Some literal } = \text{getWatch1 stateA clause} \rangle \)
using \( \langle \text{Some literal } = \text{getWatch1 stateB clause} \rangle \)
using \( \langle \text{literalTrue } ?w1 \text{ (elements (getM stateA'))} \rangle \)
using \( \langle \text{literalTrue } ?w1 \text{ (elements (getM stateB'))} \rangle \)
by (simp add:Let-def)

next

case False

hence \( \neg \text{literalTrue } ?w1 \text{ (elements (getM stateB'))} \)
using \( \langle \text{getM stateA'} = \text{getM stateB'} \rangle \)
by simp

show \(?\text{thesis}\)

proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF stateA') clause) ?w1 ?w2 (getM stateA'))

case (Some l')

hence getNonWatchedUnfalsifiedLiteral (nth (getF stateB') clause) ?w1 ?w2 (getM stateB') = Some l'
using \( \langle \text{getF stateA'} = \text{getF stateB'} \rangle \)
using \( \langle \text{getM stateA'} = \text{getM stateB'} \rangle \)
by simp

have l' el (nth (getF stateA') clause)
using Some
using getNonWatchedUnfalsifiedLiteralSomeCharacterization
by simp
hence l' el (nth (getF stateB') clause)
using \( \langle \text{getF stateA'} = \text{getF stateB'} \rangle \)
by simp

let ?\text{state}''A = setWatch2 clause l' ?\text{state}'A
let ?\text{state}''B = setWatch2 clause l' ?\text{state}'B

have
\( \text{getM state}''A = \text{getM state}''B \)
\( \text{getF state}''A = \text{getF state}''B \)
\( \text{getWatch1 state}''A = \text{getWatch1 state}''B \)
\( \text{getWatch2 state}''A = \text{getWatch2 state}''B \)
\( \text{getConflictFlag state}''A = \text{getConflictFlag state}''B \)
\( \text{getSATFlag state}''A = \text{getSATFlag state}''B \)
using \( \text{Cons} \)
unfolding setWatch2-def
unfolding swapWatches-def
by auto

from Cons(2)

have InvariantWatchesEl (getF ?state''A) (getWatch1 ?state''A) (getWatch2 ?state''A)
  using \( l' \) el (nth (getF ?state'A) clause):
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto

moreover
have getM ?state''A = getM stateA ∧
  getF ?state''A = getF stateA ∧
  getSATFlag ?state''A = getSATFlag stateA ∧
  getQ ?state''A = getQ stateA
  unfolding swapWatches-def
  unfolding setWatch2-def
  by simp

moreover
have getM ?state''B = getM stateB ∧
  getF ?state''B = getF stateB ∧
  getSATFlag ?state''B = getSATFlag stateB ∧
  getQ ?state''B = getQ stateB
  unfolding swapWatches-def
  unfolding setWatch2-def
  by simp

ultimately
show ?thesis
  using Cons(1)[of ?state''A ?state''B newWl]
  using (getM ?state''A = getM ?state''B)
  using (getF ?state''A = getF ?state''B)
  using (getWatch1 ?state''A = getWatch1 ?state''B)
  using (getWatch2 ?state''A = getWatch2 ?state''B)
  using (getConflictFlag ?state''A = getConflictFlag ?state''B)
  using (getSATFlag ?state''A = getSATFlag ?state''B)
  using Cons(3)
  using (getWatch1 ?state'A clause = Some ?w1)
  using (getWatch2 ?state'A clause = Some ?w2)
  using (getWatch1 ?state'B clause = Some ?w1)
  using (getWatch2 ?state'B clause = Some ?w2)
  using (Some literal = getWatch1 stateA clause)
  using (Some literal = getWatch1 stateB clause)
  using (= literalTrue ?w1 (elements (getM ?state'A)))
  using (= literalTrue ?w1 (elements (getM ?state'B)))
  using (getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A) clause) ?w1 ?w2 (getM ?state'A) = Some l1)
  using (getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B)
clause) (?w1 ?w2 (getM ?state'B) = Some l')
    by (simp add:Let-def)
next
  case None
  hence getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B) clause) (?w1 ?w2 (getM ?state'B) = None)
    by simp
show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state'A)))
  case True
  hence literalFalse ?w1 (elements (getM ?state'B))
    using (getM ?state'A = getM ?state'B)
    by simp
  have getM ?state''A = getM ?state''B
    getF ?state''A = getF ?state''B
    getWatch1 ?state''A = getWatch1 ?state''B
    getWatch2 ?state''A = getWatch2 ?state''B
    getConflictFlag ?state''A = getConflictFlag ?state''B
    getSATFlag ?state''A = getSATFlag ?state''B
    using Cons
    unfolding swapWatches-def
    by auto
  from Cons(2)
  have InvariantWatchesEl (getF ?state''A) (getWatch1 ?state''A)
    (getWatch2 ?state''A)
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    by auto
  moreover
  have getM ?state''A = getM ?stateA ∧
    getF ?state''A = getF ?stateA ∧
    getSATFlag ?state''A = getSATFlag ?stateA ∧
    getQ ?state''A = getQ ?stateA
    unfolding swapWatches-def
    by simp
  moreover
  have getM ?state''B = getM ?stateB ∧
    getF ?state''B = getF ?stateB ∧
    getSATFlag ?state''B = getSATFlag ?stateB ∧
    getQ ?state''B = getQ ?stateB
unfolding swapWatches-def
by simp
ultimately
show \( \text{thesis} \)
using Cons(4) Cons(5)
using Cons(1)[of \text{state}\''A \text{ state}\''B \text{ clause} \# \text{newWl}]
using (getM \text{state}\''A = getM \text{state}\''B)
using (getF \text{state}\''A = getF \text{state}\''B)
using (getWatch1 \text{state}\''A = getWatch1 \text{state}\''B)
using (getWatch2 \text{state}\''A = getWatch2 \text{state}\''B)
using (getConflictFlag \text{state}\''A = getConflictFlag \text{state}\''B)
using (getSATFlag \text{state}\''A = getSATFlag \text{state}\''B)
using Cons(3)
using (getWatch1 \text{state}'A \text{ clause} = \text{Some} \ ?w1)
using (getWatch2 \text{state}'A \text{ clause} = \text{Some} \ ?w2)
using (getWatch1 \text{state}'B \text{ clause} = \text{Some} \ ?w1)
using (getWatch2 \text{state}'B \text{ clause} = \text{Some} \ ?w2)
using (Some literal = getWatch1 \text{state}A \text{ clause})
using (Some literal = getWatch1 \text{state}B \text{ clause})
using (\neg \text{literalTrue} \ ?w1 \ (\text{elements} (getM \text{state}'A)))
using (\neg \text{literalTrue} \ ?w1 \ (\text{elements} (getM \text{state}'B))

by (simp add: Let-def)

next


case False

hence \( \neg \text{literalFalse} \ ?w1 \ (\text{elements} (getM \text{state}'B)) \)

using (getM \text{state}'A = \text{getM} \text{state}'B)

by simp

let \text{state}'A = setReason \ ?w1 \text{ clause} (\text{state}'A\{\text{getQ} := (\text{if} \ \ ?w1 \ \text{ el} \ (\text{getQ} \text{ state}'A) \ \text{then} \ (\text{getQ} \text{ state}'A) \ \text{else} \ (\text{getQ} \text{ state}'A) @ [\?w1])\})))

let \text{state}'B = setReason \ ?w1 \text{ clause} (\text{state}'B\{\text{getQ} := (\text{if} \ \ ?w1 \ \text{ el} \ (\text{getQ} \text{ state}'B) \ \text{then} \ (\text{getQ} \text{ state}'B) \ \text{else} \ (\text{getQ} \text{ state}'B) @ [\?w1])\})))

have

\begin{align*}
\text{getM} \text{state}'A & = \text{getM} \text{state}'B \\
\text{getF} \text{state}'A & = \text{getF} \text{state}'B \\
\text{getWatch1} \text{state}'A & = \text{getWatch1} \text{state}'B \\
\text{getWatch2} \text{state}'A & = \text{getWatch2} \text{state}'B \\
\text{getConflictFlag} \text{state}'A & = \text{getConflictFlag} \text{state}'B \\
\text{getSATFlag} \text{state}'A & = \text{getSATFlag} \text{state}'B \\
\text{using} \text{Cons} \\
\text{unfolding} \ \text{setReason-def}
\end{align*}
unfolding \texttt{swapWatches-def}
by \texttt{auto}

\textbf{from} Cons(2)
\textbf{have} \texttt{InvariantWatchesEl} (getF ?state"A) (getWatch1 ?state"A)
\texttt{(getWatch2 ?state"A)}
\unfolding \texttt{InvariantWatchesEl-def}
\unfolding \texttt{swapWatches-def}
\unfolding \texttt{setReason-def}
by \texttt{auto}
moreover
\textbf{have} getM ?state"A = getM stateA \land
getF ?state"A = getF stateA \land
getSATFlag ?state"A = getSATFlag stateA \land
getQ ?state"A = (if ?w1 el (getQ stateA) then (getQ stateA) else (getQ stateA) @ [?w1])
\unfolding \texttt{swapWatches-def}
\unfolding \texttt{setReason-def}
by \texttt{auto}
moreover
\textbf{have} getM ?state"B = getM stateB \land
getF ?state"B = getF stateB \land
getSATFlag ?state"B = getSATFlag stateB \land
getQ ?state"B = (if ?w1 el (getQ stateB) then (getQ stateB) else (getQ stateB) @ [?w1])
\unfolding \texttt{swapWatches-def}
\unfolding \texttt{setReason-def}
by \texttt{auto}
ultimately
\textbf{show} \texttt{thesis}
\textbf{using} Cons(4) Cons(5)
\textbf{using} Cons(1)[of \texttt{state"A} \texttt{state"B} \texttt{clause} \neq \texttt{newWl}]
\textbf{using} getM ?state"A = getM ?state"B;
\textbf{using} getF ?state"A = getF ?state"B;
\textbf{using} getWatch1 ?state"A = getWatch1 ?state"B;
\textbf{using} getWatch2 ?state"A = getWatch2 ?state"B;
\textbf{using} getConflctFlag ?state"A = getConflctFlag ?state"B;
\textbf{using} getSATFlag ?state"A = getSATFlag ?state"B;
\textbf{using} Cons(3)
\textbf{using} getWatch1 ?state'A clause = Some ?w1;
\textbf{using} getWatch2 ?state'A clause = Some ?w2;
\textbf{using} getWatch1 ?state'B clause = Some ?w1;
\textbf{using} getWatch2 ?state'B clause = Some ?w2;
\textbf{using} (Some literal = getWatch1 stateA clause);
\textbf{using} (Some literal = getWatch1 stateB clause);
\textbf{using} (¬ literalTrue ?w1 (elements (getM ?state'A)));
\textbf{using} (¬ literalTrue ?w1 (elements (getM ?state'B)));
\textbf{using} getNonWatchedUnfalsifiedLiteral (nth (getF ?state'A) clause) ?w1 ?w2 (getM ?state'A) = None)

336
using \(\langle\text{getNonWatchedUnfalsifiedLiteral (nth (getF \text{?state}\text{'B}) clause)} \rangle\) \(\langle\text{getM \text{?state}\text{'B}} = \text{None}\rangle\)
using \(\langle\neg \text{literalFalse \text{?w1 (elements (getM \text{?state}\text{'A}))}\rangle\)
using \(\langle\neg \text{literalFalse \text{?w1 (elements (getM \text{?state}\text{'B}))}\rangle\)
by (simp add:Let-def)
qed
qed
qed
next
case False
hence Some literal \(\neq\) getWatch1 \text{state}\text{'B} clause
using Cons
by simp

let \(?\text{state}\text{'A} = \text{state}\text{'A}\)
let \(?\text{state}\text{'B} = \text{state}\text{'B}\)

have
getM \(?\text{state}\text{'A} = \text{getM \text{?state}\text{'B}}\)
getF \(?\text{state}\text{'A} = \text{getF \text{?state}\text{'B}}\)
getWatch1 \(?\text{state}\text{'A} = \text{getWatch1 \text{?state}\text{'B}}\)
getWatch2 \(?\text{state}\text{'A} = \text{getWatch2 \text{?state}\text{'B}}\)
getConflictFlag \(?\text{state}\text{'A} = \text{getConflictFlag \text{?state}\text{'B}}\)
getSATFlag \(?\text{state}\text{'A} = \text{getSATFlag \text{?state}\text{'B}}\)
using Cons
by auto

let \(?\text{w1} = \text{wa}\)
have getWatch1 \(?\text{state}\text{'A} clause = \text{Some \text{?w1}}\)
using (getWatch1 state\text{'A} clause = \text{Some \text{wa}})
by auto

hence getWatch1 \(?\text{state}\text{'B} clause = \text{Some \text{?w1}}\)
using Cons
by simp

let \(?\text{w2} = \text{wb}\)
have getWatch2 \(?\text{state}\text{'A} clause = \text{Some \text{?w2}}\)
using (getWatch2 state\text{'A} clause = \text{Some \text{wb}})
by auto

hence getWatch2 \(?\text{state}\text{'B} clause = \text{Some \text{?w2}}\)
using Cons
by simp

show \(?\text{thesis}\)
proof (cases literalTrue \(?\text{w1 (elements (getM \text{?state}\text{'A)}))}\)
case True
hence literalTrue \(?\text{w1 (elements (getM \text{?state}\text{'B)}))}\)
using Cons
by simp

337
show ?thesis
  using Cons(1) of ?state' A ?state' B clause # newWl
  using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7) Cons(8) Cons(9)
using (¬ Some literal = getWatch1 state A clause)
using (¬ Some literal = getWatch1 state B clause)
using (getWatch1 ?state' A clause = Some ?w1)
using (getWatch1 ?state' B clause = Some ?w1)
using (getWatch2 ?state' A clause = Some ?w2)
using (getWatch2 ?state' B clause = Some ?w2)
using (literalTrue ?w1 (elements (getM ?state' A)) = Some ?w1)
using (literalTrue ?w1 (elements (getM ?state' B)) = Some ?w1)
by (simp add: Let-def)
next
case False
  hence (¬ literalTrue ?w1 (elements (getM ?state' B)))
  using (getM ?state' A = getM ?state' B)
  by simp
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state' A) clause) ?w1 ?w2 (getM ?state' A))
case (Some l')
  hence getNonWatchedUnfalsifiedLiteral (nth (getF ?state' B) clause) ?w1 ?w2 (getM ?state' B) = Some l'
  using (getF ?state' A = getF ?state' B)
  using (getM ?state' A = getM ?state' B)
  by simp
have l' el (nth (getF ?state' A) clause)
  using Some
  using getNonWatchedUnfalsifiedLiteralSomeCharacterization
  by simp
hence l' el (nth (getF ?state' B) clause)
  using (getF ?state' A = getF ?state' B)
  by simp
let ?state'' A = setWatch2 clause l' ?state' A
let ?state'' B = setWatch2 clause l' ?state' B
have
gtm ?state'' A = getM ?state'' B
gtf ?state'' A = getF ?state'' B
getWatch1 ?state'' A = getWatch1 ?state'' B
getWatch2 ?state'' A = getWatch2 ?state'' B
getConflictFlag ?state'' A = getConflictFlag ?state'' B
getSATFlag ?state'' A = getSATFlag ?state'' B
using Cons
unfolding setWatch2-def
by auto
from Cons(2)

have InvariantWatchesEl (getF ?state"A) (getWatch1 ?state"A)
  (getWatch2 ?state"A)
  using \l' el (nth (getF ?state′A) clause):
  unfolding InvariantWatchesEl-def
  unfolding setWatch2-def
  by auto

moreover have getM ?state"A = getM stateA ∧
  getF ?state"A = getF stateA ∧
  getSATFlag ?state"A = getSATFlag stateA ∧
  getQ ?state"A = getQ stateA
  unfolding setWatch2-def
  by simp

ultimately show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state′A)))
  case True
  hence
  unfolding getNonWatchedUnfalsifiedLiteral (nth (getF ?state′B) clause) ?w1 ?w2 (getM ?state′B) = None
  using (getF ?state′A = getF ?state′B) (getM ?state′A = getM ?state′B)
  by simp
  show ?thesis
  proof (cases literalFalse ?w1 (elements (getM ?state′A)))
    case True
hence literalFalse \( ?w1 \) \((\text{elements} \ (\text{getM} \ ?\text{state}'B))\)
  using \(\text{getM} \ ?\text{state}'A = \text{getM} \ ?\text{state}'B\)
  by simp

let \( ?\text{state}''A = ?\text{state}'A(\text{getConflictFlag} := \text{True}, \text{getConflict-Clause} := \text{clause})\)
  let \( ?\text{state}''B = ?\text{state}'B(\text{getConflictFlag} := \text{True}, \text{getConflictClause} := \text{clause})\)

  have 
    getM \ ?\text{state}''A = getM \ ?\text{state}''B 
    getF \ ?\text{state}''A = getF \ ?\text{state}''B 
    getWatch1 \ ?\text{state}''A = getWatch1 \ ?\text{state}''B 
    getWatch2 \ ?\text{state}''A = getWatch2 \ ?\text{state}''B 
    getConflictFlag \ ?\text{state}''A = getConflictFlag \ ?\text{state}''B 
    getSATFlag \ ?\text{state}''A = getSATFlag \ ?\text{state}''B 
  using Cons 
  by auto 

  from Cons(2) 
  have InvariantWatchesEl \((\text{getF} \ ?\text{state}''A) \ (\text{getWatch1} \ ?\text{state}''A)\) \((\text{getWatch2} \ ?\text{state}''A)\) 
    unfolding InvariantWatchesEl-def 
    by auto

  moreover 
  have getM \ ?\text{state}''A = getM \ ?\text{state}''A ∧ 
    getF \ ?\text{state}''A = getF \ ?\text{state}''A ∧ 
    getSATFlag \ ?\text{state}''A = getSATFlag \ ?\text{state}''A ∧ 
    getQ \ ?\text{state}''A = getQ \ ?\text{state}''A 
  by simp 

  ultimately 
  show \ ?\text{thesis} 
    using Cons(4) Cons(5) 
    using Cons(1) of \ ?\text{state}''A \ ?\text{state}''B \ ?\text{clause} ≠ \text{newWl} 
    using \(\text{getM} \ ?\text{state}''A = \text{getM} \ ?\text{state}''B\) 
    using \(\text{getF} \ ?\text{state}''A = \text{getF} \ ?\text{state}''B\) 
    using \(\text{getWatch1} \ ?\text{state}''A = \text{getWatch1} \ ?\text{state}''B\) 
    using \(\text{getWatch2} \ ?\text{state}''A = \text{getWatch2} \ ?\text{state}''B\) 
    using \(\text{getConflictFlag} \ ?\text{state}''A = \text{getConflictFlag} \ ?\text{state}''B\) 
    using \(\text{getSATFlag} \ ?\text{state}''A = \text{getSATFlag} \ ?\text{state}''B\) 
    using Cons(3) 
    using \(\text{getWatch1} \ ?\text{state}''A \ ?\text{clause} = \text{Some} \ ?w1\) 
    using \(\text{getWatch2} \ ?\text{state}''A \ ?\text{clause} = \text{Some} \ ?w2\) 
    using \(\text{getWatch1} \ ?\text{state}''B \ ?\text{clause} = \text{Some} \ ?w1\) 
    using \(\text{getWatch2} \ ?\text{state}''B \ ?\text{clause} = \text{Some} \ ?w2\) 
    using \(\text{\neg Some \ literal} = \text{getWatch1} \ ?\text{state}''A \ ?\text{clause}\) 
    using \(\text{\neg Some \ literal} = \text{getWatch1} \ ?\text{state}''B \ ?\text{clause}\) 
    using \(\text{\neg literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'A))\) 
    using \(\text{\neg literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'B))\) 
    using \(\text{getNonWatchedUnfalsifiedLiteral} \ (\text{nth} \ (\text{getF} \ ?\text{state}'A))\)
clause) ?w1 ?w2 (getM ?state'A) = None
  using ⟨getNonWatchedUnfalsifiedLiteral (nth (getF ?state'B) clause) ?w1 ?w2 (getM ?state'B) = None.
  using ⟨literalFalse ?w1 (elements (getM ?state'A))⟩;
  using ⟨literalFalse ?w1 (elements (getM ?state'B))⟩;
by (simp add:Let-def)

next
case False
  hence ¬ literalFalse ?w1 (elements (getM ?state'B))
  using ⟨getM ?state'B = getM ?state'B⟩
  by simp
  let ?state"A = setReason ?w1 clause (?state'A \{getQ := (if ?w1 el (getQ ?state'A) then (getQ ?state'A) else (getQ ?state'A) @ [?w1])\})
  let ?state"B = setReason ?w1 clause (?state'B \{getQ := (if ?w1 el (getQ ?state'B) then (getQ ?state'B) else (getQ ?state'B) @ [?w1])\})

have
  getM ?state"A = getM ?state"B
  getF ?state"A = getF ?state"B
  getWatch1 ?state"A = getWatch1 ?state"B
  getWatch2 ?state"A = getWatch2 ?state"B
  getConflictFlag ?state"A = getConflictFlag ?state"B
  getSATFlag ?state"A = getSATFlag ?state"B
using Cons
unfolding setReason-def
by auto

from Cons(2)
  unfolding InvariantWatchesEl-def
  unfolding setReason-def
  by auto
moreover
have getM ?state"A = getM ?state'A ∧
  getF ?state"A = getF ?state'A ∧
  getSATFlag ?state"A = getSATFlag ?state'A ∧
  getQ ?state"A = (if ?w1 el (getQ ?state'A) then (getQ ?state'A) else (getQ ?state'A) @ [?w1])
else (getQ ?state'A) @ [?w1])
  unfolding setReason-def
  by auto
ultimately
show ?thesis
using Cons(4) Cons(5)
using Cons(1)[of ?state"A ?state"B clause ≠ newWl]
using (getM ?state"A = getM ?state"B);
using (getF ?state"A = getF ?state"B)
using \( \text{getWatch1 ?state}'A = \text{getWatch1 ?state}'B \)
using \( \text{getWatch2 ?state}'A = \text{getWatch2 ?state}'B \)
using \( \text{getConflictFlag ?state}'A = \text{getConflictFlag ?state}'B \)
using \( \text{getSATFlag ?state}'A = \text{getSATFlag ?state}'B \)
using \( \text{Cons}(3) \)
using \( \text{getWatch1 ?state}'A \text{ clause } = \text{Some } ?w1 \)
using \( \text{getWatch2 ?state}'A \text{ clause } = \text{Some } ?w2 \)
using \( \text{getWatch1 ?state}'B \text{ clause } = \text{Some } ?w1 \)
using \( \text{getWatch2 ?state}'B \text{ clause } = \text{Some } ?w2 \)
using \( \neg \text{Some literal } = \text{getWatch1 stateA clause} \)
using \( \neg \text{Some literal } = \text{getWatch1 stateB clause} \)
using \( \neg \text{literalTrue } ?w1 \text{ (elements (getM ?state'A))} \)
using \( \neg \text{literalTrue } ?w1 \text{ (elements (getM ?state'B))} \)
using \( \text{getNonWatchedUnfalsifiedLiteral} (\text{nth (getF ?state'A) clause}) ?w1 ?w2 \text{ (getM ?state'A) } = \text{None} \)
using \( \text{getNonWatchedUnfalsifiedLiteral} (\text{nth (getF ?state'B) clause}) ?w1 ?w2 \text{ (getM ?state'B) } = \text{None} \)
using \( \neg \text{literalFalse } ?w1 \text{ (elements (getM ?state'A))} \)
using \( \neg \text{literalFalse } ?w1 \text{ (elements (getM ?state'B))} \)
by \( (\text{simp add: Let-def}) \)

\text{lemma notifyWatchesLoopPreservedWatches}:
\text{fixes literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State}
\text{assumes}
\qquad \text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}
\text{and}
\qquad \forall (c::nat). \ c \in \text{set Wl} \rightarrow \theta \leq c \land c < \text{length (getF state)}
\text{shows}
\qquad \text{let state'} = \text{notifyWatches-loop literal Wl newWl state} \text{ in}
\qquad \forall \ c. \ c \notin \text{set Wl} \rightarrow (\text{getWatch1 state'} c) = (\text{getWatch1 state} c)
\qquad \land (\text{getWatch2 state'} c) = (\text{getWatch2 state} c)

\text{using assms}
\text{proof (induct Wl arbitrary: newWl state)}
\qquad \text{case Nil}
\qquad \quad \text{thus } ?case
\qquad \qquad \text{by simp}
\qquad \text{next}
\qquad \text{case (Cons clause Wl')}
\qquad \quad \text{from } \forall (c::nat). \ c \in \text{set (clause # Wl')} \rightarrow \theta \leq c \land c < \text{length (getF state)}:
\qquad \quad \text{have } \theta \leq \text{clause} \land \text{clause} < \text{length (getF state)}
\qquad \qquad \text{by auto}
then obtain \( wa::\text{Literal} \) and \( wb::\text{Literal} \)
where \( \text{getWatch1 state clause} = \text{Some} \wa \) and \( \text{getWatch2 state clause} = \text{Some} \wb \)

using \( \text{Cons} \)

unfolding \( \text{InvariantWatchesEl-def} \)

by \( \text{auto} \)

show \( ?\text{case} \)

proof
\( \text{(cases Some literal = getWatch1 state clause)} \)

\( \text{case True} \)

let \( ?\text{state'} = \text{swapWatches clause state} \)

let \( ?\text{w1} = \wb \)

have \( \text{getWatch1 ?state'} \text{ clause} = \text{Some} ?\text{w1} \)

using \( \text{getWatch2 state clause} = \text{Some} \wb \)

unfolding \( \text{swapWatches-def} \)

by \( \text{auto} \)

let \( ?\text{w2} = \wa \)

have \( \text{getWatch2 ?state'} \text{ clause} = \text{Some} ?\text{w2} \)

using \( \text{getWatch1 state clause} = \text{Some} \wa \)

unfolding \( \text{swapWatches-def} \)

by \( \text{auto} \)

show \( ?\text{thesis} \)

proof
\( \text{(cases literalTrue ?w1 (elements (getM ?state')))} \)

\( \text{case True} \)

from \( \text{Cons}(2) \)

have \( \text{InvariantWatchesEl (getF ?state') (getWatch1 ?state')} \)

\( \text{(getWatch2 ?state')} \)

unfolding \( \text{InvariantWatchesEl-def} \)

unfolding \( \text{swapWatches-def} \)

by \( \text{auto} \)

moreover

have \( \text{getM ?state'} = \text{getM state} \) \land

\( \text{getF ?state'} = \text{getF state} \)

unfolding \( \text{swapWatches-def} \)

by \( \text{simp} \)

ultimately

show \( ?\text{thesis} \)

using \( \text{Cons}(1)[\text{of ?state' clause} \neq \text{newWl}] \)

using \( \text{Cons}(3) \)

using \( \text{(getWatch1 ?state' clause} = \text{Some} \?w1) \)

using \( \text{(getWatch2 ?state' clause} = \text{Some} \?w2) \)

using \( \text{(Some literal = getWatch1 state clause)} \)

using \( \text{literalTrue ?w1 (elements (getM ?state'))} \)

apply \( \text{(simp add:Let-def)} \)

unfolding \( \text{swapWatches-def} \)

by \( \text{simp} \)

next

\( \text{case False} \)

show \( ?\text{thesis} \)

343
\textbf{proof} \texttt{(cases \texttt{getNonWatchedUnfalsifiedLiteral} \texttt{(nth \texttt{(getF \texttt{?state'}) clause)) \texttt{?w1 \texttt{?w2} (getM \texttt{?state'})})

\texttt{case (Some \texttt{l'})
\texttt{hence \texttt{l' el (nth (getF \texttt{?state'}) clause)}
\texttt{using \texttt{getNonWatchedUnfalsifiedLiteralSomeCharacterization}}
\texttt{by simp}

\texttt{let \texttt{?state'' = setWatch2 clause \texttt{l' ?state'}}

\texttt{from Cons(2)
\texttt{have InvariantWatchesEl (getF \texttt{?state'')} (getWatch1 \texttt{?state'”})}}
\texttt{(getWatch2 \texttt{?state'”})
\texttt{using \texttt{\langle \texttt{l' el (nth (getF \texttt{?state'}) clause)\rangle}}
\texttt{unfolding InvariantWatchesEl-def}
\texttt{unfolding swapWatches-def}
\texttt{unfolding setWatch2-def}
\texttt{by auto}
\texttt{moreover
\texttt{have getM \texttt{?state'”} = getM \texttt{state} \wedge
\texttt{getF \texttt{?state'”} = getF \texttt{state}}
\texttt{unfolding swapWatches-def}
\texttt{unfolding setWatch2-def}
\texttt{by simp}
\texttt{ultimately
\texttt{show \texttt {?thesis}}
\texttt{using Cons(1)[of \texttt{?state'” newWl]}
\texttt{using Cons(3)}
\texttt{using \texttt{\langle getWatch1 \texttt{?state’ clause = Some \texttt{?w1}\rangle}}
\texttt{using \texttt{\langle getWatch2 \texttt{?state’ clause = Some \texttt{?w2}\rangle}}
\texttt{using \texttt{\langle Some literal = getWatch1 state clause\rangle}
\texttt{using \texttt{\langle \neg literalTrue \texttt{?w1 (elements (getM \texttt{?state'})\rangle}}
\texttt{using Some}
\texttt{apply (simp add: Let-def)}
\texttt{unfolding setWatch2-def}
\texttt{unfolding swapWatches-def}
\texttt{by simp}

\texttt{next}
\texttt{case None
\texttt{show \texttt {?thesis}}
\texttt{proof \texttt{(cases literalFalse \texttt{?w1 (elements (getM \texttt{?state'})})}
\texttt{case True
\texttt{let \texttt{?state'' = ?state'\langle getConflictFlag := True, getConflict-Clause := clause\rangle}}

\texttt{from Cons(2)
\texttt{have InvariantWatchesEl (getF \texttt{?state'’}) (getWatch1 \texttt{?state'”})}
\texttt{(getWatch2 \texttt{?state'”})
\texttt{unfolding InvariantWatchesEl-def}
\texttt{unfolding swapWatches-def}}
by auto
moreover
have \( \text{getM \ 'state'' = getM \ state} \wedge \)
\( \text{getF \ 'state'' = getF \ state} \)
  unfolding swapWatches-def
  by simp
ultimately
show \( ?\text{thesis} \)
  using \( \text{Cons(1)[of \ 'state'' clause \# \ newWl]} \)
  using \( \text{Cons(3)} \)
  using \( \text{getWatch1 \ 'state'} \text{ clause} = \text{Some \ ?w1} \)
  using \( \text{getWatch2 \ 'state'} \text{ clause} = \text{Some \ ?w2} \)
  using \( \text{Some \ literal} = \text{getWatch1 \ state \ clause} \)
  using \( \neg \text{literalTrue \ ?w1 \ (elements \ (getM \ 'state'}}> \)
  using \( \text{None} \)
  using \( \text{literalFalse \ ?w1 \ (elements \ (getM \ 'state'}}> \)
  apply \( \text{simp add: \ Let-def} \)
  unfolding swapWatches-def
by simp
next
case False
  let \( \text{?state'' = setReason \ ?w1 \ clause \ (\text{?state'}(\text{if \ ?w1 el \ (getQ \ ?state'}) then \ (getQ \ ?state') else \ (getQ \ ?state') @ [\text{?w1}])))} \)
from \( \text{Cons(2)} \)
have \( \text{InvariantWatchesEl \ (getF \ 'state'' \ (getWatch1 \ 'state'"> \)
(\text{getWatch2 \ 'state''}) \)
  unfolding \( \text{InvariantWatchesEl-def} \)
  unfolding swapWatches-def
  unfolding setReason-def
  by auto
moreover
have \( \text{getM \ 'state'' = getM \ state} \wedge \)
\( \text{getF \ 'state'' = getF \ state} \)
  unfolding swapWatches-def
  unfolding setReason-def
by simp
ultimately
show \( ?\text{thesis} \)
  using \( \text{Cons(1)[of \ 'state'' clause \# \ newWl]} \)
  using \( \text{Cons(3)} \)
  using \( \text{getWatch1 \ 'state'} \text{ clause} = \text{Some \ ?w1} \)
  using \( \text{getWatch2 \ 'state'} \text{ clause} = \text{Some \ ?w2} \)
  using \( \text{Some \ literal} = \text{getWatch1 \ state \ clause} \)
  using \( \neg \text{literalTrue \ ?w1 \ (elements \ (getM \ 'state)}> \)
  using \( \text{None} \)
  using \( \neg \text{literalFalse \ ?w1 \ (elements \ (getM \ 'state)}> \)
  apply \( \text{simp add: \ Let-def} \)
  unfolding setReason-def
unfolding swapWatches-def
by simp
case False
let ?state′ = state
let ?w1 = wa
have getWatch1 ?state′ clause = Some ?w1
  using (getWatch1 state clause = Some wa)
unfolding swapWatches-def
by auto
let ?w2 = wb
have getWatch2 ?state′ clause = Some ?w2
  using (getWatch2 state clause = Some wb)
unfolding swapWatches-def
by auto
have getWatch1 ?state′ clause = Some ?w1
  using ⟨getWatch1 state clause = Some wa⟩
  unfolding swapWatches-def
  by auto
have getWatch2 ?state′ clause = Some ?w2
  using ⟨getWatch2 state clause = Some wb⟩
  unfolding swapWatches-def
  by auto
show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state′)))
case True
thus ?thesis
  using Cons
  using ⟨¬ Some literal = getWatch1 state clause⟩
  using ⟨getWatch1 ?state′ clause = Some ?w1⟩
  using ⟨getWatch2 ?state′ clause = Some ?w2⟩
  using ⟨literalTrue ?w1 (elements (getM ?state′))⟩
  by (simp add:Let-def)
next
case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state′) clause) ?w1 ?w2 (getM ?state′))
case (Some l’)
hence l’ el (nth (getF ?state′) clause)
  using getNonWatchedUnfalsifiedLiteralSomeCharacterization
  by simp
let ?state′′ = setWatch2 clause l’ ?state′

from Cons(2)
  have InvariantWatchesEl (getF ?state′) (getWatch1 ?state′′)
  (getWatch2 ?state′′)
    using l’ el (nth (getF ?state′) clause)
    unfolding InvariantWatchesEl-def
    unfolding setWatch2-def
    by auto
moreover
have getM ?state′′ = getM state ∧
\[ \text{getF \ ?state'' = getF \ state} \]

unfolding setWatch2-def
by simp
ultimately
show \(?thesis
  using Cons(1)[of \ ?state'']
  using Cons(3)
  using (getWatch1 \ ?state' clause = Some \ ?w1)
  using (getWatch2 \ ?state' clause = Some \ ?w2)
  using (\neg \ Some \ literal = getWatch1 \ state \ clause)
  using (\neg \ literalTrue \ ?w1 \ (\text{elements (getM \ ?state')}))
  using Some
  apply (simp add: Let-def)
  unfolding setWatch2-def
by simp
next
  case None
  show \(?thesis
  proof (cases literalFalse \ ?w1 \ (\text{elements (getM \ ?state')}))
  case True
    let \?state'' = \?state'\[\{\text{getConflictFlag := True, getConflict-Clause := clause}\}\]

  from Cons(2)
  have InvariantWatchesEl (getF \ ?state'') (getWatch1 \ ?state'')
    (getWatch2 \ ?state'')
      unfolding InvariantWatchesEl-def
      by auto
  moreover
  have getM \ ?state'' = getM \ state ∧
      getF \ ?state'' = getF \ state
  by simp
  ultimately
  show \(?thesis
  using Cons(1)[of \ ?state'']
  using Cons(3)
  using (getWatch1 \ ?state' clause = Some \ ?w1)
  using (getWatch2 \ ?state' clause = Some \ ?w2)
  using (\neg \ Some \ literal = getWatch1 \ state \ clause)
  using (\neg \ literalTrue \ ?w1 \ (\text{elements (getM \ ?state')}))
  using None
  using (literalFalse \ ?w1 \ (\text{elements (getM \ ?state')}))
  by (simp add: Let-def)
next
  case False
  let \?state'' = setReason \ ?w1 \ clause \ (?state'\{(getQ := (if \ ?w1 el (getQ \ ?state') \ then \ (getQ \ ?state') \ else \ (getQ \ ?state') @ [\?w1]))\})

from Cons(2)
have InvariantWatchesEl (getF \ ?state'') (getWatch1 \ ?state'')
(getWatch2 ?state')
  unfolding InvariantWatchesEl-def
  unfolding setReason-def
  by auto
  moreover
  have getM ?state'' = getM state ∧
    getF ?state'' = getF state
  unfolding setReason-def
  by simp
  ultimately
  show ?thesis
    using Cons(1)[of ?state'']
    using Cons(3)
    using (getWatch1 ?state' clause = Some ?w1)
    using (getWatch2 ?state' clause = Some ?w2)
    using (∼ Some literal = getWatch1 state clause)
    using (∼ literalTrue ?w1 (elements (getM ?state')))
    using None
    using (∼ literalFalse ?w1 (elements (getM ?state')))
    apply (simp add: Let-def)
    unfolding setReason-def
    by simp
  qed
  qed
  qed
  qed

lemma InvariantWatchesElNotifyWatchesLoop:
  fixes literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State
  assumes
    InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
    and
    ∀ (c::nat). c ∈ set Wl −→ 0 ≤ c ∧ c < length (getF state)
  shows
    let state' = (notifyWatches-loop literal Wl newWl state) in
    InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state')
  using asms
proof (induct Wl arbitrary: newWl state)
  case Nil
  thus ?case
    by simp
next
  case (Cons clause Wl)
  from ∀ (c::nat). c ∈ set (clause # Wl) −→ 0 ≤ c ∧ c < length (getF state):
    have 0 ≤ clause and clause < length (getF state)
by auto
then obtain wa::Literal and wb::Literal
where getWatch1 state clause = Some wa and getWatch2 state clause = Some wb
using Cons
unfolding InvariantWatchesEl-def
by auto
show ?case
proof (cases Some literal = getWatch1 state clause)
  case True
  let ?state' = swapWatches clause state
  let ?w1 = wb
  have getWatch1 ?state' clause = Some ?w1
    using (getWatch2 state clause = Some wb)
    unfolding swapWatches-def
    by auto
  let ?w2 = wa
  have getWatch2 ?state' clause = Some ?w2
    using (getWatch1 state clause = Some wa)
    unfolding swapWatches-def
    by auto
show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state')))
  case True
  from Cons(2)
  have InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state')
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    by auto
moreover
have getF ?state' = getF state
  unfolding swapWatches-def
  by simp
ultimately
show ?thesis
  using Cons
  using (Some literal = getWatch1 state clause)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (literalTrue ?w1 (elements (getM ?state')))
  by (simp add: Let-def)
next
  case False
  show ?thesis
  proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
    case (Some l')
hence \( l' \) el \((\text{nth} (getF ?state') \text{ clause})\)
using \(\text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}\)
by \(\text{simp}\)

let \(?state'' = setWatch2 \text{ clause} l' \ ?state'\)

from \(\text{Cons}(2)\)

have \(\text{InvariantWatchesEl} \ (getF ?state'') \ (getWatch1 ?state'') \ (getWatch2 ?state'')\)
using \(\forall l' \ el \ (\text{nth} (getF \ ?state') \ \text{ clause})\)
unfolding \(\text{InvariantWatchesEl-def}\)
unfolding \(\text{swapWatches-def}\)
unfolding \(\text{setWatch2-def}\)
by \(\text{auto}\)
moreover
have \(getF ?state'' = getF \ ?state\)
unfolding \(\text{swapWatches-def}\)
unfolding \(\text{setWatch2-def}\)
by \(\text{simp}\)
ultimately
show \(?thesis\)
using \(\text{Cons}\)
using \(\langle \text{getWatch1 ?state'} \ \text{ clause} = \text{Some} \ ?w1 \rangle\)
using \(\langle \text{getWatch2 ?state'} \ \text{ clause} = \text{Some} \ ?w2 \rangle\)
using \(\langle \text{Some literal} = \text{getWatch1 \ state \ clause} \rangle\)
using \(\langle \neg \text{literalTrue ?w1 (elements (getM ?state'))} \rangle\)
using \(\text{Some}\)
by \((\text{simp add: Let-def})\)

next
case None
show \(?thesis\)
proof \((\text{cases literalFalse ?w1 (elements (getM ?state'))})\)
  case True
    let \(?state'' = ?state'\langle\text{getConflictFlag := True, getConflict-Clause := clause}\rangle\)

  from \(\text{Cons}(2)\)
  have \(\text{InvariantWatchesEl} \ (getF \ ?state'') \ (getWatch1 \ ?state'') \ (getWatch2 \ ?state'')\)
unfolding \(\text{InvariantWatchesEl-def}\)
unfolding \(\text{swapWatches-def}\)
by \(\text{auto}\)
moreover
have \(getF ?state'' = getF \ ?state\)
unfolding \(\text{swapWatches-def}\)
by \(\text{simp}\)
ultimately
show \(?thesis\)
using \(\text{Cons}\)
using \((\text{getWatch1} \ ?\text{state'} \ ?\text{clause} = \text{Some} \ ?w1)\)
using \((\text{getWatch2} \ ?\text{state'} \ ?\text{clause} = \text{Some} \ ?w2)\)
using \((\text{Some literal} = \text{getWatch1} \ ?\text{state} \ ?\text{clause})\)
using \((\neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))\)
using \(\text{None}\)
using \((\text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))\)
by (simp add: Let-def)

described

next
case False
let \(\text{state'}' = \text{setReason} \ ?w1 \ ?\text{clause} (\text{?state'}'(\text{getQ} := (\text{if} \ ?w1 \ \text{el} \ (\text{getQ} \ ?\text{state'}') \ \text{then} \ (\text{getQ} \ ?\text{state'}) \ \text{else} \ (\text{getQ} \ ?\text{state'}) \ @ \ [\ ?w1]))))\n
from \(\text{Cons}(2)\)
have \(\text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state'}) \ (\text{getWatch1} \ ?\text{state'}) \ (\text{getWatch2} \ ?\text{state'})\)
unfolding \(\text{InvariantWatchesEl-def}\)
unfolding \(\text{swapWatches-def}\)
unfolding \(\text{setReason-def}\)
by auto
moreover
have \(\text{getF} \ ?\text{state'}' = \text{getF} \ ?\text{state}\)
unfolding \(\text{swapWatches-def}\)
unfolding \(\text{setReason-def}\)
by simp
ultimately
show \(?\text{thesis}\)
using \(\text{Cons}\)
using \((\text{getWatch1} \ ?\text{state'} \ ?\text{clause} = \text{Some} \ ?w1)\)
using \((\text{getWatch2} \ ?\text{state'} \ ?\text{clause} = \text{Some} \ ?w2)\)
using \((\text{Some literal} = \text{getWatch1} \ ?\text{state} \ ?\text{clause})\)
using \((\neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))\)
using \(\text{None}\)
using \((\neg \text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))\)
by (simp add: Let-def)

qed

next
case False
let \(\text{?state'} = \text{state}\)
let \(?w1 = \text{wa}\)
have \(\text{getWatch1} \ ?\text{state'} \ ?\text{clause} = \text{Some} \ ?w1\)
using \((\text{getWatch1} \ ?\text{state} \ ?\text{clause} = \text{Some} \ \text{wa})\)
unfolding \(\text{swapWatches-def}\)
by auto
let \(?w2 = \text{wb}\)
have \(\text{getWatch2} \ ?\text{state'} \ ?\text{clause} = \text{Some} \ ?w2\)
using \((\text{getWatch2} \ ?\text{state} \ ?\text{clause} = \text{Some} \ \text{wb})\)
unfolding \(\text{swapWatches-def}\)

351
by auto

show \(?thesis

proof (cases literalTrue \(w1\) (elements (getM \(?state'\))))

  case True
  thus \(?thesis
    using Cons
    using (\(\neg\) Some literal = getWatch1 state clause)
    using (getWatch1 \(?state'\) clause = Some \(?w1\))
    using (getWatch2 \(?state'\) clause = Some \(?w2\))
    using (literalTrue \(w1\) (elements (getM \(?state'\))))
    by (simp add: Let-def)

next

  case False
  show \(?thesis

  proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF \(?state'\)) clause) \(?w1\) \(?w2\) (getM \(?state'\))))

    case (Some \(l'\))
    hence \(l'\) el (nth (getF \(?state'\)) clause)
    using getNonWatchedUnfalsifiedLiteralSomeCharacterization
    by simp

    let \(?state''\) = setWatch2 clause \(l'\) \(?state'\)

  from Cons
  have InvariantWatchesEl (getF \(?state''\)) (getWatch1 \(?state''\))
  (getWatch2 \(?state''\))
    using \(\neg\) el (nth (getF \(?state'\)) clause)
    unfolding InvariantWatchesEl-def
    unfolding setWatch2-def
    by auto

  moreover
  have getF \(?state''\) = getF state

  unfolding setWatch2-def
  by simp

  ultimately
  show \(?thesis

  using Cons
  using (getWatch1 \(?state'\) clause = Some \(?w1\))
  using (getWatch2 \(?state'\) clause = Some \(?w2\))
  using (\(\neg\) Some literal = getWatch1 state clause)
  using (\(\neg\) literalTrue \(w1\) (elements (getM \(?state'\))))
  using Some
  by (simp add: Let-def)

next

  case None
  show \(?thesis

  proof (cases literalFalse \(w1\) (elements (getM \(?state'\))))

  case True
    let \(?state''\) = \(?state'\) [getConflictFlag := True, getConflict-
Clause := clause

from Cons
have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  unfolding InvariantWatchesEl-def
  by auto
moreover
have getF ?state'' = getF state
  by simp
ultimately
show ?thesis
  using Cons
  using (getWatch1 ?state clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (¬ Some literal = getWatch1 state clause)
  using (¬ literalTrue ?w1 (elements (getM ?state')))
  using None
  using (¬ literalFalse ?w1 (elements (getM ?state')))
  by (simp add: Let-def)
next
case False
  let ?state'' = setReason ?w1 clause (?state' (getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ (?w1)]))
  from Cons(2)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  unfolding InvariantWatchesEl-def
  unfolding setReason-def
  by auto
moreover
have getF ?state'' = getF state
  unfolding setReason-def
  by simp
ultimately
show ?thesis
  using Cons
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (¬ Some literal = getWatch1 state clause)
  using (¬ literalTrue ?w1 (elements (getM ?state')))
  using None
  using (¬ literalFalse ?w1 (elements (getM ?state')))
  by (simp add: Let-def)
qed
qed
qed
qed
353
lemma InvariantWatchesDifferNotifyWatchesLoop:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
\( \forall (c::nat). c \in \text{set } Wl \longrightarrow 0 \leq c \land c < \text{length } (getF state) \)
shows
let state' = (notifyWatches-loop literal Wl newWl state) in
InvariantWatchesDiffer (getF state') (getWatch1 state') (getWatch2 state')
using assms
proof (induct Wl arbitrary: newWl state)
  case Nil
  thus ?case by simp
next
  case (Cons clause Wl')
  from \( \forall (c::nat). c \in \text{set } \text{(clause } \# Wl') \longrightarrow 0 \leq c \land c < \text{length } (getF state) \):
  have \( 0 \leq \text{clause } \land \text{clause } < \text{length } (getF state) \)
    by auto
  then obtain wa::Literal and wb::Literal
    where getWatch1 state clause = Some wa and getWatch2 state clause = Some wb
    using Cons
    unfolding InvariantWatchesEl-def
    by auto
  show ?case
  proof (cases Some literal = getWatch1 state clause)
  case True
  let ?state' = swapWatches clause state
  let ?w1 = wb
  have getWatch1 ?state' clause = Some ?w1
    using (getWatch2 state clause = Some wb)
    unfolding swapWatches-def
    by auto
  let ?w2 = wa
  have getWatch2 ?state' clause = Some ?w2
    using (getWatch1 state clause = Some wa)
    unfolding swapWatches-def
    by auto
  show ?thesis
  proof (cases literalTrue ?w1 (elements (getM ?state')))
  case True

354
from Cons(2)

have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
  (getWatch2 ?state')
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto

moreover
from Cons(3)

have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state')
  (getWatch2 ?state')
  unfolding InvariantWatchesDiffer-def
  unfolding swapWatches-def
  by auto

moreover
have getF ?state' = getF state
  unfolding swapWatches-def
  by simp

ultimately
show ?thesis
  using Cons(1)[of ?state' clause ≠ newWl]
  using Cons(4)
  using ⟨Some literal = getWatch1 state clause⟩
  using ⟨getWatch1 ?state' clause = Some ?w1⟩
  using ⟨getWatch2 ?state' clause = Some ?w2⟩
  using ⟨literalTrue ?w1 (elements (getM ?state'))⟩
  by (simp add: Let-def)

next
  case False

  show ?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
      clause) ?w1 ?w2 (getM ?state'))
      case (Some l')
      hence l' el (nth (getF ?state') clause) l' ≠ literal l' ≠ ?w1 l'
      ≠ ?w2
      using getNonWatchedUnfalsifiedLiteralSomeCharacterization
      using ⟨getWatch1 ?state' clause = Some ?w1⟩
      using ⟨getWatch2 ?state' clause = Some ?w2⟩
      using ⟨Some literal = getWatch1 state clause⟩
      unfolding swapWatches-def
      by auto

    let ?state'' = setWatch2 clause l' ?state'

from Cons(2)

have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def

355
unfolding setWatch2-def
by auto
moreover
from Cons(3)
have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  using (l' \neq ?w1)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  unfolding InvariantWatchesDiffer-def
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto
moreover
have getF ?state'' = getF state
  unfolding swapWatches-def
  unfolding setWatch2-def
  by simp
ultimately
show ?thesis
  using Cons
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (\neg literalTrue ?w1 (elements (getM ?state')))
  using Some
  by (simp add: Let-def)
next
case None
show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state')))
case True
  let ?state'' = ?state' (getConflictFlag := True, getConflict-Clause := clause)

from Cons(2)
have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto
moreover
from Cons(3)
have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  unfolding InvariantWatchesDiffer-def
  unfolding swapWatches-def
  by auto
moreover
have \( \text{getF 'state''} = \text{getF state} \)

unfolding swapWatches-def
by simp
ultimately
show \(?\text{thesis}\)
using Cons
using \(\text{getWatch1 'state' clause} = \text{Some ?w1}\) 
using \(\text{getWatch2 'state' clause} = \text{Some ?w2}\)
using \(\text{Some literal} = \text{getWatch1 state clause}\)
using \(\neg \text{literalTrue ?w1 (elements (getM 'state'))}\)
using None
using \(\text{literalFalse ?w1 (elements (getM ?state'))}\)
by (simp add: Let-def)

next
case False
let \(?\text{state''} = \text{setReason ?w1 clause (getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1]))}\)

from Cons(2)
have InvariantWatchesEl \(\text{(getF ?state')}\) \(\text{(getWatch1 ?state''})\)
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
unfolding setReason-def
by auto
moreover
from Cons(3)
have InvariantWatchesDiffer \(\text{(getF ?state')}\) \(\text{(getWatch1 ?state''})\)
unfolding InvariantWatchesDiffer-def
unfolding swapWatches-def
unfolding setReason-def
by auto
moreover
have \(\text{getF ?state''} = \text{getF state} \)
unfolding swapWatches-def
unfolding setReason-def
by simp
ultimately
show \(?\text{thesis}\)
using Cons
using \(\text{getWatch1 'state' clause} = \text{Some ?w1}\) 
using \(\text{getWatch2 'state' clause} = \text{Some ?w2}\)
using \(\text{Some literal} = \text{getWatch1 state clause}\)
using \(\neg \text{literalTrue ?w1 (elements (getM 'state'))}\)
using None
using \(\neg \text{literalFalse ?w1 (elements (getM ?state'))}\)
by (simp add: Let-def)
qed
qed
qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using ⟨getWatch1 state clause = Some wa⟩
  unfolding swapWatches-def
  by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using ⟨getWatch2 state clause = Some wb⟩
  unfolding swapWatches-def
  by auto
  show ?thesis
  proof (cases literalTrue ?w1 (elements (getM ?state')))
    case True
    thus ?thesis
      using Cons
      using ⟨¬ Some literal = getWatch1 state clause⟩
      using ⟨getWatch1 ?state' clause = Some ?w1⟩
      using ⟨getWatch2 ?state' clause = Some ?w2⟩
      using ⟨literalTrue ?w1 (elements (getM ?state'))⟩
      by (simp add: Let-def)
  next
  case False
  show ?thesis
  proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
    case (Some l')
    hence l' el (nth (getF ?state') clause) l' ≠ ?w1 l' ≠ ?w2
      using getNonWatchedUnfalsifiedLiteralSomeCharacterization
      using ⟨getWatch1 ?state' clause = Some ?w1⟩
      using ⟨getWatch2 ?state' clause = Some ?w2⟩
    unfolding swapWatches-def
    by auto
    let ?state'' = setWatch2 clause l' ?state'

    from Cons(2)
    have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
      (getWatch2 ?state'')
      using ⟨l' el (nth (getF ?state') clause)⟩
    unfolding InvariantWatchesEl-def
    unfolding setWatch2-def
    by auto
    moreover
    from Cons(3)
have InvariantWatchesDiffer (getF ?state"") (getWatch1 ?state"")
  (getWatch2 ?state"")
  unfolding InvariantWatchesDiffer-def
  unfolding InvariantWatchesEl-def
  unfolding setWatch2-def
  by auto
moreover
have getF ?state"") = getF state
  unfolding setWatch2-def
  by simp
ultimately
show ?thesis
  using Cons
  using (getWatch1 ?state clause = Some ?w1)
  using (getWatch2 ?state clause = Some ?w2)
  using (¬ Some literal = getWatch1 state clause)
  using (¬ literalTrue ?w1 (elements (getM ?state]))
  using Some
  by (simp add: Let-def)
next
case None
show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state)))
case True
  let ?state"" = ?state"(getConflictFlag := True, getConflict-Clause := clause)
  from Cons(2)
  have InvariantWatchesEl (getF ?state"") (getWatch1 ?state"")
    (getWatch2 ?state"")
    unfolding InvariantWatchesEl-def
    by auto
moreover
  from Cons(3)
  have InvariantWatchesDiffer (getF ?state"") (getWatch1 ?state"")
    (getWatch2 ?state"")
    unfolding InvariantWatchesDiffer-def
    by auto
moreover
have getF ?state"") = getF state
  by simp
ultimately
show ?thesis
  using Cons
  using (getWatch1 ?state clause = Some ?w1)
  using (getWatch2 ?state clause = Some ?w2)
  using (¬ Some literal = getWatch1 state clause)
using \(\neg \text{literalTrue } w_1 \ (\text{elements } (\text{getM } \ ?\text{state}'))\)
using None
using \(\neg \text{literalFalse } w_1 \ (\text{elements } (\text{getM } \ ?\text{state}'))\)
by (simp add: Let-def)
next
case False
let \(?\text{state}'' = \text{setReason } w_1 \ ?\text{clause } (?\text{state}' (\text{getQ } := (\text{if } w_1 \\
\text{ el } (\text{getQ } \ ?\text{state}'')) \ then \ (\text{getQ } \ ?\text{state}'') \ else \ (\text{getQ } \ ?\text{state}'') \ @ \ ?[w_1])))\)

from Cons(2)
have InvariantWatchesEl (getF ?\text{state}'' ) (getWatch1 \ ?\text{state}'') \ (getWatch2 \ ?\text{state}'')
  unfolding InvariantWatchesEl-def
  unfolding setReason-def
  by auto
moreover
from Cons(3)
  have InvariantWatchesDiffer (getF ?\text{state}'' ) (getWatch1 \\
\ ?\text{state}'') \ (getWatch2 \ ?\text{state}'')
  unfolding InvariantWatchesDiffer-def
  unfolding setReason-def
  by auto
moreover
have getF ?\text{state}'' = getF \text{state}
  unfolding setReason-def
  by simp
ultimately
show \(?\text{thesis}\)
  using Cons
  using (getWatch1 \ ?\text{state}' clause = Some \ ?w_1)
  using (getWatch2 \ ?\text{state}' clause = Some \ ?w_2)
  using \(\neg \text{Some literal } = \text{getWatch1 } \ ?\text{state} \ ?\text{clause}\)
  using \(\neg \text{literalTrue } w_1 \ (\text{elements } (\text{getM } \ ?\text{state}'))\)
  using None
  using \(\neg \text{literalFalse } w_1 \ (\text{elements } (\text{getM } \ ?\text{state}'))\)
  by (simp add: Let-def)
qed

lemma InvariantWatchListsContainOnlyClausesFromFNotifyWatches-Loop:
fixes literal :: Literal and \ Wl :: nat list and newWl :: nat list and \ state :: State
assumes
InvariantWatchListsContainOnlyClausesFromF \ (getWatchList \ \text{state})
\((\text{getF state})\) and 
\(\text{InvariantWatchesEl} (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state})\)
and 
\(\forall (c::\text{nat}). \ c \in \text{set} \ Wl \lor \ c \in \text{set} \ \text{newWl} \rightarrow 0 \leq c \land c < \text{length} \ (\text{getF state})\)
shows 
let state' = (notifyWatches-loop literal Wl \text{newWl state}) in 
\(\text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList state'}) (\text{getF state'})\)
using assms
proof (induct Wl arbitrary: \text{newWl state})
case Nil 
thus \(?case
unfolding \text{InvariantWatchListsContainOnlyClausesFromF-def}
by simp
next 
case (\text{Cons clause Wl}')
from \(\forall \ c. \ c \in \text{set} \ (\text{clause} \ # \ Wl') \lor \ c \in \text{set} \ \text{newWl} \rightarrow 0 \leq c \land c < \text{length} \ (\text{getF state})\)
have \(0 \leq \text{clause} \land \text{clause} < \text{length} \ (\text{getF state})\)
by auto
then obtain \(\text{wa::Literal} \land \text{wb::Literal}\)
where \(\text{getWatch1 state clause} = \text{Some wa} \land \text{getWatch2 state clause} = \text{Some wb}\)
using \(\text{Cons}\)
unfolding \(\text{InvariantWatchesEl-def}\)
by auto
show \(?case
proof (cases \text{Some literal} = \text{getWatch1 state clause})
case True
let \(\?state' = \text{swapWatches clause state}\)
let \(?w1 = \text{wb}\)
have \(\text{getWatch1 ?state'} \text{ clause} = \text{Some ??w1}\)
using \(\text{getWatch2 state clause} = \text{Some wb}\)
unfolding \(\text{swapWatches-def}\)
by auto
let \(?w2 = \text{wa}\)
have \(\text{getWatch2 ?state'} \text{ clause} = \text{Some ??w2}\)
using \(\text{getWatch1 state clause} = \text{Some wa}\)
unfolding \(\text{swapWatches-def}\)
by auto
show \(?thesis
proof (cases \text{literalTrue ??w1} (\text{elements (getM ?state')}))
case True 
from \(\text{Cons(2)}\)
have \(\text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList ?state'}) (\text{getF ?state'})\)
unfolding \(\text{swapWatches-def}\)
by auto
moreover
from Cons(3)
  have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
  (getWatch2 ?state')
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto
moreover
have (getF state) = (getF ?state')
  unfolding swapWatches-def
  by simp
ultimately
show ?thesis
  using Cons
  using (Some literal = getWatch1 state clause)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (literalTrue ?w1 (elements (getM ?state'))
    by (simp add: Let-def)
next
case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
clause) ?w1 ?w2 (getM ?state'))
case (Some l')
  hence l' el (nth (getF ?state') clause)
    using getNonWatchedUnfalsifiedLiteralSomeCharacterization
    by simp
let ?state'' = setWatch2 clause l' ?state'
from Cons(2)
have InvariantWatchListsContainOnlyClausesFromF (getWatchList
?state'') (getF ?state'')
  using (clause < length (getF state))
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto
moreover
from Cons(3)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  using l' el (nth (getF ?state') clause)
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto
moreover
have \((\text{getF state}) = (\text{getF ?state}')\)
  unfolding swapWatches-def
  unfolding setWatch2-def
by simp
ultimately
show \(?\text{thesis}\)
  using Cons
  using (\text{getWatch1 ?state'} clause = Some ?w1)
  using (\text{getWatch2 ?state'} clause = Some ?w2)
  using (\text{Some literal} = \text{getWatch1 state clause})
  using (\text{\neg literalTrue ?w1} \ (\text{elements (getM ?state')}));
  using Some
by (simp add: Let-def)
next
case None
show \(?\text{thesis}\)
proof (cases literalFalse ?w1 \ (\text{elements (getM ?state')}))
case True
  let \(?\text{state''} = \text{?state'}[]\text{getConflictFlag} := \text{True}, \text{getConflict-Clause} := \text{clause}[]\)
from Cons(2)
  have \(\text{InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'')} \ (\text{getF ?state'''})\)
    unfolding swapWatches-def
by auto
moreover
from Cons(3)
  have \(\text{InvariantWatchesEl (getF ?state'')} \ (\text{getWatch1 ?state'''})\)
    \ (\text{getWatch2 ?state'''})
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
by auto
moreover
have \((\text{getF state}) = (\text{getF ?state'''})\)
  unfolding swapWatches-def
by simp
ultimately
show \(?\text{thesis}\)
  using Cons
  using (\text{getWatch1 ?state'} clause = Some ?w1)
  using (\text{getWatch2 ?state'} clause = Some ?w2)
  using (\text{Some literal} = \text{getWatch1 state clause})
  using (\text{\neg literalTrue ?w1} \ (\text{elements (getM ?state')}));
  using None
  using (\text{literalFalse ?w1} \ (\text{elements (getM ?state')}));
by (simp add: Let-def)
next

363
case False
let ?state′′ = setReason ?w1 clause (?state′(getQ := (if ?w1
el (getQ ?state′) then (getQ ?state′) else (getQ ?state′) @ [?w1])))

from Cons(2)
have InvariantWatchListsContainOnlyClausesFromF (getWatchList
?state′′) (getF ?state′′)
  unfolding swapWatches-def
  unfolding setReason-def
  by auto
moreover
from Cons(3)
have InvariantWatchesEl (getF ?state′′) (getWatch1 ?state′′)
  (getWatch2 ?state′′)
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  unfolding setReason-def
  by auto
moreover
have (getF state) = (getF ?state′′)
  unfolding swapWatches-def
  unfolding setReason-def
  by simp
ultimately
show ?thesis
  using Cons
  using (getWatch1 ?state′ clause = Some ?w1)
  using (getWatch2 ?state′ clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (∼ literalTrue ?w1 (elements (getM ?state′)))
  using None
  using (∼ literalFalse ?w1 (elements (getM ?state′)))
  by (simp add: Let-def)
qed
qed
qed
next

case False
let ?state′ = state
let ?w1 = wa
have getWatch1 ?state′ clause = Some ?w1
  using (getWatch1 state clause = Some wa)
  unfolding swapWatches-def
  by auto
let ?w2 = wb
have getWatch2 ?state′ clause = Some ?w2
  using (getWatch2 state clause = Some wb)
  unfolding swapWatches-def
  by auto
show \(?thesis \\
proof \(\text{(cases literalTrue \(?w1\) (elements (getM \(\text{\(?state\)'}\)))}\) \\
\quad \text{case} \; \text{True} \\
\quad \text{thus} \; \(?thesis \\
\quad \quad \text{using} \; \text{Cons} \\
\quad \quad \text{using} \; (\neg \; \text{Some literal} = \text{getWatch1 state clause}) \\
\quad \quad \text{using} \; \text{getWatch1 \(\text{\(?state\)'}\) clause} = \text{Some \(?w1\)}; \\
\quad \quad \text{using} \; (\text{getWatch2 \(\text{\(?state\)'}\) clause} = \text{Some \(?w2\}) \\
\quad \quad \text{using} \; \text{literalTrue \(?w1\) (elements (getM \(\text{\(?state\)'}\)))} \\
\quad \quad \text{by} \; (\text{simp add:Let-def}) \\
\quad \text{next} \\
\quad \text{case} \; \text{False} \\
\quad \text{show} \; \(?thesis \\
\quad \quad \text{proof} \; \text{(cases getNonWatchedUnfalsifiedLiteral (nth (getF \(\text{\(?state\)'}\)) \(\text{\(?w1\)} \?w2\) (getM \(\text{\(?state\)'}\)))} \\
\quad \quad \text{case} \; (\text{Some \(l'\)}) \\
\quad \quad \text{hence} \; l' \; \text{el (nth (getF \(\text{\(?state\)'}\)) \(\text{\(?w1\)} \?w2\))} \\
\quad \quad \text{using} \; \text{getNonWatchedUnfalsifiedLiteralSomeCharacterization} \\
\quad \quad \text{by} \; \text{simp} \\
\text{let} \; \text{\(?state'\)'} = \text{setWatch2 clause} \; l' \; ?state' \\
\text{from} \; \text{Cons(2)} \\
\text{have} \; \text{InvariantWatchListsContainOnlyClausesFromF (getWatchList \(\text{\(?state'\)'}\)) (getF \(\text{\(?state\)'}\))} \\
\text{using} \; (\text{clause < length (getF state)}) \\
\text{unfolding} \; \text{setWatch2-def} \\
\text{unfolding} \; \text{InvariantWatchListsContainOnlyClausesFromF-def} \\
\text{by} \; \text{auto} \\
\text{moreover} \\
\text{from} \; \text{Cons(3)} \\
\text{have} \; \text{InvariantWatchesEl (getF \(\text{\(?state'\)'}\)) (getWatch1 \(\text{\(?state'\)'}\)) (getWatch2 \(\text{\(?state'\)'}\))} \\
\text{using} \; (\text{\(l'\) el (nth (getF \(\text{\(?state\)'}\)) \(\text{\(?w1\)} \?w2\))}) \\
\text{unfolding} \; \text{InvariantWatchesEl-def} \\
\text{unfolding} \; \text{setWatch2-def} \\
\text{by} \; \text{auto} \\
\text{moreover} \\
\text{have} \; (\text{getF state}) = (\text{getF \(\text{\(?state'\)'}\)}) \\
\text{unfolding} \; \text{setWatch2-def} \\
\text{by} \; \text{simp} \\
\text{ultimately} \\
\text{show} \; \(?thesis \\
\text{using} \; \text{Cons} \\
\text{using} \; (\text{getWatch1 \(\text{\(?state\)'}\) clause} = \text{Some \(?w1\)}) \\
\text{using} \; (\text{getWatch2 \(\text{\(?state\)'}\) clause} = \text{Some \(?w2\}) \\
\text{using} \; (\neg \; \text{Some literal} = \text{getWatch1 state clause}) \\
\text{using} \; (\neg \; \text{literalTrue \(?w1\) (elements (getM \(\text{\(?state\)'}\)))} \\
\text{using} \; \text{Some} 

365
by (simp add: Let-def)
next
  case None
  show ?thesis
  proof
    (cases literalFalse ?w1 (elements (getM ?state')))
    case True
    let ?state" = ?state' [getConflictFlag := True, getConflict-Clause := clause]
    from Cons(3)
    have InvariantWatchesEl (getF ?state") (getWatch1 ?state")
      (getWatch2 ?state")
      unfolding InvariantWatchesEl-def
      by auto
    moreover
    have getF ?state" = getF state
      by simp
    ultimately
    show ?thesis
      using Cons
      using (getWatch1 ?state' clause = Some ?w1)
      using (getWatch2 ?state' clause = Some ?w2)
      using (~ Some literal = getWatch1 state clause)
      using (~ literalTrue ?w1 (elements (getM ?state')))
      using None
      using (literalFalse ?w1 (elements (getM ?state')))
      by (simp add: Let-def)
next
  case False
  let ?state" = setReason ?w1 clause (?state' (getQ := (if ?w1
    el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])))
  from Cons(3)
  have InvariantWatchListsContainOnlyClausesFromF (getWatchList
    ?state") (getF ?state")
    unfolding setReason-def
    by auto
  moreover
  from Cons(3)
  have InvariantWatchesEl (getF ?state") (getWatch1 ?state")
    (getWatch2 ?state")
    unfolding InvariantWatchesEl-def
    unfolding setReason-def
    by auto
  moreover
  have getF ?state" = getF state
    unfolding setReason-def
    by simp
  ultimately
show ?thesis
  using Cons
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (\Some literal = getWatch1 state clause)
  using (\literalTrue ?w1 (elements (getM ?state')))
  using None
  using (\literalFalse ?w1 (elements (getM ?state')))
  by (simp add: Let-def)
qed
qed
qed
qed
qed

lemma InvariantWatchListsCharacterizationNotifyWatchesLoop:
  fixes literal :: Literal and Wl :: nat list and newWl :: nat list and
state :: State
  assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchListsUniq (getWatchList state)
  \forall \(c::\text{nat}) \cdot c \in \text{set Wl} \longrightarrow 0 \leq c \land c < \text{length (getF state)}
  \forall \(c::\text{nat}) \cdot \(l::\text{Literal}) \cdot l \neq \text{literal} \longrightarrow
  (c \in \text{set (getWatchList state l)}) = (\Some l = \text{getWatch1 state c})
\forall \(c::\text{nat}) \cdot \(c \in \text{set newWl} \lor c \in \text{set Wl}) = (\Some literal = (\text{getWatch1 state c}) \lor \Some literal = (\text{getWatch2 state c}))
set Wl \cap set newWl = {}
uniq Wl
uniq newWl
shows
let state' = (notifyWatches-loop literal Wl newWl state) in
  InvariantWatchListsCharacterization (getWatchList state') (getWatch1 state') (getWatch2 state') \land
  InvariantWatchListsUniq (getWatchList state')
using assms
proof (induct Wl arbitrary: newWl state)
case Nil
  thus ?case
  unfolding InvariantWatchListsCharacterization-def
  unfolding InvariantWatchListsUniq-def
  by simp
next
case \(\text{(Cons clause Wl')}\)
from \(\text{uniq (clause \# Wl')}\)
have clause \(\notin \text{set Wl'}\)
by (simp add: uniqAppendIff)

have set Wl' ∩ set (clause # newWl) = {}
  using Cons(8)
  using ⟨clause ∉ set Wl'⟩
  by simp

have uniq Wl'
  using Cons(9)
  using uniqAppendIff
  by simp

have uniq (clause ≠ newWl)
  using Cons(10) Cons(8)
  using uniqAppendIff
  by force

from ∀ c. c ∈ set (clause # Wl') → 0 ≤ c ∧ c < length (getF state)
have 0 ≤ clause and clause < length (getF state)
  by auto
then obtain wa::Literal and wb::Literal
  where getWatch1 state clause = Some wa and getWatch2 state clause = Some wb
  using Cons
  unfolding InvariantWatchesEl-def
  by auto
show ?case
proof (cases Some literal = getWatch1 state clause)
  case True
  let ?state' = swapWatches clause state
  let ?w1 = wb
  have getWatch1 ?state' clause = Some ?w1
    using ⟨getWatch2 state clause = Some wb⟩
    unfolding swapWatches-def
    by auto
  let ?w2 = wa
  have getWatch2 ?state' clause = Some ?w2
    using ⟨getWatch1 state clause = Some wa⟩
    unfolding swapWatches-def
    by auto
  show ?thesis
  proof (cases literalTrue ?w1 (elements (getM ?state')))
    case True
    from Cons(2)
    have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
      (getWatch2 ?state')
      unfolding InvariantWatchesEl-def
unfolding \texttt{InvariantWatchesDiffer-def} by \texttt{auto}
moreover from \texttt{Cons(3)} have \texttt{InvariantWatchesDiffer (getF ?state') (getWatch1 ?state') (getWatch2 ?state')}
unfolding \texttt{InvariantWatchesDiffer-def} unfolding \texttt{swapWatches-def} by \texttt{auto}
moreover from \texttt{Cons(4)} have \texttt{InvariantWatchListsUniq (getWatchList ?state')}
unfolding \texttt{InvariantWatchListsUniq-def} unfolding \texttt{swapWatches-def} by \texttt{auto}
moreover have (\texttt{getF ?state'}) = (\texttt{getF state}) \textbf{and} (\texttt{getWatchList ?state'}) = (\texttt{getWatchList state})
unfolding \texttt{swapWatches-def} by \texttt{auto}
moreover have \( \forall c. l \neq \textbf{literal} \rightarrow (c \in \texttt{set (getWatchList ?state' l)}) = (\texttt{Some l = getWatch1 ?state' c \lor Some l = getWatch2 ?state'}) \)
using \texttt{Cons(6)}
using (\texttt{getWatchList ?state'}) = (\texttt{getWatchList state})
using \texttt{swapWatchesEffect} by \texttt{auto}
moreover have \( \forall c. (c \in \texttt{set (clause # newWl)} \lor c \in \texttt{set Wl'}) = (\texttt{Some literal = getWatch1 ?state' c \lor Some literal = getWatch2 ?state'}) \)
using \texttt{Cons(7)}
using \texttt{swapWatchesEffect} by \texttt{auto}
ultimately show ?thesis
using \texttt{Cons(1))[of ?state' clause # newWl]
using \texttt{Cons(5)}
using (\texttt{Some literal = getWatch1 state clause})
using (\texttt{getWatch1 ?state' clause = Some ?w1})
using (\texttt{getWatch2 ?state' clause = Some ?w2})
using (\texttt{literalTrue ?w1 (elements (getM ?state'))})
using (\texttt{uniq Wl'})
using (\texttt{uniq (clause # newWl)})
using (\texttt{set Wl' \cap set (clause # newWl) = {}})
by (\texttt{simp add: Let-def})
next

369
case False
show ？thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state’) clause) ？w1 ？w2 (getM ?state’))
case (Some l’)
hence l’ el (nth (getF ?state’) clause) l’ ≠ literal l’ ≠ ？w1 l’ ≠ ？w2
  using getNonWatchedUnfalsifiedLiteralSomeCharacterization
  using (getWatch1 ?state’ clause = Some ？w1)
  using (getWatch2 ?state’ clause = Some ？w2)
  using (Some literal = getWatch1 state clause)
  unfolding swapWatches-def
  by auto
let ?state’’ = setWatch2 clause l’ ?state’
from Cons(2)
  have InvariantWatchesEl (getF ?state’’) (getWatch1 ?state’’)
    (getWatch2 ?state’’)
      using l’ el (nth (getF ?state’) clause)
      unfolding InvariantWatchesEl-def
      unfolding swapWatches-def
      unfolding setWatch2-def
      by auto
  moreover
  from Cons(3)
  have InvariantWatchesDiffer (getF ?state’’) (getWatch1 ?state’’)
    (getWatch2 ?state’’)
      using (getWatch1 ?state’ clause = Some ？w1)
      using (l’ ≠ ？w1)
      unfolding InvariantWatchesDiffer-def
      unfolding swapWatches-def
      unfolding setWatch2-def
      by simp
  moreover
  have clause ？ set (getWatchList state l’)
    using (l’ ≠ literal)
    using (l’ ≠ ？w1) (l’ ≠ ？w2)
    using (getWatch1 ?state’ clause = Some ？w1)
    using (getWatch2 ?state’ clause = Some ？w2)
    using Cons(6)
    unfolding swapWatches-def
    by simp
  with Cons(4)
  have InvariantWatchListsUniq (getWatchList ?state’’)
    unfolding InvariantWatchListsUniq-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    using uniqAppendIff

370
by force
moreover
have (getF ?state") = (getF state) and
(getWatchList ?state") = (getWatchList state)(l’ := clause #
(getWatchList state l’))
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
have ∀ c l. l ≠ literal →
(c ∈ set (getWatchList ?state" l)) =
(Some l = getWatch1 ?state" c ∨ Some l = getWatch2 ?state"
c)
proof
{ fix c::nat and l::Literal
assume l ≠ literal
have (c ∈ set (getWatchList ?state" l)) = (Some l =
getWatch1 ?state" c ∨ Some l = getWatch2 ?state"
c)
proof (cases c = clause)
 case True
 show ?thesis
 case True
 thus ?thesis
 using ⟨c = clause⟩
 unfolding setWatch2-def
 by simp
 next
 case False
 show ?thesis
 using Cons(6)
 using (getWatchList ?state") = (getWatchList state)(l’ := clause # (getWatchList state l’)):
 using (l ≠ l’)
 using (l ≠ literal):
 using (getWatch1 ?state’ clause = Some ?w1)
 using (getWatch2 ?state’ clause = Some ?w2)
 using (Some literal = getWatch1 state clause)
 using ⟨c = clause⟩
 using swapWatchesEffect
 unfolding swapWatches-def
 unfolding setWatch2-def
 by simp
 qed
 next
 case False
 thus ?thesis
 using Cons(6)
using (l ≠ literal)
using ((getWatchList ?state") = (getWatchList state)(l'
:= clause # (getWatchList state l'))):
  using (c ≠ clause)
  unfolding setWatch2-def
  using swapWatchesEffect[of clause state c]
  by auto
qede

thus ?thesis
  by simp
 qed

moreover
have ∀ c. (c ∈ set newWl ∨ c ∈ set Wl'
= (Some literal = getWatch1 ?state" c ∨ Some literal = getWatch2
?state" c)
  proof −
  show ?thesis
    proof
      fix c :: nat
      show (c ∈ set newWl ∨ c ∈ set Wl') =
        (Some literal = getWatch1 ?state" c ∨ Some literal =
getWatch2 ?state" c)
      proof
        assume c ∈ set newWl ∨ c ∈ set Wl'
        show Some literal = getWatch1 ?state" c ∨ Some literal
        = getWatch2 ?state" c
          proof −
            from (c ∈ set newWl ∨ c ∈ set Wl')
            have Some literal = getWatch1 state c ∨ Some literal =
getWatch2 state c
              using Cons(7)
              by auto
            from Cons(8) ⟨clause ∉ set Wl' | c ∈ set newWl ∨ c ∈
set Wl'⟩
              have c ≠ clause
              by auto
            show ?thesis
              using (Some literal = getWatch1 state c ∨ Some literal
              = getWatch2 state c).
              using (c ≠ clause)
              using swapWatchesEffect
              unfolding setWatch2-def
              by simp
qede
next
assume Some literal = getWatch1 ?state" c ∨ Some literal
= getWatch2 ?state" c
  show c ∈ set newWl ∨ c ∈ set Wl'
  proof
    have Some literal ≠ getWatch1 ?state" clause ∧ Some literal ≠ getWatch2 ?state" clause
      using (l' ≠ literal)
      using (clause < length (getF state));
      using (InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state));
      using (getWatch1 ?state' clause = Some ?w1);
      using (getWatch2 ?state' clause = Some ?w2);
      unfolding InvariantWatchesDiffer-def
      unfolding setWatch2-def
      unfolding swapWatches-def
      by auto
      thus ?thesis
    using (Some literal = getWatch1 ?state" c ∨ Some literal = getWatch2 ?state" c)
    using Cons(7)
    using swapWatchesEffect
    unfolding setWatch2-def
    by (auto split: split-if-asm)
    qed
    qed
    qed
  moreover
  have ∀ c. (c ∈ set (clause # newWl) ∨ c ∈ set Wl') =
    (Some literal = getWatch1 ?state' c ∨ Some literal = getWatch2 ?state' c)
  using Cons(7)
  using swapWatchesEffect
  by auto
  ultimately
  show ?thesis
  using Cons(1)[of ?state" newWl]
  using Cons(5)
  using (uniq Wl')
  using (uniq newWl)
  using (set Wl' ∩ set (clause # newWl) = {});
  using (getWatch1 ?state' clause = Some ?w1);
  using (getWatch2 ?state' clause = Some ?w2);
  using (Some literal = getWatch1 state clause)
  using (∼ literalTrue ?w1 (elements (getM ?state')))
  using Some
  by (simp add: Let-def fun-upd-def)
next
  case None
show ?thesis

proof (cases literalFalse ?w1 (elements (getM ?state')))
  case True
    let ?state'' = ?state' \{getConflictFlag := True, getConflict-Clause := clause\}

  from Cons(2)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    by auto
  moreover
  from Cons(3)
  have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    unfolding InvariantWatchesDiffer-def
    unfolding swapWatches-def
    by auto
  moreover
  from Cons(4)
  have InvariantWatchListsUniq (getWatchList ?state'')
    unfolding InvariantWatchListsUniq-def
    unfolding swapWatches-def
    by auto
  moreover
  have (getF state) = (getF ?state'') and (getWatchList state) = (getWatchList ?state'')
    unfolding swapWatches-def
    by auto
  moreover
  have \(\forall c. l. l \neq \text{literal} \rightarrow (c \in \text{set} \ (\text{getWatchList} ?state'') l)) =
    (\text{Some} l = \text{getWatch1} ?state'' c \lor \text{Some} l = \text{getWatch2} ?state'')
    using Cons(6)
    using \(\text{getWatchList} \text{state} = (\text{getWatchList} ?state'')\)
    using swapWatchesEffect
    by auto
  moreover
  have \(\forall c. (c \in \text{set} \ (\text{clause} \# \text{newWl}) \lor c \in \text{set} \ \text{Wl}')) =
    (\text{Some} \text{literal} = \text{getWatch1} ?state'' c \lor \text{Some} \text{literal} = \text{getWatch2} ?state'')
    using Cons(7)
    using swapWatchesEffect
    by auto
  ultimately
  show ?thesis
    using Cons(1)[of ?state'' clause # newWl]
using \textit{Cons}(5)
using \textit{(getWatch1 \(\textit{?state}'\) clause = \textit{Some} \(w1\)}
using \textit{(getWatch2 \(\textit{?state}'\) clause = \textit{Some} \(w2\)}
using \textit{(Some literal = getWatch1 \textit{state clause)}
using \textit{\neg literalTrue \(w1\) (elements (getM \(\textit{?state}'\))}.
using \textit{None}
using \textit{(literalFalse \(w1\) (elements (getM \(\textit{?state}'\))}
using \textit{\textit{uniq W1'}}
using \textit{\textit{uniq} (\textit{clause} \# \textit{newWl))}
using \textit{(set \textit{Wl}' \cap set (\textit{clause} \# \textit{newWl}) = {}}}.
by \textit{(simp add: Let-def)}

next

case \textit{False}
let \textit{?state''} = \textit{setReason ?w1 clause (\textit{?state'}(getQ := (if \(w1\) el (getQ \(\textit{?state}') then (getQ \(\textit{?state}') else (getQ \(\textit{?state}') @ [\textit{w1}])))}}

from \textit{Cons}(2)
have \textit{InvariantWatchesEl} (\textit{getF \textit{?state''}}) (\textit{getWatch1 \textit{?state''}})
(\textit{getWatch2 \textit{?state''}})
unfolding \textit{InvariantWatchesEl-def}
unfolding \textit{swapWatches-def}
unfolding setReason-def
by \textit{auto}
moreover
from \textit{Cons}(3)
have \textit{InvariantWatchesDiffer} (\textit{getF \textit{?state''}}) (\textit{getWatch1 \textit{?state''}})
(\textit{getWatch2 \textit{?state''}})
unfolding \textit{InvariantWatchesDiffer-def}
unfolding \textit{swapWatches-def}
unfolding setReason-def
by \textit{auto}
moreover
from \textit{Cons}(4)
have \textit{InvariantWatchListsUniq} (\textit{getWatchList \textit{?state''}})
unfolding \textit{InvariantWatchListsUniq-def}
unfolding \textit{swapWatches-def}
unfolding setReason-def
by \textit{auto}
moreover
have (\textit{getF \textit{state}}) = (\textit{getF \textit{?state''}}) \textbf{and} (\textit{getWatchList \textit{state}})
= (\textit{getWatchList \textit{?state''}})
unfolding \textit{swapWatches-def}
unfolding setReason-def
by \textit{auto}
moreover
have \(\forall \textit{c l}. \textit{l} \neq \textit{literal} \rightarrow\)
\((\textit{c} \in \textit{set (getWatchList \textit{?state''} \textit{l})}) =\)
\((\textit{Some} \textit{l} = \textit{getWatch1 \textit{?state''} c} \lor \textit{Some} \textit{l} = \textit{getWatch2 \textit{?state''} c})\)
using Cons(6)
using ⟨getWatchList state) = (getWatchList ?state’⟩)
using swapWatchesEffect
unfolding setReason-def
by auto
moreover
have ∀ c. (c ∈ set (clause ≠ newWl) ∨ c ∈ set Wl) =
(Some literal = getWatch1 ?state’’ c ∨ Some literal =
getWatch2 ?state’’ c)
using Cons(7)
using swapWatchesEffect
unfolding setReason-def
by auto
ultimately
show ?thesis
using Cons(1)[of ?state’’ clause ≠ newWl]
using Cons(5)
using ⟨getWatch1 ?state’ clause = Some ?w1⟩
using ⟨getWatch2 ?state’ clause = Some ?w2⟩
using ⟨Some literal = getWatch1 state clause⟩
using ⟨¬ literalTrue ?w1 (elements (getM ?state’))⟩
using None
using ⟨¬ literalFalse ?w1 (elements (getM ?state’))⟩
using ⟨uniq Wl’⟩
using ⟨uniq (clause ≠ newWl)⟩
using ⟨set Wl’’ ∩ set (clause ≠ newWl) = {}⟩
by (simp add: Let-def)
qed
qed
qed

next
case False
let ?state’ = state
let ?w1 = wa
have ⟨getWatch1 ?state’ clause = Some ?w1⟩
using ⟨getWatch1 state clause = Some wa⟩
unfolding swapWatches-def
by auto
let ?w2 = wb
have ⟨getWatch2 ?state’ clause = Some ?w2⟩
using ⟨getWatch2 state clause = Some wb⟩
unfolding swapWatches-def
by auto

have Some literal = getWatch2 state clause
using ⟨getWatch1 ?state’ clause = Some ?w1⟩
using ⟨getWatch2 ?state’ clause = Some ?w2⟩
using ⟨Some literal ≠ getWatch1 state clause⟩
using Cons(7)
by force

show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state')))  
case True
  from Cons(7) have
  \( \forall c. (c \in \{\text{clause \# newWl}\} \lor c \in Wl') = \)  
  (Some literal = getWatch1 state c \lor Some literal = getWatch2 state c)
  by auto
  thus ?thesis
    using Cons(1)[of ?state' clause \# newWl]
    using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6)
    using (\sim Some literal = getWatch1 state clause)
    using (getWatch1 ?state' clause = Some ?w1)
    using (getWatch2 ?state' clause = Some ?w2)
    using (literalTrue ?w1 (elements (getM ?state')))
    using (uniq (clause \# newWl))
    using (uniq Wl')
    using (set Wl' \cap set (clause \# newWl) = \{})
    by simp
next
  case False
  show ?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
      case (Some l')
      hence l' el (nth (getF ?state') clause) l' \neq literal l' \neq ?w1 l'
        \neq ?w2
        using getNonWatchedUnfalsifiedLiteralSomeCharacterization
        using (Some literal = getWatch2 state clause)
        using (getWatch1 ?state' clause = Some ?w1)
        using (getWatch2 ?state' clause = Some ?w2)
        by auto
      let ?state'' = setWatch2 clause l' ?state'

    from Cons(2)
      have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
        using (l' el (nth (getF ?state') clause))
        unfolding InvariantWatchesEl-def
        unfolding setWatch2-def
        by auto
    moreover
    from Cons(3)
      have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
        using (getWatch1 ?state' clause = Some ?w1)
using \( l' \neq \texttt{w1} \)

unfolding \( \textit{InvariantWatchesDiffer-def} \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{simp} \)

moreover

have \( \text{clause} \notin \text{set} (\textit{getWatchList state} \ l') \)

using \( l' \neq \texttt{w1} \)

using \( l' \neq \texttt{w2} \)

using \( \textit{getWatch1 \state'} \text{ clause} = \texttt{Some \texttt{w1}} \)

using \( \textit{getWatch2 \state'} \text{ clause} = \texttt{Some \texttt{w2}} \)

using \( \texttt{Cons(6)} \)

by \( \texttt{simp} \)

with \( \texttt{Cons(4)} \)

have \( \textit{InvariantWatchListsUniq} (\textit{getWatchList ?state'}) \)

unfolding \( \textit{InvariantWatchListsUniq-def} \)

unfolding \( \textit{setWatch2-def} \)

using \( \texttt{uniqAppendIff} \)

by \( \texttt{force} \)

moreover

have \( (\textit{getF ?state''}) = (\textit{getF state}) \text{ and} \)

\( (\textit{getWatchList ?state''}) = (\textit{getWatchList state})(l' := \text{ clause } \#) \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{auto} \)

moreover

have \( \forall c \ l. \ l \neq \texttt{literal} \rightarrow \)

\( (c \in \text{set} (\textit{getWatchList ?state''} \ l)) = \)

\( (\texttt{Some} \ l = \textit{getWatch1 ?state''} c \lor \texttt{Some} \ l = \textit{getWatch2 ?state''} c) \)

proof

\{ fi \}

fix \( c::\texttt{nat} \text{ and } l::\texttt{Literal} \)

assume \( l \neq \texttt{literal} \)

have \( (c \in \text{set} (\textit{getWatchList ?state''} \ l)) = (\texttt{Some} \ l = \textit{getWatch1 ?state''} c \lor \texttt{Some} \ l = \textit{getWatch2 ?state''} c) \)

proof (cases \( c = \texttt{clause} \))

case \( \texttt{True} \)

show \( \texttt{thesis} \)

proof (cases \( l = l' \))

case \( \texttt{True} \)

thus \( \texttt{thesis} \)

using \( (c = \texttt{clause}) \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{simp} \)

next

case \( \texttt{False} \)

show \( \texttt{thesis} \)

using \( \texttt{Cons(6)} \)

using \( (\textit{getWatchList ?state''}) = (\textit{getWatchList state})(l' \neq 378\texttt{w1}) \)

unfolding \( \textit{InvariantWatchListsUniq-def} \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{auto} \)

moreover

have \( \forall c \ l. \ l \neq \texttt{literal} \rightarrow \)

\( (c \in \text{set} (\textit{getWatchList ?state''} \ l)) = \)

\( (\texttt{Some} \ l = \textit{getWatch1 ?state''} c \lor \texttt{Some} \ l = \textit{getWatch2 ?state''} c) \)

proof (cases \( c = \texttt{clause} \))

case \( \texttt{True} \)

show \( \texttt{thesis} \)

proof (cases \( l = l' \))

case \( \texttt{True} \)

thus \( \texttt{thesis} \)

using \( (c = \texttt{clause}) \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{simp} \)

next

case \( \texttt{False} \)

show \( \texttt{thesis} \)

using \( \texttt{Cons(6)} \)

using \( (\textit{getWatchList ?state''}) = (\textit{getWatchList state})(l' \neq 378\texttt{w1}) \)

unfolding \( \textit{InvariantWatchListsUniq-def} \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{auto} \)

moreover

have \( \forall c \ l. \ l \neq \texttt{literal} \rightarrow \)

\( (c \in \text{set} (\textit{getWatchList ?state''} \ l)) = \)

\( (\texttt{Some} \ l = \textit{getWatch1 ?state''} c \lor \texttt{Some} \ l = \textit{getWatch2 ?state''} c) \)

proof (cases \( c = \texttt{clause} \))

case \( \texttt{True} \)

show \( \texttt{thesis} \)

proof (cases \( l = l' \))

case \( \texttt{True} \)

thus \( \texttt{thesis} \)

using \( (c = \texttt{clause}) \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{simp} \)

next

case \( \texttt{False} \)

show \( \texttt{thesis} \)

using \( \texttt{Cons(6)} \)

using \( (\textit{getWatchList ?state''}) = (\textit{getWatchList state})(l' \neq 378\texttt{w1}) \)

unfolding \( \textit{InvariantWatchListsUniq-def} \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{auto} \)

moreover

have \( \forall c \ l. \ l \neq \texttt{literal} \rightarrow \)

\( (c \in \text{set} (\textit{getWatchList ?state''} \ l)) = \)

\( (\texttt{Some} \ l = \textit{getWatch1 ?state''} c \lor \texttt{Some} \ l = \textit{getWatch2 ?state''} c) \)

proof (cases \( c = \texttt{clause} \))

case \( \texttt{True} \)

show \( \texttt{thesis} \)

proof (cases \( l = l' \))

case \( \texttt{True} \)

thus \( \texttt{thesis} \)

using \( (c = \texttt{clause}) \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{simp} \)

next

case \( \texttt{False} \)

show \( \texttt{thesis} \)

using \( \texttt{Cons(6)} \)

using \( (\textit{getWatchList ?state''}) = (\textit{getWatchList state})(l' \neq 378\texttt{w1}) \)

unfolding \( \textit{InvariantWatchListsUniq-def} \)

unfolding \( \textit{setWatch2-def} \)

by \( \texttt{auto} \)
\[\begin{align*}
\text{clause} \# (\text{getWatchList state l'}) \\
&\text{using } (l \neq l') \\
&\text{using } (l \neq \text{literal}) \\
&\text{using } (\text{getWatch1 ?state' clause} = \text{Some ?w1}) \\
&\text{using } (\text{getWatch2 ?state' clause} = \text{Some ?w2}) \\
&\text{using } (c = \text{clause}) \\
\end{align*}\]

unfolding setWatch2-def
by simp

qed

next

\begin{align*}
\text{case False} \\
\text{thus } \text{thesis} \\
&\text{using } \text{Cons(6)} \\
&\text{using } (l \neq \text{literal}) \\
&\text{using } (\text{getWatchList ?state''} = (\text{getWatchList state})(l'') \\
&\text{using } (c \neq \text{clause}) \\
\end{align*}

unfolding setWatch2-def
by auto

qed

moreover

have \( \forall c. \ (c \in \text{set newWl} \lor c \in \text{set Wl}') = \) \\
\( (\text{Some literal} = \text{getWatch1 ?state''} c \lor \text{Some literal} = \text{getWatch2 ?state''} c) \) \\
proof
- show \( \text{thesis} \)
  proof
  fix \( c :: \text{nat} \)
  show \( (c \in \text{set newWl} \lor c \in \text{set Wl}') = \) \\
  \( (\text{Some literal} = \text{getWatch1 ?state''} c \lor \text{Some literal} = \text{getWatch2 ?state''} c) \) \\
  proof
    assume \( c \in \text{set newWl} \lor c \in \text{set Wl}' \)
    show \( \text{Some literal} = \text{getWatch1 ?state''} c \lor \text{Some literal} = \text{getWatch2 ?state''} c \) \\
    proof
      from \( (c \in \text{set newWl} \lor c \in \text{set Wl'}) \)
      have \( \text{Some literal} = \text{getWatch1 state} c \lor \text{Some literal} = \text{getWatch2 state} c \)
      using \( \text{Cons(7)} \)
      by auto
    from \( \text{Cons(8)} \) \( \text{clause} \notin \text{set Wl'} \) \( c \in \text{set newWl} \lor c \in \text{set Wl'} \)
have $c \neq \text{clause}$
by auto

show $\exists \text{thesis}$
using $(\exists \text{literal} = \text{getWatch1 state } c \lor \exists \text{literal} = \text{getWatch2 state } c)$
using $(c \neq \text{clause})$
unfolding setWatch2-def
by simp
qed

next
assume $\exists \text{literal} = \text{getWatch1 state'' } c \lor \exists \text{literal} = \text{getWatch2 state'' } c$

show $c \in \text{set newWl} \lor c \in \text{set Wl'}$

proof

have $\exists \text{literal} \neq \text{getWatch1 state'' clause} \land \exists \text{literal} \neq \text{getWatch2 state'' clause}$
using $(l' \neq \text{literal})$
using $(\text{clause < length } (\text{getF state}))$
using $(\text{InvariantWatchesDiffer } (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state}))$
using $(\text{getWatch1 state'} \text{clause} = \text{Some w1})$
using $(\text{getWatch2 state'} \text{clause} = \text{Some w2})$
using $(\exists \text{literal} = \text{getWatch2 state clause})$
unfolding InvariantWatchesDiffer-def
unfolding setWatch2-def
by auto
thus $\exists \text{thesis}$
using $(\exists \text{literal} = \text{getWatch1 state'' } c \lor \exists \text{literal} = \text{getWatch2 state'' } c)$
using Cons(7)
unfolding setWatch2-def
by (auto split: split-if-asm)
qed
qed
qed

moreover
have $\forall c \cdot (c \in \text{set } (\text{clause }\neq \text{newWl}) \lor c \in \text{set Wl'}) =$ $(\exists \text{literal} = \text{getWatch1 state'} c \lor \exists \text{literal} = \text{getWatch2 state'} c)$
using Cons(7)
by auto
ultimately
show $\exists \text{thesis}$
using Cons(1)[of $\exists \text{state'' } \text{newWl}$]
using Cons(5)
using (uniq Wl')
using (uniq newWl)

380
using \(\{\text{set } Wl' \cap \text{set } (\text{clause } \# \text{ newWl}) = \{\}\}\)
using \(\text{getWatch1 } \text{?state'} \text{ clause } = \text{Some } \text{?w1}\)
using \(\text{getWatch2 } \text{?state'} \text{ clause } = \text{Some } \text{?w2}\)
using \((\Rightarrow \text{Some literal } = \text{getWatch1 state clause})\)
using \((\Rightarrow \text{literalTrue } \text{?w1} (\text{elements } (\text{getM } \text{?state'})))\)
using \(\text{Some}\)
by \((\text{simp add: Let-def fun-upd-def})\)

next

case None

show \(\text{thesis}\)

proof \((\text{cases literalFalse } \text{?w1} (\text{elements } (\text{getM } \text{?state'})))\)

case True

let \(\text{?state''} = \text{?state'}[\text{getConflictFlag ::= } \text{True}, \text{getConflict-Clause ::= clause}]\)

from Cons(2)

have \(\text{InvariantWatchesEl } (\text{getF } \text{?state''}) (\text{getWatch1 } \text{?state''}) (\text{getWatch2 } \text{?state''})\)

unfolding \(\text{InvariantWatchesEl-def}\)

by auto

moreover

from Cons(3)

have \(\text{InvariantWatchesDiffer } (\text{getF } \text{?state''}) (\text{getWatch1 } \text{?state''}) (\text{getWatch2 } \text{?state''})\)

unfolding \(\text{InvariantWatchesDiffer-def}\)

by auto

moreover

from Cons(4)

have \(\text{InvariantWatchListsUniq } (\text{getWatchList } \text{?state''})\)

unfolding \(\text{InvariantWatchListsUniq-def}\)

by auto

moreover

have \((\text{getF state}) = (\text{getF } \text{?state''})\)

by auto

moreover

have \(\forall \text{c l. } l \neq \text{literal } \rightarrow \)
\(\{\text{c } \in \text{ set } (\text{getWatchList } \text{?state'' l})\} = \)
\((\text{Some } l = \text{getWatch1 } \text{?state'' c } \lor \text{Some } l = \text{getWatch2 } \text{?state'' c})\)

using Cons(6)

by simp

moreover

have \(\forall \text{c. } (\text{c } \in \text{ set } (\text{clause } \# \text{ newWl}) \lor \text{c } \in \text{ set } \text{Wl'}) = \)
\((\text{Some literal } = \text{getWatch1 } \text{?state'' c } \lor \text{Some literal } = \text{getWatch2 } \text{?state'' c})\)

using Cons(7)

by auto

ultimately

have \(\text{let state'} = \text{notifyWatches-loop literal Wl'} \text{ (clause } \#\)
newWl) ?state'' in

InvariantWatchListsCharacterization (getWatchList state') (getWatch1 state') (getWatch2 state') ∧

InvariantWatchListsUniq (getWatchList state')

using Cons(1)[of ?state'' clause # newWl]
using Cons(5)
using uniq WI'
using uniq (clause # newWl);
using (set WI' ∩ set (clause # newWl) = {});
apply (simp only: Let-def)
by (simp (no-asm-use)) (simp)

thus ?thesis

using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
using (Some literal ≠ getWatch1 state clause)
using None
using (literalFalse ?w1 (elements (getM ?state')))
by (simp add: Let-def)

next

case False

let ?state'' = setReason ?w1 clause (?state' (getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])))

from Cons(2)

have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')

(getWatch2 ?state'')

unfolding InvariantWatchesEl-def
unfolding setReason-def
by auto

moreover
from Cons(3)

have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')

unfolding InvariantWatchesDiffer-def
unfolding setReason-def
by auto

moreover
from Cons(4)

have InvariantWatchListsUniq (getWatchList ?state'')

unfolding InvariantWatchListsUniq-def
unfolding setReason-def
by auto

moreover

have (getF state) = (getF ?state'')

unfolding setReason-def
by auto

moreover

382
have \( \forall c.l, l \neq \text{literal} \rightarrow \)
\( (c \in \text{set}\ (\text{getWatchList}\ ?\text{state''}\ l)) = \)
\( (\text{Some } l = \text{getWatch1}\ ?\text{state''}\ c \vee \text{Some } l = \text{getWatch2}\ ?\text{state''}\ c) \)
using Cons(6)
unfolding setReason-def
by auto

moreover
have \( \forall c. (c \in \text{set}\ (\text{clause}\ #\ newWl) \vee c \in \text{set}\ Wl) = \)
\( (\text{Some}\ \text{literal} = \text{getWatch1}\ ?\text{state''}\ c \vee \text{Some}\ \text{literal} = \text{getWatch2}\ ?\text{state''}\ c) \)
using Cons(7)
unfolding setReason-def
by auto

ultimately
show \( ?\text{thesis} \)
using Cons(1)
of \( ?\text{state''}\ \text{clause} \neq \text{newWl} \)
using Cons(5)
using (getWatch1 ?state clause = Some ?w1)
using (getWatch2 ?state clause = Some ?w2)
using (\( \neg \) Some literal = getWatch1 state clause)
using (\( \neg \) literalTrue ?w1 (elements (getM ?state')))
using None
using (\( \neg \) literalFalse ?w1 (elements (getM ?state')))
using (uniq Wl)
using (uniq (clause \# newWl))
using (set Wl \( \cap \) set (clause \# newWl) = \{};
by (simp add: Let-def)
qed

lemma NotifyWatchesLoopWatchCharacterizationEffect:
fixes literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantConsistent (getM state) and
InvariantUniq (getM state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) M
\( \forall (c::\text{nat}). \ c \in \text{set}\ Wl \rightarrow \emptyset \leq c \wedge c < \text{length}\ (\text{getF}\ \text{state}) \) and
getM state = M \( @ \ [(\text{opposite}\ \text{literal}, \text{decision})] \)
uniq Wl
∀ (c::nat). c ∈ set Wl → Some literal = (getWatch1 state c) ∨ Some literal = (getWatch2 state c)

shows
let state' = notifyWatches-loop literal Wl newWl state in
∀ (c::nat). c ∈ set Wl → (∀ w1 w2.(Some w1 = (getWatch1 state' c) ∧ Some w2 = (getWatch2 state' c)) →
  (watchCharacterizationCondition w1 w2 (getM state') (nth (getF state') c) ∧
  watchCharacterizationCondition w2 w1 (getM state') (nth (getF state') c)))
)

using assms
proof (induct Wl arbitrary: newWl state)
case Nil
  thus ?case
    by simp
next
case (Cons clause Wl')
  from (∀ (c::nat). c ∈ set (clause ≠ Wl') → 0 ≤ c ∧ c < length (getF state)):
    have 0 ≤ clause ∧ clause < length (getF state)
      by auto
  then obtain wa::Literal and wb::Literal
     where getWatch1 state clause = Some wa and getWatch2 state clause = Some wb
       using Cons
       unfolding InvariantWatchesEl-def
       by auto
      have uniq Wl' clause ∉ set Wl'
        using Cons(9)
        by (auto simp add: uniqAppendIff)
     show ?case
   proof (cases Some literal = getWatch1 state clause)
case True
  let ?state' = swapWatches clause state
  let ?w1 = wb
  have getWatch1 ?state' clause = Some ?w1
    using (getWatch2 state clause = Some wb)
    unfolding swapWatches-def
    by auto
  let ?w2 = wa
  have getWatch2 ?state' clause = Some ?w2
    using (getWatch1 state clause = Some wa)
    unfolding swapWatches-def
    by auto
  with True have
    ?w2 = literal
    unfolding swapWatches-def

384
by simp

from (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state))
have \(?w_1\) el (nth (getF state) clause) \(?w_2\) el (nth (getF state) clause)
  using (getWatch1 ?state' clause = Some \(?w_1\))
  using (getWatch2 ?state' clause = Some \(?w_2\))
  using (0 ≤ clause ∧ clause < length (getF state))
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto

from (InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state))
have \(?w_1\) ≠ \(?w_2\)
  using (getWatch1 ?state' clause = Some \(?w_1\))
  using (getWatch2 ?state' clause = Some \(?w_2\))
  using (0 ≤ clause ∧ clause < length (getF state))
  unfolding InvariantWatchesDiffer-def
  unfolding swapWatches-def
  by auto

have ¬ literalFalse \(?w_2\) (elements M)
  using (\(?w_2\) = literal)
  using Cons(5)
  using Cons(8)
  unfolding InvariantUniq-def
  by (simp add: uniqAppendIff)

show \(?\text{thesis}\)
proof (cases literalTrue \(?w_1\) (elements (getM ?state')))
  case True

    let \(?f\text{State}\) = notifyWatches-loop literal Wl' (clause # newWl) ?state'

from Cons(2)
  have invariantWatchesEl (getF ?state') (getWatch1 ?state')
    (getWatch2 ?state')
    unfolding invariantWatchesEl-def
    unfolding swapWatches-def
    by auto
moreover
from Cons(3)
  have invariantWatchesDiffer (getF ?state') (getWatch1 ?state')
    (getWatch2 ?state')
    unfolding invariantWatchesDiffer-def
    unfolding swapWatches-def

385
by auto
moreover
from Cons(4)
have InvariantConsistent (getM ?state')
  unfolding InvariantConsistent-def
  unfolding swapWatches-def
  by simp
moreover
from Cons(5)
have InvariantUniq (getM ?state')
  unfolding InvariantUniq-def
  unfolding swapWatches-def
  by simp
moreover
from Cons(6)
have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') M
  unfolding swapWatches-def
  unfolding InvariantWatchCharacterization-def
  unfolding watchCharacterizationCondition-def
  by simp
moreover
have getM ?state' = getM state
  getF ?state' = getF state
  unfolding swapWatches-def
  by auto
moreover
have (∀ (c::nat). c ∈ set Wl' → Some literal = (getWatch1 ?state' c) ∨ Some literal = (getWatch2 ?state' c))
  using Cons(10)
  unfolding swapWatches-def
  by auto
moreover
have getWatch1 ?fState clause = getWatch1 ?state' clause ∧
  getWatch2 ?fState clause = getWatch2 ?state' clause
  using (clause ∈ set Wl')
  using invariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getF ?state' = getF state)
  using Cons(7)
  using notifyWatchesLoopPreservedWatches[of ?state' Wl' literal clause # newWl]
  by (simp add: Let-def)
moreover
have watchCharacterizationCondition ?w1 ?w2 (getM ?fState)
  (getF ?fState ! clause) ∧
  watchCharacterizationCondition ?w2 ?w1 (getM ?fState)
  (getF ?fState ! clause)
proof−
  have (getM ?fState) = (getM state) ∧ (getF ?fState = getF(386)
state)
    using notifyWatchesLoopPreservedVariables[of ?state' Wl']
literal clause # newWl
    using InvariantWatchesEl (getF ?state') (getWatch1 ?state')
    (getWatch2 ?state') (getF ?state' = getF state)
    using Cons(7)
    unfolding swapWatches-def
    by (simp add: Let-def)
moreover
have ¬ literalFalse ?w1 (elements M)
    using (literalTrue ?w1 (elements (getM ?state')) (w1 ≠ w2)
    (w2 = literal)
    unfolding Cons(4) Cons(8)
    unfolding InvariantConsistent-def
    unfolding swapWatches-def
    by (auto simp add: inconsistentCharacterization)
moreover
have elementLevel (opposite ?w2 (getM ?state')) (currentLevel
    (getM ?state'))
    using (w2 = literal)
    unfolding Cons(5) Cons(8)
    unfolding InvariantUniq-def
    unfolding swapWatches-def
    by (auto simp add: uniqAppendIff elementOnCurrentLevel)
ultimately
show ?thesis
using getWatch1 ?fState clause = getWatch1 ?state' clause
∧ getWatch2 ?fState clause = getWatch2 ?state' clause:
    using (w2 = literal) (w1 ≠ w2)
    using (literalTrue ?w1 (elements (getM ?state')))
    unfolding watchCharacterizationCondition-def
    unfolding elementLevelLeqCurrentLevel[of ?w1 getM ?state']
    using notifyWatchesLoopPreservedVariables[of ?state' Wl']
literal clause # newWl
    using InvariantWatchesEl (getF ?state') (getWatch1 ?state')
    (getWatch2 ?state') (getF ?state' = getF state)
    using Cons(7)
    using Cons(8)
    unfolding swapWatches-def
    by (auto simp add: Let-def)
qed
ultimately
show ?thesis
using Cons(1)[of ?state' clause # newWl]
using Cons(7) Cons(8)
using uniq Wl'
using (getWatch1 ?state' clause = Some ?w1);
using (getWatch2 ?state' clause = Some ?w2);
using \(\text{Some literal} = \text{getWatch1 state clause}\)
using \(\text{literalTrue} ?w1 \text{ (elements (getM ?state'))}\)
by (simp add: Let-def)

next

case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
case (Some l')
hence \(l' \in\) (nth (getF ?state') clause) \(l' \neq \text{?w1 \ l' \neq \text{?w2} \Rightarrow \text{literalFalse \ l'}}\)
using \(\langle\text{getWatch1 ?state'} \text{ clause} = \text{Some \ ?w1}\rangle\)
using \(\langle\text{getWatch2 ?state'} \text{ clause} = \text{Some \ ?w2}\rangle\)
using \(\langle\text{getNonWatchedUnfalsifiedLiteralSomeCharacterization}\rangle\)
by auto

let \(?state'' = \text{setWatch2 clause l'} \text{ ?state'}\)
let \(?\text{State} = \text{notifyWatches-loop literal Wl' newWl ?state''}\)

from Cons(2)
have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
using \(l' \in\) (nth (getF ?state') clause):
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
from Cons(3)
have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
using \(l' \neq \text{?w1}\)
using \(\langle\text{getWatch1 ?state'} \text{ clause} = \text{Some \ ?w1}\rangle\)
using \(\langle\text{getWatch2 ?state'} \text{ clause} = \text{Some \ ?w2}\rangle\)
unfolding InvariantWatchesDiffer-def
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
from Cons(4)
have InvariantConsistent (getM ?state'')
unfolding InvariantConsistent-def
unfolding setWatch2-def
unfolding swapWatches-def
by simp
moreover
from Cons(5)
have InvariantUniq (getM ?state'')
unfolding InvariantUniq-def
unfolding setWatch2-def
unfolding swapWatches-def
by simp
moreover
have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') M
proof -
  { fix c::nat and ww1::Literal and ww2::Literal
   assume a: 0 ≤ c ∧ c < length (getF ?state'') ∧ Some ww1 = (getWatch1 ?state'' c) ∧ Some ww2 = (getWatch2 ?state'' c)
   assume b: literalFalse ww1 (elements M)
   have (∃ l. l el ((getF ?state'') ! c) ∧ literalTrue l (elements M)) ∧ elementLevel l M ≤ elementLevel (opposite ww1) M ∨
       (∀ l. l el ((getF ?state'') ! c) ∧ l ≠ ww1 ∧ l ≠ ww2 −→
           literalFalse l (elements M) ∧ elementLevel (opposite l) M ≤ elementLevel (opposite ww1) M)
   proof (cases c = clause)
    case False
    thus ?thesis
    using a and b
    using Cons(6)
    unfolding InvariantWatchCharacterization-def
    unfolding watchCharacterizationCondition-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    by simp
  next
  case True
  with a
  have ww1 = ?w1 and ww2 = l'
    using ⟨getWatch1 ?state' clause = Some ?w1⟩
    using ⟨getWatch2 ?state' clause = Some ?w2⟩ [THEN sym]
    unfolding setWatch2-def
    unfolding swapWatches-def
    by auto
  have ¬ (∀ l. l el (getF state ! clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 −→
   literalFalse l (elements M))
    using Cons(8)
    using ⟨l' ≠ ?w1 and l' ≠ ?w2⟩ [THEN clause]
    using ⟨¬ literalFalse l' (elements (getM ?state'))⟩
    using a and b
    using ⟨c = clause⟩
    unfolding swapWatches-def
    unfolding setWatch2-def

389
by auto
moreover
have (\exists l. l \in (getF ?state ! clause) \land literalTrue l (elements M)) \land
  \ (\exists l. elementLevel l M \leq elementLevel (opposite \ ?w1) M) \lor
  \ (\forall l. l \in (getF ?state ! clause) \land l \neq \ ?w1 \land l \neq \ ?w2 \rightarrow
  literalFalse l (elements M)))
  using Cons(6)
  unfolding InvariantWatchCharacterization-def
  unfolding watchCharacterizationCondition-def
  using (0 \leq clause \land clause < length (get\ ?state))
  using (getWatch1 ?state' clause = Some \ ?w1)[THEN
sym]
  using (getWatch2 ?state' clause = Some \ ?w2)[THEN
sym]
  using (literalFalse \ ?w1 (elements M))
  using (\ ?w1 = \ ?w1)
  unfolding setWatch2-def
  unfolding swapWatches-def
  by auto
ultimately
show \ ?thesis
  using (\ ?w1 = \ ?w1)
  using (c = clause)
  unfolding setWatch2-def
  unfolding swapWatches-def
  by auto
qed

moreover
{
  fix c :: nat and \ ww1 :: Literal and \ ww2 :: Literal
  assume a: 0 \leq c \land c < length (get\ ?state") \land Some \ ww1 = (getWatch1 ?state") c \land Some \ ww2 = (getWatch2 ?state" c)
  assume b: literalFalse \ ?w2 (elements M)

  have (\exists l. l \in ((get\ ?state") ! c) \land literalTrue l (elements M) \land elementLevel l M \leq elementLevel (opposite \ ww2) M) \lor
    (\forall l. l \in ((get\ ?state") ! c) \land l \neq \ ?w1 \land l \neq \ ?w2 \rightarrow
    literalFalse l (elements M) \land elementLevel (opposite \ l) M \leq elementLevel (opposite \ ww2) M)
  proof (cases c = clause)
    case False
    thus \ ?thesis
    using a and b
    using Cons(6)
    unfolding InvariantWatchCharacterization-def
    unfolding watchCharacterizationCondition-def
    unfolding swapWatches-def

  qed

390
unfolding setWatch2-def
by auto
next
case True
with a
have ww1 = ?w1 and ww2 = l'
  using ⟨getWatch1 ?state' clause = Some ?w1⟩
  using ⟨getWatch2 ?state' clause = Some ?w2⟩[THEN
sym]
  unfolding setWatch2-def
  unfolding swapWatches-def
  by auto
with (∼ literalFalse l' (elements (getM ?state'))): b
Cons(8)
have False
  unfolding swapWatches-def
  by simp
  thus ?thesis
  by simp
qed
}
ultimately
show ?thesis
  unfolding InvariantWatchCharacterization-def
  unfolding watchCharacterizationCondition-def
  by blast
qed
moreover
have ∀ (c::nat). c ∈ set Wl' → Some literal = (getWatch1 ?state'' c) ∨ Some literal = (getWatch2 ?state'' c)
  using Cons(10)
  using ⟨clause ∉ set Wl'⟩
  unfolding swapWatchesEffect[of clause state]
  unfolding setWatch2-def
  by simp
moreover
have getM ?state'' = getM state
getF ?state'' = getF state
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto
moreover
have getWatch1 ?state'' clause = Some ?w1 getWatch2 ?state'' clause = Some l'
  using ⟨getWatch1 ?state' clause = Some ?w1⟩
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto
  hence getWatch1 ?fState clause = getWatch1 ?state'' clause ∧
getWatch2 ?fState clause = Some l'

using ⟨clause ∈ set Wl'⟩
using ⟨InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state") : getF ?state" = getF state⟩
using Cons(T)
using notifyWatchesLoopPreservedWatches[of ?state" Wl'
literal newWl]
by (simp add: Let-def)

moreover
have watchCharacterizationCondition ?w1 l' (getM ?fState)
(getF ?fState ! clause) ∧
watchCharacterizationCondition l' ?w1 (getM ?fState) (getF ?fState ! clause)
proof –
have (getM ?fState) = (getM state) (getF ?fState) = (getF state)
using notifyWatchesLoopPreservedVariables[of ?state" Wl'
literal newWl]
using ⟨InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state") : getF ?state" = getF state⟩
using Cons(T)
unfolding setWatch2-def
unfolding swapWatches-def
by (auto simp add: Let-def)

have literalFalse ?w1 (elements M) →
(∃ l. l el (nth (getF ?state") clause) ∧ literalTrue l (elements M) ∧
elementLevel l M ≤ elementLevel (opposite ?w1) M)
proof
assume literalFalse ?w1 (elements M)
show ∃ l. l el (nth (getF ?state") clause) ∧ literalTrue l (elements M) ∧
elementLevel l M ≤ elementLevel (opposite ?w1) M
proof –
have ¬ (∀ l. l el (nth (getF state) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements M))
using ⟨∃ l. l el (nth (getF ?state") clause) l' ≠ ?w1. l' ≠ ?w2⟩
using Cons(8)
unfolding swapWatches-def
by auto

from ⟨literalFalse ?w1 (elements M), Cons(6)⟩
have
(∃ l. l el (getF state ! clause) ∧ literalTrue l (elements M) ∧
elementLevel l M ≤ elementLevel (opposite ?w1) M) ∨
(∀ l. l el (getF state ! clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 →
literalFalse l (elements M) ∧
elementLevel (opposite l) M ≤ elementLevel (opposite ?w1) M)
using ⟨0 ≤ clause ∧ clause < length (getF state)⟩
using \langle\text{getWatch1 ?state'} clause = Some ?w1\rangle [THEN

sym]

using \langle\text{getWatch2 ?state'} clause = Some ?w2\rangle [THEN

sym]

unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
unfolding swapWatches-def
by simp
with (\neg (\forall l. l el (nth (getF state) clause) \land l \neq ?w1 \land l \
\neq ?w2 \rightarrow literalFalse l (elements M))):
have \exists l. l el (getF state ! clause) \land literalTrue l (elements M) \land elementLevel l M \leq elementLevel (opposite ?w1) M
by auto
thus \thesis
unfolding setWatch2-def
unfolding swapWatches-def
by simp
qed
qed

have watchCharacterizationCondition l' ?w1 (getM ?fState) (getF ?fState ! clause)
using (\neg literalFalse l' (elements (getM ?state')))
using (getM ?fState = getM state);
unfolding swapWatches-def
unfolding watchCharacterizationCondition-def
by simp
moreover
have watchCharacterizationCondition ?w1 l' (getM ?fState) (getF ?fState ! clause)
proof (cases literalFalse ?w1 (elements (getM ?fState')))
case True
hence literalFalse ?w1 (elements M)
using notifyWatchesLoopPreservedVariables[of ?state'' Wl']
literal newWl'

using (InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')) (getWatch2 ?state'') (getF ?state'' = getF state)
using Cons(7) Cons(8)
using (?w1 \neq ?w2) (?w2 = literal)
unfolding setWatch2-def
unfolding swapWatches-def
by (simp add: Let-def)
with (literalFalse ?w1 (elements M) \rightarrow
(\exists l. l el (nth (getF ?state'') clause) \land literalTrue l (elements M) \land elementLevel l M \leq elementLevel (opposite ?w1) M):
obtain l::Literal
where l el (nth (getF ?state'') clause) and
literalTrue l (elements M) and
elementLevel l M \leq elementLevel (opposite ?w1) M

393
by auto

hence \( \text{elementLevel} \ l \ (\text{getM} \ state) \leq \text{elementLevel} \ (\text{opposite} \ ?w1) \ (\text{getM} \ state) \)

using Cons(8)
using literalTrue l (\( \text{elements} \ M \)) (\( \text{literalFalse} \ ?w1 \ (\text{elements} \ M) \))

using elementLevelAppend[of l M [(\( \text{opposite} \ \text{l literal}, \ \text{decision} \))]]
using elementLevelAppend[of opposite ?w1 M [(\( \text{opposite} \ \text{l literal}, \ \text{decision} \))]]

by auto
thus \(?\text{thesis}\)
using \( l \ \text{el} \ (\text{nth} \ (\text{getF} \ ?\text{state}'') \ \text{clause}) \) (\( \text{literalTrue} \ l \ (\text{elements} \ M) \))

using \( \text{getM} \ ?\text{State} = \text{getM} \ \text{state} \) (\( \text{getF} \ ?\text{State} = \text{getF} \ \text{state} \)) (\( \text{getM} \ ?\text{state}''' = \text{getM} \ \text{state} \) (\( \text{getF} \ ?\text{state}''' = \text{getF} \ \text{state} \))

using Cons(8)

unfolding \( \text{watchCharacterizationCondition-def} \)

by auto
next
case False

thus \(?\text{thesis}\)

unfolding \( \text{watchCharacterizationCondition-def} \)

by simp

qed

ultimately

show \(?\text{thesis}\)

by simp

qed

ultimately

show \(?\text{thesis}\)

using Cons(1)[of \( ?\text{state}'' \ \text{newWl} \)]

using Cons(7) Cons(8)

using \( \text{getWatch1} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w1 \)

using \( \text{getWatch2} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w2 \)

using \( \text{Some literal} = \text{getWatch1} \ \text{state clause} \)

using \( \sim \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}') ) \)

using \( \text{getWatch1} ?\text{state}'' \ \text{clause} = \text{Some} \ ?w1 \)

using \( \text{getWatch2} ?\text{state}'' \ \text{clause} = \text{Some} \ l' \)

using \( \text{Some} \)

using \( \text{uniq} \ \text{Wl} \)

by (simp add: Let-def)
next
case None

show \(?\text{thesis}\)

proof (cases literalFalse \ ?w1 \ (\text{elements} \ (\text{getM} ?\text{state}')))

case True

let \(?\text{state}'' = \text{?state}'\langle\text{getConflictFlag} := \text{True}, \text{getConflict-Clause} := \text{clause}\rangle \)
let ?fState = notifyWatches-loop literal Wl' (clause # newWl) ?state"

from Cons\(2\)
have InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state')
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto
moreover
from Cons\(3\)
have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state') (getWatch2 ?state')
  unfolding InvariantWatchesDiffer-def
  unfolding swapWatches-def
  by auto
moreover
from Cons\(4\)
have InvariantConsistent (getM ?state')
  unfolding InvariantConsistent-def
  unfolding swapWatches-def
  by simp
moreover
from Cons\(5\)
have InvariantUniq (getM ?state')
  unfolding InvariantUniq-def
  unfolding swapWatches-def
  by simp
moreover
from Cons\(6\)
have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') M
  unfolding swapWatches-def
  unfolding InvariantWatchCharacterization-def
  unfolding watchCharacterizationCondition-def
  by simp
moreover
have \(\forall (c:\text{nat}). c \in \text{set Wl'} \rightarrow \text{Some literal } = (\text{getWatch1 ?state'} c) \land \text{Some literal } = (\text{getWatch2 ?state'} c)\)
  using Cons\(10\)
  using \(\text{clause } \notin \text{ set Wl'}\)
  using swapWatchesEffect[of clause state]
  by simp
moreover
have getM ?state'' = getM state
  getF ?state'' = getF state
  unfolding swapWatches-def
  by auto
moreover
have \( \text{getWatch1 ?fState clause} = \text{getWatch1 ?state}'' \text{ clause} \) \
getWatch2 ?fState clause = \( \text{getWatch2 ?state}'' \text{ clause} \)
using \( \langle \text{clause } \notin \text{ set } Wl \rangle \)
using \( \langle \text{InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getF ?state'' = getF state) \rangle \)
using Cons(7)
using notifyWatchesLoopPreservedWatches[of ?state'' Wl'
literal clause ≠ newWl]
by \((\text{simp add: Let-def})\)
moreover
have \( \text{literalFalse ?w1 (elements M)} \)
using \( \langle \text{literalFalse ?w1 (elements (getM ?state))} \rangle \)
\( \langle ?w1 \neq ?w2 \rangle \langle ?w2 = \text{ literal} \rangle \) Cons(8)
unfolding swapWatches-def
by auto

have \( \neg \text{ literalTrue ?w2 (elements M)} \)
using Cons(4)
using Cons(8)
using \( \langle ?w2 = \text{ literal} \rangle \)
using inconsistentCharacterization[of elements M @ [opposite literal]]
unfolding InvariantConsistent-def
by force

have \( \ast: \forall \ l. l \ el (\text{nth (getF state) clause}) \land l \neq ?w1 \land l \neq ?w2 \Rightarrow \) \
\( \text{literalFalse l (elements M)} \land \text{elementLevel (opposite l) M} \leq \) \
\( \text{elementLevel (opposite ?w1) M} \)
proof
have \( \neg (\exists \ l. l \ el (\text{nth (getF state) clause}) \land \text{ literalTrue l}) \)
(\( \text{elements M}) \)\)
proof
assume \( \exists \ l. l \ el (\text{nth (getF state) clause}) \land \text{ literalTrue l}\)
(\( \text{elements M}) \)\)
show False
proof–
from \( \exists \ l. l \ el (\text{nth (getF state) clause}) \land \text{ literalTrue l}\)
(\( \text{elements M}) \)
obtain \( l \)
where \( l \ el (\text{nth (getF state) clause}) \)\( \text{ literalTrue l}\)
(\( \text{elements M}) \)
by auto
hence \( l \neq ?w1 \land l \neq ?w2 \)
using \( \langle \neg \text{ literalTrue ?w1 (elements (getM ?state'))} \rangle \)
using \( \langle \neg \text{ literalTrue ?w2 (elements M)} \rangle \)
unfolding swapWatches-def
using Cons(8)
by auto
with \( l \in \text{nth} \ (\text{getF state clause}) \)

have \( \text{literalFalse} \ l \ (\text{elements} \ (\text{getM state})) \)
  using \( \text{getWatch1} \ ?\text{state'} \ \text{clause} = \text{Some} \ ?w1 \)
  using \( \text{getWatch2} \ ?\text{state'} \ \text{clause} = \text{Some} \ ?w2 \)
  using \( \text{None} \)
  using \( \text{getNonWatchedUnfalsifiedLiteralNoneCharacterization}[\text{of nth} \ (\text{getF state'}) \ \text{clause} \ ?w1 \ ?w2 \ \text{getM state}] \)
  unfolding \( \text{swapWatches-def} \)
  by \( \text{simp} \)

  with \( l \neq ?w2 \ ; \ ?w2 = \text{literal} \ \text{Cons}(8) \)
  have \( \text{literalFalse} \ l \ (\text{elements} \ M) \)
  unfolding \( \text{swapWatches-def} \)
  by \( \text{simp} \)

  with \( \text{Cons}(4) \ ; \text{literalTrue} \ l \ (\text{elements} \ M) \)
  show \( \oplus \) using \( \text{Cons}(8) \)
  by \( \text{(auto simp add: inconsistentCharacterization)} \)

qed

with \( \text{InvariantWatchCharacterization} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \ M \)
show \( \oplus \) unfolding \( \text{InvariantWatchCharacterization-def} \)
  using \( \text{literalFalse} \ ?w1 \ (\text{elements} \ M) \)
  using \( \text{getWatch1} \ ?\text{state'} \ \text{clause} = \text{Some} \ ?w1 \ \text{THEN sym} \)
  using \( \text{getWatch2} \ ?\text{state'} \ \text{clause} = \text{Some} \ ?w2 \ \text{THEN sym} \)
  using \( \text{elementLevel} \ (\text{opposite} \ ?w1) \ (\text{getM state'}) \leq \text{elementLevel} \ (\text{opposite} \ ?w1) \ (\text{getM state'}) \)

proof

\{ \fix \ l::\text{Literal} \\
\assume \ l \in \text{nth} \ (\text{getF state''}) \ \text{clause} \land \ l \neq ?w1 \land l \neq ?w2 \} \\

have \( \text{literalFalse} \ l \ (\text{elements} \ (\text{getM state''})) \land \\
\text{elementLevel} \ (\text{opposite} \ l) \ (\text{getM state''}) \leq \text{elementLevel} \ (\text{opposite} \ ?w1) \ (\text{getM state''}) \)

397
proof –
from * l el (nth (getF ?fState′′) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2}
have literalFalse l (elements M) elementLevel (opposite l) M ≤ elementLevel (opposite ?w1) M unfolding swapWatches-def by auto thus ?thesis using elementLevelAppend[of opposite l M [opposite literal, decision]] using literalFalse ?w1 (elements M)
using elementLevelAppend[of opposite ?w1 M [opposite literal, decision]] using Cons(8) unfolding swapWatches-def by simp
qedeq}

\[\text{have } \forall l. \text{literalFalse } l \Rightarrow \text{elementLevel } (\text{opposite } l) \text{ (getM ?fState) } \leq \text{elementLevel } (\text{opposite } ?w1) \text{ (getM ?fState)}\]

proof –
have elementLevel (opposite ?w2) (getM ?fState) = currentLevel (getM ?fState)
using Cons(8)
using ((getM ?fState) = (getM state)) 398
using ($\neg$ literalFalse $\omega_2$ (elements $M$))
using ($\omega_2 = \text{literal}$)
using elementOnCurrentLevel[of opposite $\omega_2$ $M$ decision]
by simp
thus $\neg$thesis
by (simp add: elementLevelLeqCurrentLevel)
qed
ultimately
show $\neg$thesis
using Cons(1)[of $?\text{state}$" clause # newWl]
using Cons(7) Cons(8)
using (getWatch1 $?\text{state}$' clause = Some $?w_1')
using (getWatch2 $?\text{state}$' clause = Some $?w_2')
using (Some literal = getWatch1 state clause)
using ($\neg$ literalTrue $\omega_1$ (elements (getM $?\text{state}$')))
using None
using (literalFalse $\omega_1$ (elements (getM $?\text{state}$')))
using (uniq Wl)
unfolding watchCharacterizationCondition-def
by (simp add: Let-def)
next
case False

let $?\text{state}$" = setReason $\omega_1$ clause (if $?w_1$
el (getQ $?\text{state}$') then (getQ $?\text{state}$') else (getQ $?\text{state}$') @ [?w_1])]
let $?\text{state}$ = notifyWatches-loop literal Wl' (clause # newWl)
$\text{state}$"

from Cons(2)
have InvariantWatchesEl (getF $?\text{state}$") (getWatch1 $?\text{state}$")
(getWatch2 $?\text{state}$")
unfolding InvariantWatchesEl-def
unfolding setReason-def
unfolding swapWatches-def
by auto
moreover
from Cons(3)
have InvariantWatchesDiffer (getF $?\text{state}$") (getWatch1 $?\text{state}$")
(getWatch2 $?\text{state}$")
unfolding InvariantWatchesDiffer-def
unfolding setReason-def
unfolding swapWatches-def
by auto
moreover
from Cons(4)
have InvariantConsistent (getM $?\text{state}$")
unfolding InvariantConsistent-def
unfolding setReason-def
unfolding swapWatches-def
by simp
moreover
from Cons(5)
have InvariantUniq (getM ?state'')
    unfolding InvariantUniq-def
    unfolding setReason-def
    unfolding swapWatches-def
    by simp
moreover
from Cons(6)
have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') M
    unfolding swapWatches-def
    unfolding setReason-def
    unfolding InvariantWatchCharacterization-def
    unfolding watchCharacterizationCondition-def
    by simp
moreover
have \( \forall (c::nat), c \in \text{set } Wl' \implies \text{Some literal} = (\text{getWatch1 ?state''} c) \lor \text{Some literal} = (\text{getWatch2 ?state''} c) \)
    using Cons(10)
    using \( \langle \text{clause} \notin \text{set } Wl' \rangle \)
    using swapWatchesEffect[of clause state]
    unfolding setReason-def
    by simp
moreover
have getM ?state'' = getM state
    getF ?state'' = getF state
    unfolding setReason-def
    unfolding swapWatches-def
    by auto
moreover
have getWatch1 ?state'' clause = Some ?w1 getWatch2 ?state'' clause = Some ?w2
    using \( \langle \text{getWatch1 ?state' clause} = \text{Some ?w1} \rangle \)
    using \( \langle \text{getWatch2 ?state' clause} = \text{Some ?w2} \rangle \)
    unfolding setReason-def
    unfolding swapWatches-def
    by auto
moreover
have getWatch1 ?state'' clause = Some ?w1 getWatch2 ?state'' clause = Some ?w2
    using \( \langle \text{clause} \notin \text{set } Wl' \rangle \)
    using InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getF ?state'') = getF state
    using Cons(7)
    using notifyWatchesLoopPreservedWatches[of ?state'' Wl']
literal clause \# new\text{Wl} ]
    by (auto simp add: Let-def)
moreover
    have (getM ?fState) = (getM state) (getF ?fState) = (getF state)
    using notifyWatchesLoopPreservedVariables[of ?state"] Wi'
literal clause \# new\text{Wl}
    using (InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state") (getF ?state" = getF state)
    using Cons(7)
    unfolding setReason-def
    unfolding swapWatches-def
    by (auto simp add: Let-def)
ultimately
    have \forall c. c \in \text{set \text{Wl}} \longrightarrow (\forall w1 \ w2. \ \text{Some \ w1} = \text{getWatch1} \ ?fState \ c \wedge \text{Some \ w2} = \text{getWatch2} \ ?fState \ c \longrightarrow
      \text{watchCharacterizationCondition\ w1 \ w2 \ (getM ?fState)}
    (getF ?fState \ c) \wedge
      \text{watchCharacterizationCondition\ w2 \ w1 \ (getM ?fState)}
    (getF ?fState \ c)) and
    ?fState = notifyWatches-loop literal (clause \# \text{Wl'}) new\text{Wl}
state
    using Cons(1)[of ?state" clause \# new\text{Wl}]
    using Cons(7) Cons(8)
    using (getWatch1 ?state' clause = Some ?w1)
    using (getWatch2 ?state' clause = Some ?w2)
    using (Some literal = getWatch1 state clause)
    using None
    using (\neg \text{literalTrue} \ w1 (elements (getM ?state')))
    using None
    using (\neg \text{literalFalse} \ w1 (elements (getM ?state')))
    using (uniq \text{Wl'})
    by (auto simp add: Let-def)
moreover
    have*: \forall l. l \in (\text{nth (getF ?fState')} \ clause) \wedge l \neq ?w1 \wedge l \neq ?w2 \longrightarrow\text{literalFalse l (elements (getM ?state'))}
    using None
    using (getWatch1 ?state' clause = Some ?w1)
    using (getWatch2 ?state' clause = Some ?w2)
    using getNonWatchedUnfalsifiedLiteralNoneCharacterization[of nth (getF ?fState') clause ?w1 ?w2 getM ?fState]
    using Cons(8)
    unfolding setReason-def
    unfolding swapWatches-def
    by auto

    have**: \forall l. l \in (\text{nth (getF ?fState')} \ clause) \wedge l \neq ?w1 \wedge l \neq ?w2 \longrightarrow\text{literalFalse l (elements (getM ?fState'))}
    using (getM ?fState) = (getM state); (getF ?fState) = (getF state):
using *
using (getM ?state"" = getM state)
using (getF ?state"" = getF state)
unfolding swapWatches-def
by auto

have ***: ∀ l. literalFalse l (elements (getM ?fState)) →
elementLevel (opposite l) (getM ?fState) ≤ elementLevel
(opposite ?w2) (getM ?fState)
proof −
  have elementLevel (opposite ?w2) (getM ?fState) = cur-
rentLevel (getM ?fState)
  using Cons(8)
  using ((getM ?fState) = (getM state))
  using (¬ literalFalse ?w2 (elements M))
  using (?w2 = literal)
  using elementOnCurrentLevel[of opposite ?w2 M decision]
  by simp
  thus ?thesis
  by (simp add: elementLevelLeqCurrentLevel)
qed

have (∀ w1 w2. Some w1 = getWatch1 ?fState clause ∧ Some
w2 = getWatch2 ?fState clause →
watchCharacterizationCondition w1 w2 (getM ?fState) (getF
?fState ! clause) ∧
watchCharacterizationCondition w2 w1 (getM ?fState) (getF
?fState ! clause))
proof −
  { fix w1 w2
  assume Some w1 = getWatch1 ?fState clause ∧ Some w2
= getWatch2 ?fState clause
  hence w1 = ?w1 w2 = ?w2
  using (getWatch1 ?fState clause = Some ?w1)
  using (getWatch2 ?fState clause = Some ?w2)
  by auto
  hence watchCharacterizationCondition w1 w2 (getM
?fState) (getF ?fState ! clause) ∧
watchCharacterizationCondition w2 w1 (getM ?fState)
(getF ?fState ! clause)
  unfolding watchCharacterizationCondition-def
  using ** ***
  unfolding watchCharacterizationCondition-def
  using ((getM ?fState) = (getM state)) (getF ?fState) =
(getF state):
  using (¬ literalFalse ?w1 (elements (getM ?state'))):
  unfolding swapWatches-def
  by simp

402
thus thesis
    by auto
qed
ultimately
show thesis
    by simp
qed
qed
qed
next
 case False
 let ?state' = state
 let ?w1 = wa
 have getWatch1 ?state' clause = Some ?w1
     using (getWatch1 state clause = Some wa)
     by auto
 let ?w2 = wb
 have getWatch2 ?state' clause = Some ?w2
     using (getWatch2 state clause = Some wb)
     by auto

from (∼ Some literal = getWatch1 state clause)
  (∀ (c::nat). c ∈ set (clause ≠ W1) → Some literal = (getWatch1 state c))
  ∨ Some literal = (getWatch2 state c))
  have Some literal = getWatch2 state clause
     by auto
  hence ?w2 = literal
     using (getWatch2 ?state' clause = Some ?w2)
     by simp
  hence literalFalse ?w2 (elements (getM state))
     using Cons(8)
     by simp

from (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state))
  have ?w1 el (nth (getF state) clause) ?w2 el (nth (getF state) clause)
     using (getWatch1 ?state' clause = Some ?w1)
     using (getWatch2 ?state' clause = Some ?w2)
     unfolding InvariantWatchesEl-def
     by auto

from (InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state))
  have ?w1 ≠ ?w2
     using (getWatch1 ?state' clause = Some ?w1)
     using (getWatch2 ?state' clause = Some ?w2)
using \(0 \leq \text{clause} \land \text{clause} < \text{length (getF state)}\)

unfolding InvariantWatchesDiffer-def

by auto

have \(\neg \text{literalFalse } ?w2 \ (\text{elements } M)\)
using \(?w2 = \text{literal}\)
using Cons(5)
using Cons(8)
unfolding InvariantUniq-def

by (simp add: uniqAppendIff)

show \(?thesis\)

proof (cases literalTrue \(?w1 \ (\text{elements } (\text{getM } ?\text{state}'))\))

case True

let \(?fState = \text{notifyWatches-loop literal } Wl' \ (\text{clause } \# \ \text{newWl})\) \(?state'\)

have \(\text{getWatch1 } ?fState \ \text{clause} = \text{getWatch1 } ?\text{state' } \text{clause} \land\)
\(\text{getWatch2 } ?fState \ \text{clause} = \text{getWatch2 } ?\text{state' } \text{clause}\)
using \(\text{clause } \notin \text{set } Wl'\)
using Cons(2)
using Cons(7)
using notifyWatchesLoopPreservedWatches[of \(?\text{state' } Wl' \ \text{literal}\) \(\text{clause } \# \ \text{newWl}\)]
by (simp add: Let-def)

moreover

have \(\text{watchCharacterizationCondition } ?w1 ?w2 \ (\text{getM } ?fState)\)
\((\text{getF } ?fState \ ! \ \text{clause}) \land\)
\(\text{watchCharacterizationCondition } ?w2 ?w1 \ (\text{getM } ?fState)\)
\((\text{getF } ?fState \ ! \ \text{clause})\)

proof–

have \((\text{getM } ?fState) = (\text{getM } ?\text{state}) \land (\text{getF } ?f\text{State} = \text{getF } \text{state})\)

using notifyWatchesLoopPreservedVariables[of \(?\text{state' } Wl' \ \text{literal}\) \(\text{clause } \# \ \text{newWl}\)]
using Cons(2)
using Cons(7)
by (simp add: Let-def)

moreover

have \(\neg \text{literalFalse } ?w1 \ (\text{elements } M)\)
using \(\text{literalTrue } ?w1 \ (\text{elements } (\text{getM } ?\text{state}'))\) \(\langle ?w1 \neq ?w2 \rangle\)
\(\langle ?w2 = \text{literal}\rangle\)
using Cons(4) Cons(8)
unfolding InvariantConsistent-def

by (auto simp add: inconsistentCharacterization)

moreover

have \(\text{elementLevel } (\text{opposite } ?w2) \ (\text{getM } ?\text{state'}) = \text{currentLevel } (\text{getM } ?\text{state'})\)
using ⟨?w2 = literal⟩
using Cons(5) Cons(8)
unfolding InvariantUniq-def
by (auto simp add: uniqAppendIff elementOnCurrentLevel)
ultimately
show ?thesis
using Cons(5) Cons(8)
by (auto simp add: Let-def)
qed
ultimately
show ?thesis
using assms
using Cons(1)[of ?state clause ≠ newWl]
using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7) Cons(8) Cons(9) Cons(10)
using (uniq Wl)
using (getWatch1 ?state clause = Some ?w1)
using (getWatch2 ?state clause = Some ?w2)
using (Some literal = getWatch2 state clause)
using (literalTrue ?w1 (elements (getM ?state')))
using (?w1 ≠ ?w2)
by (simp add: Let-def)
next
case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
case (Some l')
hence l' el (nth (getF ?state') clause) l' ≠ ?w1 l' ≠ ?w2 ¬ literalFalse l' (elements (getM ?state'))
using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
using getNonWatchedUnfalsifiedLiteralSomeCharacterization
by auto
let ?state'' = setWatch2 clause l' ?state'
let ?fState = notifyWatches-loop literal Wl' newWl ?state''
from Cons(2)

have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  using \text{l}' el (nth (getF ?state') clause).
  unfolding InvariantWatchesEl-def
  unfolding setWatch2-def
  by auto
moreover
from Cons(3)

have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  using \langle \text{l}' \neq ?w1 \rangle
  using \langle \text{getWatch1 ?state'} \text{ clause} = \text{Some ?w1} \rangle
  using \langle \text{getWatch2 ?state'} \text{ clause} = \text{Some ?w2} \rangle
  unfolding InvariantWatchesDiffer-def
  unfolding setWatch2-def
  by auto
moreover
from Cons(4)

have InvariantConsistent (getM ?state'')
  unfolding InvariantConsistent-def
  unfolding setWatch2-def
  by simp
moreover
from Cons(5)

have InvariantUniq (getM ?state'')
  unfolding InvariantUniq-def
  unfolding setWatch2-def
  by simp
moreover
have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') M
proof −
\{ 
  fix c::nat and \text{ww1::Literal} and \text{ww2::Literal}
  assume a: 0 \leq c \land c < \text{length (getF ?state'')} \land \text{Some \text{ww1} = (getWatch1 ?state'' c) \land Some \text{ww2} = (getWatch2 ?state'' c)}
  assume b: literalFalse \text{ww1} (\text{elements M})

  have (\exists l. l el ((getF ?state'') ! c) \land literalTrue l (\text{elements M}) \land elementLevel l M \leq elementLevel (\text{opposite \text{ww1}}) M) \lor
  (\forall l. l el ((getF ?state'') ! c) \land l \neq \text{ww1} \land l \neq \text{ww2} \rightarrow
  literalFalse l (\text{elements M}) \land elementLevel (\text{opposite \text{ww1}}) M \land elementLevel (\text{opposite \text{ww2}}) M \land \text{elementLevel (opposite \text{ww2}}) M)

  proof (cases c = \text{clause})
  case False
  thus \text{?thesis}
  using a and b
using Cons(6)
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
unfolding setWatch2-def
by simp

next
case True
with a
have \( w_1 = ?w_1 \) and \( w_2 = l' \)
  using \( \langle \text{getWatch1 ?state' clause = Some ?w_1} \rangle \)
  using \( \langle \text{getWatch2 ?state' clause = Some ?w_2} \rangle \)

sym]

unfolding setWatch2-def
by auto

have \( \neg (\forall l. l \in (\text{getF state}!\text{clause}) \land l \neq ?w_1 \land l \neq ?w_2 \rightarrow \text{literalFalse l (elements M)}) \)
using Cons(8)
using \( l' \neq ?w_1 \) and \( l' \neq ?w_2 \) and \( l' \in (\text{nth (getF ?state')}) \)

clause):

using \( \langle \neg \text{literalFalse l'} (\text{elements (getM ?state'))} \rangle \)
using a and b
using \( \langle c = \text{clause} \rangle \)
unfolding setWatch2-def
by auto

moreover
have \( \exists l. l \in (\text{getF state}!\text{clause}) \land \text{literalTrue l (elements M)} \) \land
  \( \text{elementLevel l M} \leq \text{elementLevel (opposite ?w_1) M} \) \lor
  \( \forall l. l \in (\text{getF state}!\text{clause}) \land l \neq ?w_1 \land l \neq ?w_2 \rightarrow \text{literalFalse l (elements M)} \)
using Cons(6)

unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
using \( 0 \leq \text{clause} \land \text{clause < length (getF state)} \)
using \( \langle \text{getWatch1 ?state' clause = Some ?w_1} \rangle \)

sym]

using \( \langle \text{getWatch2 ?state' clause = Some ?w_2} \rangle \)

sym]

using \( \langle \text{literalFalse w_1 (elements M)} \rangle \)
using \( \langle w_1 = ?w_1 \rangle \)
unfolding setWatch2-def
by auto

ultimately
show \( \text{thesis} \)
  using \( \langle w_1 = ?w_1 \rangle \)
  using \( \langle c = \text{clause} \rangle \)
  unfolding setWatch2-def
  by auto

407
qed
} moreover
{
  fix c::nat and ww1::Literal and ww2::Literal
  assume a: 0 ≤ c ∧ c < length (getF ?state") ∧ Some ww1 = (getWatch1 ?state" c) ∧ Some ww2 = (getWatch2 ?state" c)
  assume b: literalFalse ww2 (elements M)
  have (∃ l. l el ((getF ?state") ! c) ∧ literalTrue l (elements M) ∧ elementLevel l M ≤ elementLevel (opposite ww2) M) ∨
        (∀ l. l el ((getF ?state") ! c) ∧ l ≠ ww1 ∧ l ≠ ww2 −→
            literalFalse l (elements M) ∧ elementLevel (opposite l) M ≤ elementLevel (opposite ww2) M)
  proof (cases c = clause)
    case False
    thus ?thesis using a and b using Cons(6)
    unfolding InvariantWatchCharacterization-def
    unfolding watchCharacterizationCondition-def
    unfolding setWatch2-def
    by auto
  next
    case True
    with a
    have ww1 = ?w1 and ww2 = l'
      using ⟨getWatch1 ?state' clause = Some ?w1⟩
      using ⟨getWatch2 ?state' clause = Some ?w2⟩[THEN sym]
      unfolding setWatch2-def
      by auto
    with (¬ literalFalse l' (elements (getM ?state'))) b
      Cons(8)
    have False by simp
    thus ?thesis by simp
  qed
}
ultimately
show ?thesis
  unfolding InvariantWatchCharacterization-def
  unfolding watchCharacterizationCondition-def
  by blast
qed
moreover
  have ∀ (c::nat). c ∈ set Wl' −→ Some literal = (getWatch1 ?state" c) ∨ Some literal = (getWatch2 ?state" c)
using Cons(10)
using ⟨clause ∉ set Wl⟩
unfolding setWatch2-def
by simp

moreover
have getM ?state" = getM state
getF ?state" = getF state
unfolding setWatch2-def
by auto

moreover
have getWatch1 ?state" clause = Some ?w1 getWatch2 ?state"
clause = Some l'
using ⟨getWatch1 ?state' clause = Some ?w1⟩
unfolding setWatch2-def
by auto

hence getWatch1 ?fState clause = getWatch1 ?state" clause ∧
getWatch2 ?fState clause = Some l'
using ⟨clause ∉ set Wl⟩
using ⟨InvariantWatchesEl (getF ?state") (getWatch1 ?state")
(getWatch2 ?state")⟩ (getF ?state" = getF state)
using Cons(7)
using notifyWatchesLoopPreservedWatches[of ?state" Wl'
literal newWl]
by (simp add: Let-def)

moreover
have watchCharacterizationCondition ?w1 l' (getM ?fState)
(getF ?fState ! clause) ∧
watchCharacterizationCondition l' ?w1 (getM ?fState) (getF
?fState ! clause)
proof−
have (getM ?fState) = (getM state) (getF ?fState) = (getF
state)
using notifyWatchesLoopPreservedVariables[of ?state" Wl'
literal newWl]
using ⟨InvariantWatchesEl (getF ?state") (getWatch1
?state") (getWatch2 ?state")⟩ (getF ?state" = getF state)
using Cons(7)
unfolding setWatch2-def
by (auto simp add: Let-def)

have literalFalse ?w1 (elements M) →
(∃ l. l el (nth (getF ?state") clause) ∧ literalTrue l (elements
M) ∧ elementLevel l M ≤ elementLevel (opposite ?w1) M)
proof
assume literalFalse ?w1 (elements M)
show ∃ l. l el (nth (getF ?state") clause) ∧ literalTrue l
(elements M) ∧ elementLevel l M ≤ elementLevel (opposite ?w1) M
proof−
have ¬ (∀ l. l el (nth (getF state) clause) ∧ l ≠ ?w1 ∧ l

409
\[ \neg \ ?w_2 \rightarrow \text{literalFalse } l \left( \text{elements } M \right) \]

using \( l' \) el \( \langle \text{nth } \text{getF } ?\text{state}' \rangle \) clause \( \langle l' \neq ?w_1 : l' \neq \ ?w_2 \rangle \to \text{literalFalse } l' \left( \text{elements } \text{getM } ?\text{state}' \rangle \right) \)

using \( \text{Cons}(8) \)

unfolding \( \text{swapWatches-def} \)

by auto

from \( \langle \text{literalFalse } ?w_1 \left( \text{elements } M \right) \rangle \) Cons(6)

have

\( \exists l. l \text{ el } \langle \text{getF state} ! \text{clause} \rangle \land \text{literalTrue } l \left( \text{elements } M \right) \land \text{elementLevel } l M \leq \text{elementLevel } \langle \text{opposite } ?w_1 \rangle M \) \lor

\( \forall l. l \text{ el } \langle \text{getF state} ! \text{clause} \rangle \land l \neq ?w_1 \land l \neq ?w_2 \to \text{literalFalse } l \left( \text{elements } M \right) \land \text{elementLevel } l M \leq \text{elementLevel } \langle \text{opposite } l \rangle M \)

using \( 0 \leq \text{clause} \land \text{clause} < \text{length } \langle \text{getF state} \rangle \)

using \( \langle \text{getWatch1 } ?\text{state}' \rangle \text{ clause } = \text{Some } ?w_1 \rangle [\text{THEN sym}] \)

using \( \langle \text{getWatch2 } ?\text{state}' \rangle \text{ clause } = \text{Some } ?w_2 \rangle [\text{THEN sym}] \)

unfolding \( \text{InvariantWatchCharacterization-def} \)

unfolding \( \text{watchCharacterizationCondition-def} \)

by simp

with \( \to \langle \forall l. l \text{ el } \langle \text{getF state} ! \text{clause} \rangle \land l \neq ?w_1 \land l \neq ?w_2 \to \text{literalFalse } l \left( \text{elements } M \right) \rangle \)

have \( \exists l. l \text{ el } \langle \text{getF state} ! \text{clause} \rangle \land \text{literalTrue } l \left( \text{elements } M \right) \land \text{elementLevel } l M \leq \text{elementLevel } \langle \text{opposite } ?w_1 \rangle M \)

by auto

thus \( ?\text{thesis} \)

unfolding \( \text{setWatch2-def} \)

by simp

qed

qed

moreover

have \( \text{watchCharacterizationCondition } ?w_1 \ ?w_1' \left( \text{getM } ?\text{fState} \rangle \right) \left( \text{getF } ?\text{fState} ! \text{clause} \right) \)

using \( \langle \to \text{literalFalse } l' \left( \text{elements } \text{getM } ?\text{state}' \rangle \right) \rangle \)

using \( \langle \text{getM } ?\text{fState } = \text{getM } \text{state} \rangle \)

unfolding \( \text{watchCharacterizationCondition-def} \)

by simp

moreover

have \( \text{watchCharacterizationCondition } ?w_1 \ ?w_1' \left( \text{getM } ?\text{fState} \rangle \left( \text{getF } ?\text{fState} ! \text{clause} \right) \right) \)

proof \( \langle \text{cases literalFalse } ?w_1 \left( \text{elements } \text{getM } ?\text{fState} \rangle \right) \rangle \)

case True

hence \( \text{literalFalse } ?w_1 \left( \text{elements } M \right) \)

using \( \text{notifyWatchesLoopPreservedVariables} \langle \text{of } ?\text{state}' \rangle \langle ?w_1 \rangle \text{ literal newWL} \)

using \( \langle \text{InvariantWatchesEl } \langle \text{getF } ?\text{state}' \rangle \rangle \langle \text{getWatch1 } ?\text{state}' \rangle \langle \text{getWatch2 } ?\text{state}' \rangle \rangle \langle \text{getF } ?\text{state}' \rangle = \text{getF state} \rangle \)

410
using Cons(7) Cons(8)
using (?w1 ≠ ?w2): (?w2 = literal)
unfolding setWatch2-def
by (simp add: Let-def)
with ⟨literalsemFalse ?w1 (elements M) ⟩ ⟨literalsemTrue ?w2 (elements M) ⟩
(∃ l. l el (nth (getF ?state") clause) ∧ literalTrue l (elements M) ∧
  elementLevel l M ≤ elementLevel (opposite ?w1) M):
  obtain l::Literal
              where l el (nth (getF ?state") clause) and
              literalTrue l (elements M) and
              elementLevel l M ≤ elementLevel (opposite ?w1) M
              by auto
  hence elementLevel l (getM state) ≤ elementLevel (opposite ?w1) (getM state)
       using Cons(8)
       using ⟨literalsemTrue l (elements M) ⟩ ⟨literalsemFalse ?w1 (elements M) ⟩
       using elementLevelAppend[of l M [(opposite literal, decision)]]
       using elementLevelAppend[of opposite ?w1 M [(opposite literal, decision)]]
       by auto
thus ?thesis
using ⟨l el (nth (getF ?state") clause): (literalsemTrue l (elements M)) ⟩
using ⟨getM ?fState = getM state ⟩ ⟨getF ?fState = getF state ⟩ ⟨getM ?state" = getM state ⟩ ⟨getF ?state" = getF state ⟩
using Cons(8)
unfolding watchCharacterizationCondition-def
by auto
next
  case False
  thus ?thesis
unfolding watchCharacterizationCondition-def
by simp
qed
ultimately
show ?thesis
by simp
qed
ultimately
show ?thesis
using Cons(1)[of ?state" newWl]
using Cons(7) Cons(8)
using ⟨getWatch1 ?state' clause = Some ?w1 ⟩
using ⟨getWatch2 ?state' clause = Some ?w2 ⟩
using ⟨Some literal = getWatch2 state clause ⟩
using ⟨¬ literalTrue ?w1 (elements (getM ?state")) ⟩
using ⟨getWatch1 ?state" clause = Some ?w1 ⟩
using ⟨getWatch2 ?state' clause = Some l'⟩
using Some
using ⟨uniq Wl'⟩
using ⟨?w1 ≠ ?w2⟩
by (simp add: Let-def)

next
case None
  show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state')))
case True
  let ?state'' = ?state'⟨getConflictFlag := True, getConflict-Clause := clause⟩
  let ?fState = notifyWatches-loop literal Wl' (clause # newWl) ?state''

  from Cons(2)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    unfolding InvariantWatchesEl-def
    by auto
  moreover
  from Cons(3)
  have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    unfolding InvariantWatchesDiffer-def
    by auto
  moreover
  from Cons(4)
  have InvariantConsistent (getM ?state')
    unfolding InvariantConsistent-def
    by simp
  moreover
  from Cons(5)
  have InvariantUniq (getM ?state')
    unfolding InvariantUniq-def
    by simp
  moreover
  from Cons(6)
  have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') M
    unfolding InvariantWatchCharacterization-def
    unfolding watchCharacterizationCondition-def
    by simp
  moreover
  have ∀ (c::nat). c ∈ set Wl' → Some literal = (getWatch1 ?state'' c) ∨ Some literal = (getWatch2 ?state'' c)
    using Cons(10)
    using ⟨clause ∉ set Wl'⟩
    by simp

412
moreover
have $\text{getM } ?\text{state}'' = \text{getM state}$
getF $?\text{state}'' = \text{getF state}$
by auto
moreover
have $\text{getWatch1 } ?\text{fState clause} = \text{getWatch1 } ?\text{state}'' \text{ clause}$ ∧
$\text{getWatch2 } ?\text{state clause} = \text{getWatch2 } ?\text{state}'' \text{ clause}$
using $\langle \text{clause } \notin \text{ set } \text{W}' \rangle$
using $\langle \text{InvariantWatchesEl } (\text{getF } ?\text{state}'') (\text{getWatch1 } ?\text{state}'') (\text{getWatch2 } ?\text{state}'') \rangle (\text{getF } ?\text{state}'' = \text{getF state})$
using $\text{Cons}(7)$
using notifyWatchesLoopPreservedWatches[of \text{?state}'' W]
literal clause ≠ new\text{W}'
by (simp add: Let-def)
moreover
have $\text{literalFalse } \text{w1 (elements } M)$
using $\langle \text{literalFalse } \text{w1 (elements } \text{getM ?state}') \rangle$
$\langle \text{w1 } \neq \text{w2 }, \text{w2 } = \text{literal} \rangle \text{ Cons}(8)$
by auto
have $\neg \text{ literalTrue } \text{w2 (elements } M)$
using $\text{Cons}(4)$
using $\text{Cons}(8)$
using $\langle \text{w2 } = \text{literal} \rangle$
using inconsistentCharacterization[of elements $M \triangleq \text{opposite literal}]
unfolding \text{InvariantConsistent-def}
by force

have $*: \forall l. l \text{ el (nth } \text{getF state) clause} \land l \neq \text{w1 } \land l \neq \text{w2}$
$\rightarrow$
$\text{literalFalse l (elements } M) \land \text{elementLevel (opposite l) } M \leq$
$\text{elementLevel (opposite } \text{w1) } M$
proof−
have $\neg (\exists l. l \text{ el (nth } \text{getF state) clause} \land \text{literalTrue l}$
$\langle \text{elements } M \rangle)$
proof
assume $\exists l. l \text{ el (nth } \text{getF state) clause} \land \text{literalTrue l}$
$\langle \text{elements } M \rangle$
show False
proof−
from $\exists l. l \text{ el (nth } \text{getF state) clause} \land \text{literalTrue l}$
$\langle \text{elements } M \rangle$
obtain l
where $l \text{ el (nth } \text{getF state) clause} \land \text{literalTrue l}$
$\langle \text{elements } M \rangle$
by auto
hence $l \neq \text{w1 } l \neq \text{w2}$
using $\langle \neg \text{ literalTrue } \text{w1 (elements } \text{getM ?state}') \rangle$
using (¬ literalTrue ?w2 (elements M))
using Cons(8)
by auto

with l el (nth (getF state) clause)

have literalFalse l (elements (getM ?state'))
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using None
  using getNonWatchedUnfalsifiedLiteralNoneCharacter-
   ization[of nth (getF ?state') clause ?w1 ?w2 getM ?state']
  by simp

with l ≠ ?w2 : ?w2 = literal? Cons(8)

have literalFalse l (elements M)
  by simp

with Cons(4) (literalTrue l (elements M))

show ?thesis
  unfolding InvariantConsistent-def
  using Cons(8)
  by (auto simp add: inconsistentCharacterization)

qed

qed

with InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) M

show ?thesis
  unfolding InvariantWatchCharacterization-def
  using literalFalse ?w1 (elements M)
  using (getWatch1 ?state' clause = Some ?w1) [THEN sym]
  using (getWatch2 ?state' clause = Some ?w2) [THEN sym]
  using (0 ≤ clause ∧ clause < length (getF state))
  unfolding watchCharacterizationCondition-def
  by (simp (blast)

qed

have **: ∀ l. l el (nth (getF ?state'') clause) ∧ l ≠ ?w1 ∧ l
≠ ?w2 →
  literalFalse l (elements (getM ?state'')) ∧
  elementLevel (opposite l) (getM ?state'') ≤ elementLevel
  (opposite ?w1) (getM ?state'')

proof

{ fix l::Literal
  assume l el (nth (getF ?state'') clause) ∧ l ≠ ?w1 ∧ l ≠
  ?w2

  have literalFalse l (elements (getM ?state'')) ∧
    elementLevel (opposite l) (getM ?state'') ≤ elementLevel
    (opposite ?w1) (getM ?state'')

  414
proof
  from * l el (nth (getF ?state") clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2

  have literalFalse l (elements M) elementLevel (opposite l) M ≤ elementLevel (opposite ?w1) M
    by auto
  thus ?thesis
    using elementLevelAppend[of opposite l M [(opposite literal, decision)]]
    using :literalFalse ?w1 (elements M)
    using elementLevelAppend[of opposite ?w1 M [(opposite literal, decision)]]
    using Cons(8)
    by simp
  qed

  thus ?thesis
    by simp
  qed

  have (getM ?fState) = (getM state) (getF ?fState) = (getF state)
    using notifyWatchesLoopPreservedVariables[of ?state" Wl"
    literal clause ≠ newWl]
    using (InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state") (getF ?state" = getF state)
    using Cons(7)
    by (auto simp add: Let-def)
  hence ∀ l. l el (nth (getF ?fState) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 −→
    literalFalse l (elements (getM ?fState)) ∧
    elementLevel (opposite l) (getM ?fState) ≤ elementLevel (opposite ?w1) (getM ?fState)
    using **
    using (getM ?state" = getM state)
    using (getF ?state" = getF state)
    by simp
  moreover
  have ∀ l. literalFalse l (elements (getM ?fState)) −→
    elementLevel (opposite l) (getM ?fState) ≤ elementLevel (opposite ?w2) (getM ?fState)
  proof
    have elementLevel (opposite ?w2) (getM ?fState) = currentLevel (getM ?fState)
      using Cons(8)
      using ((getM ?fState) = (getM state))
      using (~ literalFalse ?w2 (elements M))
      using (?w2 = literal)
      using elementOnCurrentLevel[of opposite ?w2 M decision]
by simp
thus ?thesis
by (simp add: elementLevelLeqCurrentLevel)
qed
ultimately
show ?thesis
  using Cons(1)[of ?state’’ clause # newWl]
  using Cons(7) Cons(8)
  using (getWatch1 ?state’ clause = Some ?w1)
  using (getWatch2 ?state’ clause = Some ?w2)
  using (Some literal = getWatch2 state clause)
  using (∼ literalTrue ?w1 (elements (getM ?state’')))
  using None
  using (literalFalse ?w1 (elements (getM ?state’’)))
  using (uniq WI’)
  using (?w1 ≠ ?w2)
  unfolding watchCharacterizationCondition-def
  by (simp add: Let-def)
next
case False

  let ?state’’ = setReason ?w1 clause (?state’’(getQ := (if ?w1
    el (getQ ?state’’) then (getQ ?state’’) else (getQ ?state’’) @ [?w1])))
  let ?fState = notifyWatches-loop literal WI’’ (clause # newWl)
  ?state’”

  from Cons(2)
  have InvariantWatchesEl (getF ?state’’)(getWatch1 ?state’’)
    (getWatch2 ?state’’)
    unfolding InvariantWatchesEl-def
    unfolding setReason-def
    by auto
  moreover
  from Cons(3)
  have InvariantWatchesDiffer (getF ?state’’)(getWatch1
    ?state’’)(getWatch2 ?state’’)
    unfolding InvariantWatchesDiffer-def
    unfolding setReason-def
    by auto
  moreover
  from Cons(4)
  have InvariantConsistent (getM ?state’’)
    unfolding InvariantConsistent-def
    unfolding setReason-def
    by simp
  moreover
  from Cons(5)
  have InvariantUniq (getM ?state’’)
    unfolding InvariantUniq-def
unfolding setReason-def
by simp
moreover
from Cons(6)
have InvariantWatchCharacterization (getF ?state") (getWatch1 ?state") (getWatch2 ?state") M
  unfolding setReason-def
  unfolding InvariantWatchCharacterization-def
  unfolding watchCharacterizationCondition-def
  by simp
moreover
have \( \forall (c::nat). c \in \text{set}\ Wl' \rightarrow \text{Some literal} = (\text{getWatch1}\ ?state"\ c) \lor \text{Some literal} = (\text{getWatch2}\ ?state"\ c) \)
  using Cons(10)
  using \( \{\text{clause} \notin \text{set}\ Wl}\)\n  unfolding setReason-def
  by simp
moreover
have \( \text{getM}\ ?state" = \text{getM}\ state \)
  \( \text{getF}\ ?state" = \text{getF}\ state \)
  unfolding setReason-def
  by auto
moreover
have getWatch1 ?state" clause = Some ?w1 getWatch2 ?state" clause = Some ?w2
  using \( \{\text{getWatch1}\ ?state'\ clause = \text{Some}\ ?w1\)\)
  using \( \{\text{getWatch2}\ ?state'\ clause = \text{Some}\ ?w2\)\)
  unfolding setReason-def
  by auto
moreover
have getWatch1 ?fState clause = Some ?w1 getWatch2 ?fState clause = Some ?w2
  using \( \{\text{getWatch1}\ ?state"\ clause = \text{Some}\ ?w1\) (getWatch2 ?state"\ clause = Some ?w2\)\)
  using \( \{\text{clause} \notin \text{set}\ Wl'\)\)
  using \( \{\text{InvariantWatchesEl}\ (\text{getF}\ ?state")\ (\text{getWatch1}\ ?state")\ (\text{getWatch2}\ ?state")\ (\text{getF}\ ?state" = \text{getF}\ state)\)
  using Cons(7)
  using notifyWatchesLoopPreservedWatches[of ?state" Wl' literal clause # newWl]\n  by (auto simp add: Let-def)
moreover
have \( \{\text{getM}\ ?fState\} = (\text{getM}\ state) \)
  \( \{\text{getF}\ ?fState\} = (\text{getF}\ state)\)
  using notifyWatchesLoopPreservedVariables[of ?state" Wl' literal clause # newWl]\n  using \( \{\text{InvariantWatchesEl}\ (\text{getF}\ ?state")\ (\text{getWatch1}\ ?state")\ (\text{getWatch2}\ ?state")\ (\text{getF}\ ?state" = \text{getF}\ state)\)
  using Cons(7)
unfolding setReason-def

by (auto simp add: Let-def)

ultimately

have \( \forall c. c \in \text{ set } Wl' \rightarrow (\forall w1 \ w2. \text{ Some } w1 = \text{ getWatch1 } ?\text{fState } c \ \wedge \ \text{ Some } w2 = \text{ getWatch2 } ?\text{fState } c \rightarrow \text{ watchCharacterizationCondition } w1 \ w2 \ (\text{ getM } ?\text{fState }) \ (\text{ getF } ?\text{fState } c) \ \wedge \ \text{ watchCharacterizationCondition } w2 \ w1 \ (\text{ getM } ?\text{fState }) \ (\text{ getF } ?\text{fState } c)) \) and

\( \text{ ?State } = \text{ notifyWatches-loop literal (clause } \# \ Wl) \text{ newWl state} \)

using Cons(1)[of ?state” clause # newWl]

using Cons(7) Cons(8)

using (getWatch1 ?state’ clause = Some ?w1)

using (getWatch2 ?state’ clause = Some ?w2)

using (Some literal = getWatch2 state clause)

using (\( \neg \text{ literalTrue } ?w1 \ (\text{ elements } (\text{ getM } ?\text{state’})) \))

using None

using (\( \neg \text{ literalFalse } ?w1 \ (\text{ elements } (\text{ getM } ?\text{state’})) \))

using (uniq Wl’)

by (auto simp add: Let-def)

moreover

have \( \ast: \forall \ l. \ l \in \ (\text{ nth } (\text{ getF } ?\text{state”}) \ clause) \land \ l \neq \ ?w1 \land \ l \neq \ ?w2 \rightarrow \text{ literalFalse } l \ (\text{ elements } (\text{ getM } ?\text{state”})) \)

using None

using (getWatch1 ?state’ clause = Some ?w1)

using (getWatch2 ?state’ clause = Some ?w2)

using (\text{ getNonWatchedUnfalsifiedLiteralNoneCharacterization}[of nth (getF ?state’) clause ?w1 ?w2 getM ?state’])

using Cons(8)

unfolding setReason-def

by auto

have \( \ast\ast: \forall \ l. \ l \in \ (\text{ nth } (\text{ getF } ?\text{fState}) \ clause) \land \ l \neq \ ?w1 \land \ l \neq \ ?w2 \rightarrow \text{ literalFalse } l \ (\text{ elements } (\text{ getM } ?\text{fState})) \)

using (getM ?fState) = (getM state)

\( \ast\ast\ast: \forall \ l. \ \text{ literalFalse } l \ (\text{ elements } (\text{ getM } ?\text{fState})) \rightarrow \) elementLevel (opposite ?w2) (getM ?fState) \leq elementLevel (opposite ?w2) (getM ?fState)

proof-

have elementLevel (opposite ?w2) (getM ?fState) = currentLevel (getM ?fState)

using Cons(8)
using \( (\text{getM } \text{?fState}) = (\text{getM state}) \)  
using \( \neg \text{literalFalse } \text{?w2 } \)  
using \( \text{?w2 } = \text{literal} \)  
using \( \text{elementOnCurrentLevel} \)  
by simp  
thus \( \text{?thesis} \)  
by (simp add: elementLevelLeqCurrentLevel)  
qed

have \( \forall \text{w1 w2}. \text{Some } \text{w1 } = \text{getWatch1 } \text{?fState clause} \land \text{Some } \text{w2 } = \text{getWatch2 } \text{?fState clause} \rightarrow \)  
\( \text{watchCharacterizationCondition } \text{w1 } \text{w2 } \)  
\( \text{getM } \text{?fState} \)  
\( \text{getF } \text{?fState} \)  
\( \text{?fState } ! \)  
\( \text{clause} \)  
\( \land \)  
\( \text{watchCharacterizationCondition } \text{w2 } \text{w1 } \)  
\( \text{getM } \text{?fState} \)  
\( \text{getF } \text{?fState} \)  
\( \text{!clause} \)  
proof-  
  \{  
  fix \text{w1 w2}  
  assume Some \text{w1 } = \text{getWatch1 } \text{?fState clause} \land \text{Some } \text{w2 } = \text{getWatch2 } \text{?fState clause}  
  hence \text{w1 } = \text{?w1 w2 } = \text{?w2}  
  using \( \text{getWatch1 } \text{?fState clause} = \text{Some } \text{?w1} \)  
  using \( \text{getWatch2 } \text{?fState clause} = \text{Some } \text{?w2} \)  
  by auto  
  hence \text{watchCharacterizationCondition } \text{w1 } \text{w2 } \)  
\( \text{getM } \text{?fState} \)  
\( \text{getF } \text{?fState} \)  
\( \text{?fState } ! \)  
\( \text{clause} \)  
unfolding \( \text{watchCharacterizationCondition-def} \)  
using ** ***  
unfolding \( \text{watchCharacterizationCondition-def} \)  
using \( \neg (\text{getM } \text{?fState}) = (\text{getM state}) \rightarrow \) \( \text{(getF } \text{?fState}) = \text{?w2 } \)  
\( \text{elements } \) \( \text{(getM ?state')} \)  
by simp  
  \}  
thus \( \text{?thesis} \)  
by auto  
qed
ultimately  
show \( \text{?thesis} \)  
by simp  
qed  
qed  
qed  
419
lemma NotifyWatchesLoopConflictFlagEffect:

fixes literal :: Literal and \( Wl :: \text{nat list} \) and newWl :: \text{nat list} and state :: State

assumes

\[ \text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)} \]

and

\[ \forall (c::\text{nat}). \ c \in \text{set } Wl \implies 0 \leq c \land c < \text{length (getF state)} \] and

\[ \text{InvariantConsistent (getM state)} \]

\[ \forall (c::\text{nat}). \ c \in \text{set } Wl \implies \text{Some literal} = (\text{getWatch1 state } c) \lor \text{Some literal} = (\text{getWatch2 state } c) \]

\[ \text{literalFalse literal (elements (getM state))} \]

\[ \text{uniq } Wl \]

shows

let state' = notifyWatches-loop literal Wl newWl state in

getConflictFlag state' =

\[ (\text{getConflictFlag state} \lor \exists \text{clause. clause } \in \text{set } Wl \land \text{clauseFalse (nth (getF state) clause)} \ (\text{elements (getM state)})) \]

using assms

proof (induct Wl arbitrary: newWl state)

case Nil

thus \(?case\)

by simp

next

case (Cons clause Wl')

from \(\text{uniq (clause } \# Wl')\):

have \(\text{uniq } Wl' \land \text{clause } \notin \text{set } Wl'\)

by (auto simp add: uniqAppendIff)

from \(\forall (c::\text{nat}). \ c \in \text{set } (\text{clause } \# Wl') \implies 0 \leq c \land c < \text{length (getF state)}\):

have \(0 \leq \text{clause clause} < \text{length (getF state)}\)

by auto

then obtain \(wa::\text{Literal} \land wb::\text{Literal}\)

where getWatch1 state clause = Some wa and getWatch2 state clause = Some wb

using Cons

unfolding InvariantWatchesEl-def

by auto

show \(?case\)

proof (cases Some literal = getWatch1 state clause)

case True

let \(?state' = \text{swapWatches clause state}\)

let \(?w1 = wb\)

have getWatch1 ?state' clause = Some ?w1

using getWatch2 state clause = Some \(wb\)

unfolding swapWatches-def

by auto
let \( w_2 = w_a \)

have \( \text{getWatch2 \ ?state'} \ clause = \text{Some} \ ?w_2 \)
  using \( \text{getWatch1 \ state} \ clause = \text{Some} \ w_a \):
  unfolding \( \text{swapWatches-def} \)
  by \( \text{auto} \)

from \( \langle \text{Some literal} = \text{getWatch1 \ state} \ clause \rangle \)
  \( \langle \text{getWatch2 \ ?state'} \ clause = \text{Some} \ ?w_2 \rangle \)
  \( \langle \text{literalFalse literal (elements \ (getM \ state))} \rangle \):
  have \( \text{literalFalse \ ?w_2 \ (elements \ (getM \ state))} \)
  unfolding \( \text{swapWatches-def} \)
  by \( \text{simp} \)

from \( \langle \text{InvariantWatchesEl} \ (\text{getF \ state}) \ (\text{getWatch1 \ state}) \ (\text{getWatch2 \ state}) \rangle \):
  have \( \ ?w_1 \ \text{el} \ (\text{nth} \ (\text{getF \ state}) \ \text{clause}) \)
  using \( \langle \text{getWatch1 \ ?state'} \ clause = \text{Some} \ ?w_1 \rangle \)
  using \( \langle \text{getWatch2 \ ?state'} \ clause = \text{Some} \ ?w_2 \rangle \)
  using \( \langle \text{clause < length \ (getF \ state)} \rangle \):
  unfolding \( \text{InvariantWatchesEl-def} \)
  unfolding \( \text{swapWatches-def} \)
  by \( \text{auto} \)

show \( ?\text{thesis} \)
proof (cases \( \text{literalTrue \ ?w_1 \ (elements \ (getM \ ?state'))} \))
  case \( \text{True} \)

  from \( \text{Cons}(2) \)
  have \( \text{InvariantWatchesEl} \ (\text{getF \ ?state'}) \ (\text{getWatch1 \ ?state'}) \)
  \( (\text{getWatch2 \ ?state'}) \)
  unfolding \( \text{InvariantWatchesEl-def} \)
  unfolding \( \text{swapWatches-def} \)
  by \( \text{auto} \)
moreover
  have \( \text{getF \ ?state'} = \text{getF \ state} \land \)
  \( \text{getM \ ?state'} = \text{getM \ state} \land \)
  \( \text{getConflictFlag \ ?state'} = \text{getConflictFlag \ state} \)
  unfolding \( \text{swapWatches-def} \)
  by \( \text{simp} \)
moreover
  have \( \forall \ c. \ c \in \text{set} \ \text{Wl'} \rightarrow \text{Some literal} = \text{getWatch1 \ ?state'} \ c \lor \)
  \( \text{Some literal} = \text{getWatch2 \ ?state'} \ c \)
  using \( \text{Cons}(5) \)
  unfolding \( \text{swapWatches-def} \)
  by \( \text{auto} \)
moreover
  have \( \neg \ \text{clauseFalse \ (nth \ (getF \ state) \ clause) \ (elements \ (getM \ state))} \)
using ⟨?w1 el (nth (getF state) clause)⟩
using ⟨literalTrue ?w1 (elements (getM ?state'))⟩
using ⟨InvariantConsistent (getM state)⟩
unfolding ⟨InvariantConsistent-def⟩
unfolding ⟨swapWatches-def⟩
by (auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization)
ultimately show ?thesis

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>using Cons(1)</td>
<td>of ?state' clause ≠ newWI</td>
<td></td>
</tr>
<tr>
<td>using Cons(3)</td>
<td>Cons(4) Cons(6)</td>
<td></td>
</tr>
<tr>
<td>using ⟨getWatch1 ?state' clause = Some ?w1⟩</td>
<td></td>
<td></td>
</tr>
<tr>
<td>using ⟨getWatch2 ?state' clause = Some ?w2⟩</td>
<td></td>
<td></td>
</tr>
<tr>
<td>using ⟨Some literal = getWatch1 state clause⟩</td>
<td></td>
<td></td>
</tr>
<tr>
<td>using ⟨literalTrue ?w1 (elements (getM ?state'))⟩</td>
<td></td>
<td></td>
</tr>
<tr>
<td>using ⟨uniq WI'⟩</td>
<td></td>
<td></td>
</tr>
<tr>
<td>by (auto simp add: Let-def)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

next
case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
case ⟨Some l'⟩
hence l' el (nth (getF ?state') clause) ¬ literalFalse l' (elements (getM ?state'))
using ⟨getNonWatchedUnfalsifiedLiteralSomeCharacterization⟩
by auto

let ?state'' = setWatch2 clause l' ?state'

from Cons(2)
have ⟨InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')⟩
using ⟨l' el (nth (getF ?state') clause)⟩
unfolding ⟨InvariantWatchesEl-def⟩
unfolding ⟨swapWatches-def⟩
unfolding ⟨setWatch2-def⟩
by auto
moreover
from Cons(4)
have ⟨InvariantConsistent (getM ?state'')⟩
unfolding ⟨setWatch2-def⟩
unfolding ⟨swapWatches-def⟩
by simp
moreover
have ⟨getM ?state'' = getM state ∧ getF ?state'' = getF state ∧ getConflictFlag ?state'' = getConflictFlag state⟩
unfolding ⟨swapWatches-def⟩
unfolding setWatch2-def
by simp
moreover
have ∀ c ∈ set Wl′ → Some literal = getWatch1 ?state′′ c
∨ Some literal = getWatch2 ?state′′ c
using Cons(5)
using ⟨clause \notin set Wl′⟩
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
have ¬ clauseFalse (nth (getF state) clause) (elements (getM state))
using ⟨l′ el (nth (getF ?state′) clause)⟩
using ⟨¬ literalFalse l′ (elements (getM ?state′))⟩
using ⟨InvariantConsistent (getM state)⟩
unfolding InvariantConsistent-def
unfolding swapWatches-def
by (auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization)
ultimately
show ?thesis
using Cons(1)[of ?state′′ newWl]
using Cons(3) Cons(4) Cons(6)
using ⟨getWatch1 ?state′ clause = Some ?w1⟩
using ⟨getWatch2 ?state′ clause = Some ?w2⟩
using ⟨Some literal = getWatch1 state clause⟩
using ⟨¬ literalTrue ?w1 (elements (getM ?state′))⟩
using ⟨uniq Wl′⟩
using Some
by (auto simp add: Let-def)
next
case None
hence ∀ l. l el (nth (getF ?state′) clause) ∧ l \neq ?w1 ∧ l \neq ?w2 → literalFalse l (elements (getM ?state′))
using getNonWatchedUnfalsifiedLiteralNoneCharacterization
by simp
show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state′)))
case True
let ?state′′ = ?state'⟨getConflictFlag := True, getConflictClause := clause⟩

from Cons(2)
have InvariantWatchesEl (getF ?state′′) (getWatch1 ?state′′) (getWatch2 ?state′′)
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
by auto
moreover
from Cons(4)
have InvariantConsistent (getM ?state"
  unfolding setWatch2-def
  unfolding swapWatches-def
  by simp
moreover
have getM ?state" = getM state \ngetF ?state" = getF state \ngetSATFlag ?state" = getSATFlag state
  unfolding swapWatches-def
  by simp
moreover
have \forall c. c \in set \textit{WI}' \rightarrow \text{Some literal} = \text{getWatch1} ?\text{state}"
c \lor \text{Some literal} = \text{getWatch2} ?\text{state}"
c
  using Cons(5)
  using \langle \text{clause} \notin \text{set \textit{WI}} \rangle
  unfolding swapWatches-def
  unfolding setWatch2-def
  by auto
moreover
have clauseFalse (nth (getF state) clause) (elements (getM state))
  using \forall l. l el (nth (getF ?state') clause) \land l \neq \textcolor{red}{?w1} \land l \neq \textcolor{red}{?w2}
  \rightarrow literalFalse l (elements (getM ?state'))
  using literalFalse \textcolor{red}{?w1} (elements (getM ?state'))
  using literalFalse \textcolor{red}{?w2} (elements (getM state))
  unfolding swapWatches-def
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
ultimately
show \textcolor{red}{?thesis}
  using Cons(1)\[ of \textcolor{red}{?state'}\] clause \# new\textit{WI}\]
  using Cons(3) Cons(4) Cons(6)
  using \langle \text{getWatch1} \textcolor{red}{?state'}\rangle \text{clause} = \text{Some} \textcolor{red}{?w1}
  using \langle \text{getWatch2} \textcolor{red}{?state'}\rangle \text{clause} = \text{Some} \textcolor{red}{?w2}
  using \langle \textcolor{red}{\text{Some literal}} = \text{getWatch1} \textcolor{red}{state'}\text{clause} \rangle
  using None
  using \langle literalFalse \textcolor{red}{?w1} \text{elements (getM ?state')} \rangle
  using \langle \textcolor{red}{\text{uniq \textit{WI}}} \rangle
  by (auto simp add: \textcolor{red}{Let-def})
next
case False
  let \textcolor{red}{?state'}' = setReason \textcolor{red}{?w1} clause \langle\textcolor{red}{?state'}(\textcolor{red}{get\textcolor{red}{Q} := (if \textcolor{red}{?w1} el (\textcolor{red}{get\textcolor{red}{Q} ?state'}) then (\textcolor{red}{get\textcolor{red}{Q} ?state'}) else (\textcolor{red}{get\textcolor{red}{Q} ?state'}) \@ \textcolor{red}{?w1})\rangle\rangle
from Cons(2)
have InvariantWatchesEl (getF ?state") (getWatch1 ?state")
  (getWatch2 ?state")
unfolding \textit{InvariantWatchesEl-def}

unfolding \textit{swapWatches-def}

unfolding \textit{setReason-def}

by \textit{auto}

moreover

from \textit{Cons(4)}

have \textit{InvariantConsistent (getM ?state'')}

unfolding \textit{swapWatches-def}

unfolding \textit{setReason-def}

by \textit{simp}

moreover

have \textit{getM ?state'' \mathbin{=} getM state} \land

\textit{getF ?state'' \mathbin{=} getF state} \land

\textit{getSATFlag ?state'' \mathbin{=} getSATFlag state}

unfolding \textit{swapWatches-def}

unfolding \textit{setReason-def}

by \textit{simp}

moreover

have \(\forall c. \ (c \in \text{set } \mathit{Wl} \implies \text{Some literal} = \text{getWatch1 ?state''} \ c \lor \text{Some literal} = \text{getWatch2 ?state''} \ c)

using \textit{Cons(5)}

using \(\langle \text{clause} \not\in \text{set } \mathit{Wl}' \rangle\)

unfolding \textit{swapWatches-def}

unfolding \textit{setReason-def}

by \textit{auto}

moreover

have \(\neg \text{clauseFalse (nth (getF state) clause) (elements (getM state))}

using \(\langle ?w1 \in (\text{nth (getF state) clause}) \rangle\)

using \(\neg \text{literalFalse ?w1 (elements (getM ?state'))}\)

using \(\text{InvariantConsistent (getM state)}\)

unfolding \textit{InvariantConsistent-def}

unfolding \textit{swapWatches-def}

by (\textit{auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization})

ultimately

show \ ?thesis

using \textit{Cons(1)[of ?state''] clause \# newWl]}

using \textit{Cons(3) Cons(4) Cons(6)}

using \(\langle \text{getWatch1 ?state' clause = Some ?w1} \rangle\)

using \(\langle \text{getWatch2 ?state' clause = Some ?w2} \rangle\)

using \(\langle \text{Some literal = getWatch1 state clause} \rangle\)

using \(\neg \text{literalTrue ?w1 (elements (getM ?state'))}\)

using \textit{None}

using \(\neg \text{literalFalse ?w1 (elements (getM ?state'))}\)

using \textit{uniq Wl'}

apply (\textit{simp add: Let-def})

unfolding \textit{setReason-def}

unfolding \textit{swapWatches-def}
by auto
qed
ded
ded
next
case False
let ?state' = state
let ?w1 = wa
have getWatch1 ?state' clause = Some ?w1
  using (getWatch1 state clause = Some wa)
  unfolding swapWatches-def
  by auto
let ?w2 = wb
have getWatch2 ?state' clause = Some ?w2
  using (getWatch2 state clause = Some wb)
  unfolding swapWatches-def
  by auto
from (¬ Some literal = getWatch1 state clause)
  (∀ (c::nat). c ∈ set (clause ≠ W1) → Some literal = (getWatch1 state c)
      ∀ Some literal = (getWatch2 state c))
  have Some literal = getWatch2 state clause
  by auto
hence literalFalse ?w2 (elements (getM state))
  using ⟨getWatch2 ?state' clause = Some ?w2⟩
  ⟨literalFalse literal (elements (getM state))⟩
  by simp
from (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state))
  have ?w1 el (nth (getF state) clause)
    using (getWatch1 ?state' clause = Some ?w1)
    using (getWatch2 ?state' clause = Some ?w2)
    using (clause < length (getF state))
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    by auto
show ?thesis
proof (cases literalTrue ?w1 (elements (getM ?state')))
case True

  have ¬ clauseFalse (nth (getF state) clause) (elements (getM state))
    using (?w1 el (nth (getF state) clause))
    using (literalTrue ?w1 (elements (getM ?state')))
    using (InvariantConsistent (getM state))
    unfolding InvariantConsistent-def

426
unfolding swapWatches-def
by (auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization)

thus ?thesis
using True
using Cons(1)[of ?state' clause ≠ newWl]
using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6)
using (∼ Some literal = getWatch1 state clause)
using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
using (literalTrue ?w1 (elements (getM ?state')))
using (uniq Wl')
by (auto simp add:Let-def)

next
case False
show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
case (Some l')
  hence l' el (nth (getF ?state') clause) ∨ literalFalse l' (elements (getM ?state'))
    using getNonWatchedUnfalsifiedLiteralSomeCharacterization
    by auto

  let ?state'' = setWatch2 clause l' ?state'

  from Cons(2)
  haveInvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
    using l' el (nth (getF ?state') clause);
  unfoldingInvariantWatchesEl-def
  unfolding setWatch2-def
  by auto
  moreover
  from Cons(4)
  haveInvariantConsistent (getM ?state'')
    unfolding setWatch2-def
    by simp
  moreover
  have getM ?state'' = getM state ∧
    getF ?state'' = getF state ∧
    getConflictFlag ?state'' = getConflictFlag state
    unfolding setWatch2-def
    by simp
  moreover
  have ∃ c. c ∈ set Wl' → Some literal = getWatch1 ?state'' c
    ∨ Some literal = getWatch2 ?state'' c
    using Cons(5)
using \langle \text{clause} \notin \text{set } W_l \rangle

unfolding setWatch2-def
by auto

moreover
have \neg \text{clauseFalse} \ (\text{nth} \ (\text{getF state}) \ \text{clause}) \ (\text{elements} \ (\text{getM state}))

using \langle l' \ el \ (\text{nth} \ (\text{getF ?state'}) \ \text{clause}) \rangle

using \langle \neg \text{literalFalse} \ l' \ (\text{elements} \ (\text{getM ?state'})) \rangle

using \langle \text{InvariantConsistent} \ (\text{getM state}) \rangle

unfolding InvariantConsistent-def
by \ (\text{auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistentCharacterization})

ultimately
show ?thesis
proof
(\cases literalFalse \ ?w1 \ (\text{elements} \ (\text{getM ?state'})))

case True
let ?state'' = ?state' [getConflictFlag := True, getConflict-Clause := clause]

from Cons(2)
have InvariantWatchesEl \ (getF ?state'') \ (getWatch1 ?state'') \ (getWatch2 ?state'')

unfolding InvariantWatchesEl-def
by auto

moreover
from Cons(4)
have InvariantConsistent \ (getM ?state'')

unfolding setWatch2-def
by simp

moreover
have \text{getM ?state''} = \text{getM state} \land
\text{getF ?state''} = \text{getF state} \land
\text{getSATFlag ?state''} = \text{getSATFlag state}

428
by simp
moreover
have ∀ c ∈ Wl' → Some literal = getWatch1 ?state''
c ∨ Some literal = getWatch2 ?state'' c
  using Cons(5)
  using (clause /∈ set Wl')
  unfolding setWatch2-def
by auto
moreover
have clauseFalse (nth (getF state) clause) (elements (getM state))
  using (l, l el (nth (getF ?state') clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM ?state')));
  using (literalFalse ?w1 (elements (getM ?state')));
  using (literalFalse ?w2 (elements (getM state)));
by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
ultimately
show ?thesis
  using Cons(1)[of ?state" clause ≠ newWl]
  using Cons(3) Cons(4) Cons(6)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (¬ Some literal = getWatch1 state clause)
  using (¬ literalTrue ?w1 (elements (getM ?state')));
  using None
  using (literalFalse ?w1 (elements (getM ?state')));
  using (uniq Wl')
by (auto simp add: Let-def)
next
case False
  let ?state"'' = setReason ?w1 clause (?state"''(getQ := (if ?w1 el (getQ ?state')) then (getQ ?state') else (getQ ?state') @ [?w1])))

from Cons(2)
have InvariantWatchesEl (getF ?state"') (getWatch1 ?state"')
  (getWatch2 ?state"')
unfolding InvariantWatchesEl-def
unfolding setReason-def
by auto
moreover
from Cons(4)
have InvariantConsistent (getM ?state"')
  unfolding setReason-def
by simp
moreover
have getM ?state"'' = getM state ∧
  getF ?state"'' = getF state ∧
  getSATFlag ?state"'' = getSATFlag state
  unfolding setReason-def
by simp
moreover
have \( \forall c. c \in \text{set } Wl' \rightarrow \text{Some literal } = \text{getWatch1 } \text{?state'' } c \)
  using Cons(5)
  using (clause \notin \text{set } Wl')
  unfolding setReason-def
by auto
moreover
have \( \neg \text{clauseFalse } (\text{nth } (\text{getF state}) \text{ clause}) \text{ (elements } (\text{getM state})) \)
  using (?w1 el (\text{nth } (\text{getF state}) \text{ clause}))
  using (\neg \text{literalFalse } ?w1 \text{ (elements } (\text{getM } \text{?state'})))
  using (InvariantConsistent (\text{getM state}))
  unfolding InvariantConsistent-def
by (auto simp add: clauseFalseIffAllLiteralsAreFalse inconsistantCharacterization)
ultimately
show ?thesis
  using Cons(1)[of ?state" clause # newWl]
  using Cons(3) Cons(4) Cons(6)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (\neg \text{Some literal } = \text{getWatch1 state clause})
  using (\neg \text{literalTrue } ?w1 \text{ (elements } (\text{getM } \text{?state'})))
  using None
  using (\neg \text{literalFalse } ?w1 \text{ (elements } (\text{getM } \text{?state'})))
  using (uniq Wl')
  apply (simp add: Let-def)
  unfolding setReason-def
by auto
qed
qed
qed
qed

**lemma** NotifyWatchesLoopQEffect:
**fixes** literal :: Literal and Wl :: nat list and newWl :: nat list and state :: State
**assumes**
(\text{getM state}) = M @ [(opposite literal, decision)] \text{ and } \text{InvariantWatchesEl } (\text{getF state}) \text{ (getWatch1 state)} \text{ (getWatch2 state)} \text{ and } \text{InvariantWatchesDiffer } (\text{getF state}) \text{ (getWatch1 state)} \text{ (getWatch2 state)} \text{ and } \forall (c::nat). c \in \text{set } Wl \rightarrow 0 \leq c \land c < \text{length } (\text{getF state}) \text{ and } \text{InvariantConsistent } (\text{getM state}) \text{ and}
∀ (c::nat), c ∈ set Wl → Some literal = (getWatch1 state c) ∨ Some literal = (getWatch2 state c)
and uniq Wl and InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) M

shows
let state' = notifyWatches-loop literal Wl newWl state in
((∀ l, l ∈ (set (getQ state') − set (getQ state)) →
   (∃ clause. (clause el (getF state) ∧
       literal el clause ∧
       (isUnitClause clause l (elements (getM state)))))) ∧
(∀ clause. clause ∈ set Wl →
   (∀ l, (isUnitClause (nth (getF state) clause) l (elements (getM state)))))) →
   l ∈ (set (getQ state')))))
(is let state' = notifyWatches-loop literal Wl newWl state in (?Cond1 state' state ∧ ?Cond2 WI state' state))

using assms

proof (induct Wl arbitrary: newWl state)
case Nil
  thus ?case
  by simp

next
  case (Cons clause Wl')

  from ‹uniq ‹clause # Wl'››
  have uniq Wl' and clause /∈ set Wl'
    by (auto simp add: uniqAppendIff)

  from (∀ (c::nat), c ∈ set (clause # Wl') → 0 ≤ c ∧ c < length (getF state)):
  have 0 ≤ clause clause < length (getF state)
    by auto
  then obtain wa::Literal and wb::Literal
    where getWatch1 state clause = Some wa and getWatch2 state clause = Some wb
  using Cons
  unfolding InvariantWatchesEl-def
  by auto

  from ⟨0 ≤ clause⟩ ⟨clause < length (getF state)⟩:
  have (nth (getF state) clause) el (getF state)
    by simp

  show ?case
  proof (cases Some literal = getWatch1 state clause)
    case True
    let ?state' = swapWatch clause state
    let ?w1 = wb
have \( \text{getWatch1 } \text{?state'} \text{ clause } = \text{Some } ?w1 \)
using \(\text{getWatch2 state clause } = \text{Some } \text{wb}\)
unfolding \(\text{swapWatches-def}\)
by auto
let \(?w2 = \text{wa}\)

have \(\text{getWatch2 } \text{?state'} \text{ clause } = \text{Some } ?w2\)
using \(\text{getWatch1 state clause } = \text{Some } \text{wa}\)
unfolding \(\text{swapWatches-def}\)
by auto

have \(?w2 = \text{literal}\)
using \( (\text{Some literal } = \text{getWatch1 state clause})\)
using \( (\text{getWatch2 } \text{?state'} \text{ clause } = \text{Some } ?w2)\)
unfolding \(\text{swapWatches-def}\)
by simp

hence \(\text{literalFalse } ?w2 \ (\text{elements } (\text{getM state}))\)
using \( (\text{getM state}) = \text{M } @ \[(\text{opposite literal, decision})]\) 
by simp

from \((\text{InvariantWatchesEl} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}))\)

have \(?w1 \ \text{el} \ (\text{nth } (\text{getF state}) \text{ clause}) \ ?w2 \ \text{el} \ (\text{nth } (\text{getF state}) \text{ clause})\)
using \( (\text{getWatch1 } \text{?state'} \text{ clause } = \text{Some } ?w1)\)
using \( (\text{getWatch2 } \text{?state'} \text{ clause } = \text{Some } ?w2)\)
using \( (\text{clause} < \text{length } (\text{getF state}))\)
unfolding \(\text{InvariantWatchesEl-def}\)
unfolding \(\text{swapWatches-def}\)
by auto

from \((\text{InvariantWatchesDiffer} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}))\)

have \(?w1 \neq ?w2\)
using \( (\text{getWatch1 } \text{?state'} \text{ clause } = \text{Some } ?w1)\)
using \( (\text{getWatch2 } \text{?state'} \text{ clause } = \text{Some } ?w2)\)
using \( (\text{clause} < \text{length } (\text{getF state}))\)
unfolding \(\text{InvariantWatchesDiffer-def}\)
unfolding \(\text{swapWatches-def}\)
by auto

show \(?\text{thesis}\)
proof (cases \(\text{literalTrue } ?w1 \ (\text{elements } (\text{getM } \text{?state'}))\))
case \(\text{True}\)
from \(\text{Cons}(3)\)
have \(\text{InvariantWatchesEl} \ (\text{getF } \text{?state'}) \ (\text{getWatch1 } \text{?state'}) \ (\text{getWatch2 } \text{?state'})\)
unfolding \(\text{InvariantWatchesEl-def}\)
unfolding swap Watches-def by auto
moreover from Cons(4)
have Invariant Watches Differ (getF ?state') (get Watch1 ?state')
  (get Watch2 ?state')
unfolding Invariant Watches Differ-def
unfolding swap Watches-def by auto
moreover have getF ?state' = getF state ∧
  getM ?state' = getM state ∧
  getQ ?state' = getQ state ∧
  get Conflict Flag ?state' = get Conflict Flag state

unfolding swap Watches-def by simp
moreover have ∀ c. c ∈ set Wl' — Some literal = get Watch1 ?state' c ∨
  Some literal = get Watch2 ?state' c
  using Cons(7)
  unfolding swap Watches-def by auto
moreover have Invariant Watch Characterization (getF ?state') (get Watch1
  ?state') (get Watch2 ?state') M
  using Cons(9)
  unfolding swap Watches-def
  unfolding Invariant Watch Characterization-def by auto
moreover have ¬ (∃ l. is Unit Clause (nth (getF state) clause) l (elements
  (getM state)))
  using (?w1 el (nth (getF state) clause))
  using (literal True ?w1 (elements (getM ?state')))
  using (Invariant Consistent (getM state))
  unfolding Invariant Consistent-def
  unfolding swap Watches-def
  by (auto simp add: is Unit Clause-def inconsistent Characterization)
ultimately show ?thesis
using Cons(1)[of ?state' clause ≠ new Wl]
using Cons(2) Cons(5) Cons(6)
using (get Watch1 ?state' clause = Some ?w1)
using (get Watch2 ?state' clause = Some ?w2)
using (Some literal = get Watch1 state clause)
using (literal True ?w1 (elements (getM ?state')))
using (uniq Wl)
by (simp add:Lem-def)
next
case False
  show ?thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
  case (Some l')
    hence l' el (nth (getF ?state') clause) ¬ literalFalse l' (elements (getM ?state')) l' ≠ ?w1 l' ≠ ?w2
      using getNonWatchedUnfalsifiedLiteralSomeCharacterization
    by auto

  let ?state'' = setWatch2 clause l' ?state'

from Cons(3)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    using (l' ≠ ?w1)
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    by auto
moreover
from Cons(4)
  have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    using (l' ≠ ?w1)
    unfolding InvariantWatchesDiffer-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    by auto
moreover
from Cons(6)
  have InvariantConsistent (getM ?state'')
    unfolding setWatch2-def
    unfolding swapWatches-def
    by simp
moreover
  have getM ?state'' = getM state ∧
    getF ?state'' = getF state ∧
    getQ ?state'' = getQ state ∧
    getConflictFlag ?state'' = getConflictFlag state
    unfolding swapWatches-def
    unfolding setWatch2-def
    by simp
moreover
  have ∀ c. c ∈ set Wl' → Some literal = getWatch1 ?state'' c
\[ \text{Some literal} = \text{getWatch2 \ state''} c \]

using Cons(7)
using ⟨\text{clause} \notin \text{set Wl}'\rangle
unfolding swapWatches-def
unfolding setWatch2-def
by auto

moreover
have InvariantWatchCharacterization (getF \ state") (getWatch1 \ state'') (getWatch2 \ state'') M

proof -
{ 
  fix \(c::\text{nat}\) and \(ww1::\text{Literal}\) and \(ww2::\text{Literal}\)
  assume \(a: 0 \leq c \land c < \text{length (getF \ state'')} \land \text{Some \(ww1\)} = (\text{getWatch1 \ state''} c) \land \text{Some \(ww2\)} = (\text{getWatch2 \ state''} c)\)
  assume \(b: \text{literalFalse \(ww1\)} (\text{elements M})\)

  have (\(\exists l. \text{el \((getF \ state'') \! c\)} \land \text{literalTrue \(l\)} (\text{elements M}) \land \text{elementLevel \(l\) \(M\)} \leq \text{elementLevel \((\text{opposite \(ww1\)} \ M)\)} \lor
  (\(\forall l. \text{el \((getF \ state'') \! c\)} \land l \neq \text{ww1} \land l \neq \text{ww2} \rightarrow \text{literalFalse \(l\)} (\text{elements M}) \land \text{elementLevel \((\text{opposite \(l\)} \ M)\)}) \land \text{elementLevel \((\text{opposite \(ww1\)} \ M)\)} \leq \text{elementLevel \((\text{opposite \(ww1\)} \ M)\)} \)

  proof (cases \(c = \text{clause}\))
  case False
  thus \(?\text{thesis}\)
  using \(a\) and \(b\)
  using Cons(9)
  unfolding InvariantWatchCharacterization-def
  unfolding watchCharacterizationCondition-def
  unfolding swapWatches-def
  unfolding setWatch2-def
  by simp

  next
  case True
  with \(a\)
  have \(ww1 = ?w1\) and \(ww2 = l'\)
  using ⟨\text{getWatch1 \ state' clause} = \text{Some \(ww1\)}⟩
  using ⟨\text{getWatch2 \ state' clause} = \text{Some \(ww2\)}⟩[THEN sym]

  unfolding setWatch2-def
  unfolding swapWatches-def
  by auto

  have \(\lnot (\forall l. \text{el \((getF state'! clause) \land l \neq \text{ww1} \land l \neq \text{ww2} \rightarrow \text{literalFalse \(l\)} (\text{elements M}))\)}
  using Cons(2)
  using \(l' \neq \text{ww1}\) and \(l' \neq \text{ww2}\) \(l'\) \(\text{el \((\text{nth \((getF \ state'\)})\)} \)

  proof (\(\forall l. \text{el \((getF state'\)} \land l \neq \text{ww1} \land l \neq \text{ww2} \rightarrow \text{literalFalse \(l\)} (\text{elements \((\text{getM \ state'})\)})\))
  using \(a\) and \(b\)

435
using \( c = \text{clause} \)
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
have \( \exists l. l \in (\text{getF state} ! \text{clause}) \land \text{literalTrue} l \) (elements \( M \)) \land 
\( \text{elementLevel} l M \leq \text{elementLevel} (\text{opposite} ?w1) M \) \lor 
\( \forall l. l \in (\text{getF state} ! \text{clause}) \land l \neq ?w1 \land l \neq ?w2 \rightarrow \text{literalFalse} l \) (elements \( M \))
using Cons(9)
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
using \( \langle \text{clause} < \text{length} (\text{getF state}) \rangle \)
using \( \langle \text{getWatch1} ?\text{state'} \text{clause} = \text{Some} ?w1 \rangle \) [THEN sym]
using \( \langle \text{getWatch2} \text{state'} \text{clause} = \text{Some} ?w2 \rangle \) [THEN sym]
using \( \langle \text{literalFalse} \text{ww1} \rangle \) (elements \( M \))
using \( \langle \text{ww1} = ?w1 \rangle \)
unfolding setWatch2-def
unfolding swapWatches-def
by auto
ultimately
show \( ?\text{thesis} \)
using \( \langle \text{ww1} = ?w1 \rangle \)
using \( \langle c = \text{clause} \rangle \)
unfolding setWatch2-def
unfolding swapWatches-def
by auto
qed
moreover
fix \( c::\text{nat} \) and \( \text{ww1}::\text{Literal} \) and \( \text{ww2}::\text{Literal} \)
assume \( a: 0 \leq c \land c < \text{length} (\text{getF} ?\text{state''}) \land \text{Some} \text{ww1} = (\text{getWatch1} ?\text{state''} c) \land \text{Some} \text{ww2} = (\text{getWatch2} ?\text{state''} c) \)
assume \( b: \text{literalFalse} \text{ww2} \) (elements \( M \))

have \( \exists l. l \in ((\text{getF} ?\text{state''}) ! c) \land \text{literalTrue} l \) (elements \( M \)) \land \text{elementLevel} l M \leq \text{elementLevel} (\text{opposite} \text{ww2}) M \land 
\( \forall l. l \in ((\text{getF} ?\text{state''}) ! c) \land l \neq \text{ww1} \land l \neq \text{ww2} \rightarrow \) 
\( \text{literalFalse} l \) (elements \( M \)) \land \text{elementLevel} (\text{opposite} \text{ww1} M) \land \text{elementLevel} (\text{opposite} \text{ww2} M)
proof (cases \( c = \text{clause} \))
case False
thus \( ?\text{thesis} \)
using \( a \) and \( b \)
using Cons(9)
unfolding \textit{InvariantWatchCharacterization-def}
unfolding \textit{watchCharacterizationCondition-def}
unfolding \textit{swapWatches-def}
unfolding \textit{setWatch2-def}

by auto

next

\begin{itemize}
  \item \textbf{case} True
  \item with a
  \begin{itemize}
    \item have $\text{ww1} = \text{w1}$ and $\text{ww2} = \text{l'}$
      \begin{itemize}
        \item using \langle \text{getWatch1 \ ?state' clause = Some \ ?w1} \rangle
        \item using \langle \text{getWatch2 \ ?state' clause = Some \ ?w2} \rangle
      \end{itemize}
    \end{itemize}
  \end{itemize}

\textbf{sym}

unfolding \textit{setWatch2-def}
unfolding \textit{swapWatches-def}

by auto

with $\sim \text{literalFalse \ l'} \ (\text{elements (getM \ ?state')})$; b

Cons(2)

have False

unfolding \textit{swapWatches-def}

by simp

thus \textit{?thesis}

by simp

qed

\}

ultimately

\begin{itemize}
  \item \textbf{show} \textit{?thesis}
  \item unfolding \textit{InvariantWatchCharacterization-def}
  \item unfolding \textit{watchCharacterizationCondition-def}
  \item by blast
\end{itemize}

qed

moreover

\begin{itemize}
  \item have $\sim (\exists \ l. \ \text{isUnitClause (nth (getF \ state) clause) \ l (elements (getM \ state)})$)
  \end{itemize}

\textbf{proof} --

\begin{itemize}
  \item assume $\sim \textit{?thesis}$
  \item then obtain l
    \begin{itemize}
      \item where isUnitClause (nth (getF \ state) clause) \ l (elements (getM \ state))
      \item by auto
    \end{itemize}
    \begin{itemize}
      \item with $\ l' \ el (nth (getF \ ?state') \ clause); (\sim \text{literalFalse \ l'})$
      \item (elements (getM \ ?state')))
      \item have $l = l'$
      \item unfolding \textit{isUnitClause-def}
      \item unfolding \textit{swapWatches-def}
      \item by auto
    \end{itemize}
    \begin{itemize}
      \item with $l' \neq \text{w1}$ have
      \item literalFalse \ w1 \ (elements (getM \ ?state'))
    \end{itemize}
\end{itemize}

437
using \langle \text{isUnitClause} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\text{elements} \ (\text{getM} \ \text{state})) \rangle
using \langle \text{\?w1 el} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \rangle
\text{unfolding isUnitClause-def}
\text{unfolding swapWatches-def}
\text{by simp}
\text{with} \langle \text{\?w1 \ \neq \ ?w2} \ : \ ?w2 = \text{literal} \rangle
Cons(2)
\text{have} \text{literalFalse} \ ?w1 \ (\text{elements} \ M)
\text{unfolding swapWatches-def}
\text{by simp}

\text{from} \langle \text{isUnitClause} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\text{elements} \ (\text{getM} \ \text{state})) \rangle
Cons(6)
\text{have} \neg \ (\exists \ l. \ (l \ el \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ (\text{getM} \ \text{state}))))
\text{using containsTrueNotUnit[of -} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ \text{elements} \ (\text{getM} \ \text{state})]
\text{unfolding InvariantConsistent-def}
\text{by auto}

\text{from} \langle \text{InvariantWatchCharacterization} \ (\text{getF} \ \text{state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \ M \rangle
\langle \text{clause < length} \ (\text{getF} \ \text{state}) \rangle
\langle \text{\text{literalFalse} ?w1} \ (\text{elements} \ M) \rangle
\langle \text{getWatch1 \ ?state’ clause} = \text{Some ?w1} \ \rangle \text{THEN sym}
\langle \text{getWatch2 \ ?state’ clause} = \text{Some ?w2} \ \rangle \text{THEN sym}
\text{have} \ (\exists \ l. \ l \ el \ (\text{getF} \ \text{state} \ ! \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ M) \ \wedge \ \text{elementLevel} \ l \ M \ \leq \ \text{elementLevel} \ (\text{opposite} \ ?w1) \ M) \ \lor
\ (\forall \ l. \ l \ el \ (\text{getF} \ \text{state} \ ! \ \text{clause}) \ \wedge \ l \ \neq \ ?w1 \ \wedge \ l \ \neq \ ?w2 \ \rightarrow \ \text{literalFalse} \ l \ (\text{elements} \ M))
\text{unfolding InvariantWatchCharacterization-def}
\text{unfolding watchCharacterizationCondition-def}
\text{unfolding swapWatches-def}
\text{by auto}
\text{with} \langle \neg \ (\exists \ l. \ (l \ el \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ \wedge \ \text{literalTrue} \ l \ (\text{elements} \ (\text{getM} \ \text{state})))) \rangle
Cons(2)
\text{have} \ (\forall \ l. \ l \ el \ (\text{getF} \ \text{state} \ ! \ \text{clause}) \ \wedge \ l \ \neq \ ?w1 \ \wedge \ l \ \neq \ ?w2 \ \rightarrow \ \text{literalFalse} \ l \ (\text{elements} \ M))
\text{by auto}
\text{with} \langle \ l’ \ el \ (\text{getF} \ ?state’ ! \ \text{clause}) \ \langle l’ \ \neq \ ?w1 \ \rangle \ \langle l’ \ \neq \ ?w2 \ \rangle \ \neg \ \text{literalFalse} \ l’ \ (\text{elements} \ (\text{getM} \ ?state’)) \rangle
Cons(2)
\text{have} \text{False}
\text{unfolding swapWatches-def}
\text{by simp}
}

438
thus \( ?\text{thesis} \)
by auto
qed
ultimately
show \( ?\text{thesis} \)
using \( \text{Cons}(1)\)[of \( ?\text{state}'' \) new\( Wl \)]
using \( \text{Cons}(2) \) \( \text{Cons}(5) \) \( \text{Cons}(6) \)
using \( \langle \text{getWatch1} \ ?\text{state'} \ \text{clause} = \text{Some} \ ?w1 \rangle \)
using \( \langle \text{getWatch2} \ ?\text{state'} \ \text{clause} = \text{Some} \ ?w2 \rangle \)
using \( \langle \text{Some literal} = \text{getWatch1} \ ?\text{state} \ \text{clause} \rangle \)
using \( \langle \neg \text{literalTrue} \ ?w1 \ \langle \text{elements} \ (\text{getM} \ ?\text{state'}) \rangle \rangle \)
using \( \langle \text{uniq} \ Wl \rangle \)
using \( \langle \text{getWatch1} \ ?\text{state} \ \text{clause} = \text{Some} \ ?w1 \rangle \)
by (simp add: Let-def)
next
case None
hence \( \forall \ l. \ l \in \text{cl} \ \langle \text{nth} \ (\text{getF} \ ?\text{state'}) \ \text{clause} \rangle \ \land \ ?w1 \ \land \ ?w2 \ 
using \text{getNonWatchedUnfalsifiedLiteralNoneCharacterization}
by simp
show \( ?\text{thesis} \)
proof (cases literalFalse \( ?w1 \ \langle \text{elements} \ (\text{getM} \ ?\text{state'}) \rangle \))
case True
let \( ?\text{state}'' = ?\text{state}'[\langle \text{getConflictFlag} := \text{True}, \text{getConflictClause} := \text{clause} \rangle] \)
from \( \text{Cons}(3) \)
have \( \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state''}) \ (\text{getWatch1} \ ?\text{state''}) \)
(\text{getWatch2} \ ?\text{state''})
unfolding \( \text{InvariantWatchesEl-def} \)
unfolding \( \text{swapWatches-def} \)
by auto
moreover
from \( \text{Cons}(4) \)
have \( \text{InvariantWatchesDiffer} \ (\text{getF} \ ?\text{state''}) \ (\text{getWatch1} \ ?\text{state''}) \)
(\text{getWatch2} \ ?\text{state''})
unfolding \( \text{InvariantWatchesDiffer-def} \)
unfolding \( \text{swapWatches-def} \)
by auto
moreover
from \( \text{Cons}(6) \)
have \( \text{InvariantConsistent} \ (\text{getM} \ ?\text{state''}) \)
unfolding \( \text{swapWatches-def} \)
by simp
moreover
have \( \text{getM} \ ?\text{state''} = \text{getM} \ ?\text{state} \ \land \ ^\ 
\text{getF} \ ?\text{state''} = \text{getF} \ ?\text{state} \ \land \ ^\ 
\text{getQ} \ ?\text{state''} = \text{getQ} \ ?\text{state} \ \land \ ^\ 
\text{getSATFlag} \ ?\text{state''} = \text{getSATFlag} \ ?\text{state} \)

439
unfolding swapWatches-def
  by simp
moreover
  have $\forall c. c \in \text{set } Wl' \rightarrow \text{Some literal } = \text{getWatch1 } \text{?state'}$ 
  $c \lor \text{Some literal } = \text{getWatch2 } \text{?state'}$ 
  using Cons(7)
  using $\text{clause } \notin \text{set } Wl'$
  unfolding swapWatches-def
  by auto
moreover
  have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') M
  using Cons(9)
  unfolding swapWatches-def
  unfolding InvariantWatchCharacterization-def
  by auto
moreover
  have clauseFalse (nth (getF ?state) clause) (elements (getM ?state))
    using $\forall l. l \in \text{elements (getM ?state')} \land l \neq w1 \land l \neq w2 \rightarrow \text{literalFalse } l$ (elements (getM ?state'))
    using (literalFalse w1 (elements (getM ?state')))
    using (literalFalse w2 (elements (getM ?state')))
    unfolding swapWatches-def
    by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
hence $\neg (\exists l. \text{isUnitClause (nth (getF ?state) clause) l (elements (getM ?state'))})$
  unfolding isUnitClause-def
  by (simp add: clauseFalseIffAllLiteralsAreFalse)
ultimately
  show ?thesis
  using Cons(1) (of ?state'' clause # newWl)
  using Cons(2) Cons(5) Cons(6)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using None
  using (literalFalse w1 (elements (getM ?state')))
  using (uniq Wl')
  by (simp add: Let-def)
next
  case False
  let ?state'' = setReason ?w1 clause (?state'(getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])))
  from Cons(3)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
unfolding setReason-def
by auto
moreover
from Cons(4)
  have InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'')
    unfolding InvariantWatchesDiffer-def
    unfolding swapWatches-def
    unfolding setReason-def
    by auto
moreover
from Cons(6)
  have InvariantConsistent (getM ?state'')
    unfolding swapWatches-def
    unfolding setReason-def
    by simp
moreover
  have getM ?state'' = getM state ∧
    getF ?state'' = getF state ∧
    getSATFlag ?state'' = getSATFlag state ∧
    getQ ?state'' = (if ?w1 el (getQ state) then (getQ state) else
    (getQ state @ ?w1))
    unfolding swapWatches-def
    unfolding setReason-def
    by simp
moreover
  have ∀ c. c ∈ set Wl' → Some literal = getWatch1 ?state''
c ∨ Some literal = getWatch2 ?state'' c
    using Cons(7)
    using (clause ∉ set Wl')
    unfolding swapWatches-def
    unfolding setReason-def
    by auto
moreover
  have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') M
    using Cons(9)
    unfolding swapWatches-def
    unfolding setReason-def
    unfolding InvariantWatchCharacterization-def
    by auto
ultimately
  have let state' = notifyWatches-loop literal Wl' (clause # newWl) ?state'' in
    ?Cond1 state' ?state'' ∧ ?Cond2 Wl' state' ?state''
    using Cons(1) of ?state'' clause # newWl
    using Cons(2) Cons(5)

441
using ⟨uniq Wl⟩
by (simp add: Let-def)
moreover
have notifyWatches-loop literal Wl' (clause # newWl) ?state'' = notifyWatches-loop literal (clause # Wl') newWl state
using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
using (Some literal = getWatch1 state clause)
using (∼ literalTrue ?w1 (elements (getM ?state')))
using None
using (∼ literalFalse ?w1 (elements (getM ?state')))
by (simp add: Let-def)
ultimately
have let state' = notifyWatches-loop literal (clause # Wl') newWl state in
?Cond1 state' ?state'' ∧ ?Cond2 Wl' state ?state''
by simp

have isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))
using (∀ l. l el (nth (getF ?state') clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM ?state')))
using (?w1 el (nth (getF state) clause))
using (?w2 el (nth (getF state) clause))
using (literalFalse ?w2 (elements (getM state)))
using (∼ literalFalse ?w1 (elements (getM ?state')))
using (∼ literalTrue ?w1 (elements (getM ?state')))
unfolding swapWatches-def
unfolding isUnitClause-def
by auto

show ?thesis
proof−
{
  fix l::Literal
  assume let state' = notifyWatches-loop literal (clause # Wl') newWl state in
  l ∈ set (getQ state') – set (getQ state)
  have ∃ clause. clause el (getF state) ∧ literal el clause ∧ isUnitClause clause l (elements (getM state))
  proof (cases l ≠ ?w1)
  case True
  hence let state' = notifyWatches-loop literal (clause # Wl') newWl state in
  l ∈ set (getQ state') – set (getQ ?state'')
  using (getWl state' = notifyWatches-loop literal (clause # Wl') newWl state in
  l ∈ set (getQ state') – set (getQ state)
  unfolding setReason-def

442
unfolding swapWatches-def
by (simp add: Let-def)
with (!state' = notifyWatches-loop literal (clause # Wl'))
newWl state in

?Cond1 state' ?state'' ∧ ?Cond2 Wl' state' ?state''

show thesis
unfolding setReason-def
unfolding swapWatches-def
by (simp add: Let-def del: notifyWatches-loop.simps)

next

case False
thus thesis
using ⟨(nth (getF state) clause) el (getF state):

(?w2 = literal)

(?w2 el (nth (getF state) clause):

isUnitClause (nth (getF state) clause) ?w1 (elements (getM state)))

by (auto simp add: Let-def)

qed

hence let state' = notifyWatches-loop literal (clause # Wl')
newWl state in

!state' state
by simp

moreover

{ fix c

assume c ∈ set (clause # Wl')

have let state' = notifyWatches-loop literal (clause # Wl')
newWl state in

! l. isUnitClause (nth (getF state) c) l (elements (getM state)) → l ∈ set (getQ state')

proof (cases c = clause)

  case True

  { fix l::Literal
    assume isUnitClause (nth (getF state) c) l (elements (getM state))

    with isUnitClause (nth (getF state) clause) ?w1 (elements (getM state)): (c = clause)

    have l = ?w1

    unfolding isUnitClause-def

    by auto

    have isPrefix (getQ ?state'') (getQ (notifyWatches-loop literal Wl' (clause # newWl) ?state''))
    using !InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
        (getWatch2 ?state'')

    unfolding notifyWatchesLoopPreservedVariables[of ?state''

    Wl' literal clause # newWl]
using Cons(5)
unfolding swapWatches-def
unfolding setReason-def
by (simp add: Let-def)
hence set (getQ ?state'') ⊆ set (getQ (notifyWatches-loop literal Wi' (clause # newWi) ?state''))
using prefixIsSubset[of getQ ?state'' getQ (notifyWatches-loop literal Wi' (clause # newWi) ?state'')]
by auto
hence l ∈ set (getQ (notifyWatches-loop literal Wi' (clause # newWi) ?state''))
using (l = ?w1)
unfolding swapWatches-def
unfolding setReason-def
by auto

thus ?thesis
using (notifyWatches-loop literal Wi' (clause # newWi) ?state'') = notifyWatches-loop literal (clause # Wi') newWi state
by (simp add: Let-def)

next
case False
hence c ∈ set Wi'
using (c ∈ set (clause # Wi'))
by simp

{ fix l::Literal
assume isUnitClause (nth (getF state) c) l (elements (getM state))
hence isUnitClause (nth (getF ?state'') c) l (elements (getM ?state''))
unfolding setReason-def
unfolding swapWatches-def
by simp
with (let state' = notifyWatches-loop literal (clause # Wi') newWi state in
?Cond1 state' ?state'' ∧ ?Cond2 Wi' state' ?state'';
 c ∈ set Wi')
have let state' = notifyWatches-loop literal (clause # Wi') newWi state in l ∈ set (getQ state')
by (simp add: Let-def)
}
thus ?thesis
by (simp add: Let-def)

qed

} hence ?Cond2 (clause # Wi') (notifyWatches-loop literal (clause # Wi') newWi state) state
by (simp add: Let-def)
ultimately

show ?thesis
  by (simp add: Let-def)

qed

next

  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using ⟨getWatch1 state clause = Some wa⟩
  unfolding swapWatches-def
  by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using ⟨getWatch2 state clause = Some wb⟩
  unfolding swapWatches-def
  by auto

from (¬ Some literal = getWatch1 state clause)
  (∀ (c::nat). c ∈ set (clause ≠ Wl) ─→ Some literal = (getWatch1 state c) ∨ Some literal = (getWatch2 state c))
  have Some literal = getWatch2 state clause
    by auto
  hence ?w2 = literal
    using ⟨getWatch2 ?state' clause = Some ?w2⟩
    by simp
  hence literalFalse ?w2 (elements (getM state))
    using Cons(2)
    by simp

from (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state))
  have ?w1 el (nth (getF state) clause) ?w2 el (nth (getF state) clause)
    using ⟨getWatch1 ?state' clause = Some ?w1⟩
    using ⟨getWatch2 ?state' clause = Some ?w2⟩
    using ⟨clause < length (getF state)⟩
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
  by auto

from (InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state))
  have ?w1 ≠ ?w2
    using ⟨getWatch1 ?state' clause = Some ?w1⟩
using \( \langle \text{getWatch2} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w2 \rangle \)
using \( \langle \text{clause} < \text{length} \ (\text{getF} \ \text{state}) \rangle \)
unfolding InvariantWatchesDiffer-def
unfolding swapWatches-def
by auto

show \( \text{?thesis} \)
proof (cases \( \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \))
case True
have \( \neg (\exists \ l. \ \text{isUnitClause} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \ l \ (\text{elements} \ (\text{getM} \ \text{state}'))) \)
using \( \langle \ ?w1 \ \text{el} \ (\text{nth} \ (\text{getF} \ \text{state}) \ \text{clause}) \rangle \)
using \( \langle \ \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \rangle \)
using InvariantConsistent (getM state)
unfolding InvariantConsistent-def
by (auto simp add: isUnitClause-def inconsistentCharacterization)
thus \( \text{?thesis} \)
using True
using Cons(1)[of \( \text{?state}' \ \text{clause} \neq \text{newWl} \)]
using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7) Cons(8) Cons(9)
using \( \neg \ \text{Some literal} = \text{getWatch1} \ \text{state} \ \text{clause} \)
using \( \text{getWatch1} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w1 \)
using \( \text{getWatch2} \ ?\text{state}' \ \text{clause} = \text{Some} \ ?w2 \)
using \( \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)
using uniq Wl'
by (simp add: Let-def)
next
case False
show \( \text{?thesis} \)
proof (cases getNonWatchedUnfalsifiedLiteral (\text{nth} \ (\text{getF} ?\text{state}') \ \text{clause}) \ ?w1 \ ?w2 \ (\text{getM} ?\text{state}'))
case (Some l')
hence \( l' \ \text{el} \ (\text{nth} \ (\text{getF} ?\text{state}') \ \text{clause}) \neg \text{literalFalse} \ l' \ (\text{elements} \ (\text{getM} ?\text{state}')) \ l' \neq \ ?w1 \ l' \neq \ ?w2 \)
using getNonWatchedUnfalsifiedLiteralSomeCharacterization
by auto

let \( \text{?state}'' = \text{setWatch2} \ \text{clause} \ l' \ ?\text{state}' \)

from Cons(3)
have InvariantWatchesEl (\text{getF} ?\text{state}'') (\text{getWatch1} ?\text{state}'') (\text{getWatch2} ?\text{state}'')
using \( \langle \ l' \ \text{el} \ (\text{nth} \ (\text{getF} ?\text{state}') \ \text{clause}) \rangle \)
unfolding InvariantWatchesEl-def
unfolding setWatch2-def
by auto
moreover
from Cons(4)
have InvariantWatchesDiffer (getF ?state") (getWatch1 ?state")
  (getWatch2 ?state")
  using (l' ≠ ?w1)
  using (getWatch1 ?state’ clause = Some ?w1)
  using (getWatch2 ?state’ clause = Some ?w2)
  unfolding InvariantWatchesDiffer-def
  unfolding setWatch2-def
  by auto
moreover
from Cons(6)
have InvariantConsistent (getM ?state")
  unfolding setWatch2-def
  by simp
moreover
have getM ?state" = getM state ∧
  getF ?state" = getF state ∧
  getQ ?state" = getQ state ∧
  getConflictFlag ?state" = getConflictFlag state
  unfolding setWatch2-def
  by simp
moreover
have ∀ c. c ∈ set Wl' −→ Some literal = getWatch1 ?state" c
  ∨ Some literal = getWatch2 ?state" c
  using Cons(7)
  using (clause /∈ set Wl'
  unfolding setWatch2-def
  by auto
moreover
have InvariantWatchCharacterization (getF ?state") (getWatch1
  ?state") (getWatch2 ?state") M
proof −
  { fix c::nat and ww1::Literal and ww2::Literal
    assume a: 0 ≤ c ∧ c < length (getF ?state") ∧ Some ww1
    = (getWatch1 ?state" c) ∧ Some ww2 = (getWatch2 ?state" c)
    assume b: literalFalse ww1 (elements M)
    have (∃ l. l el ((getF ?state") ! c) ∧ literalTrue l (elements
      M) ∧ elementLevel l M ≤ elementLevel (opposite ww1) M) ∨
      (∀ l. l el (getF ?state” ! c) ∧ l ≠ ww1 ∧ l ≠ ww2 −→
        literalFalse l (elements M) ∧ elementLevel (opposite l)
      M ≤ elementLevel (opposite ww1) M)
    proof (cases c = clause)
      case False
      thus ?thesis
      using a and b
      using Cons(9)
      unfolding InvariantWatchCharacterization-def

447
next
next

have \( \forall l. l \in \text{elements} \ M \) \\
\rightarrow \text{literalFalse} \ l \ (\text{elements} \ M) \\
by \ auto
moreover
have \( \exists l. l \in \text{elements} \ M \) \\
\land \ \text{literalTrue} \ l \ (\text{elements} \ M) \\
\lor \\
(\forall l. l \in \text{elements} \ M \) \\
\rightarrow \text{literalFalse} \ l \ (\text{elements} \ M) \\
by \ auto
moreover
ultimately
show \ \text{thesis} \\
by \ auto
qed
{ fix c::nat and ww1::Literal and ww2::Literal
  assume a: 0 ≤ c ∧ c < length (getF ?state") ∧ Some ww1 = (getWatch1 ?state" c) ∧ Some ww2 = (getWatch2 ?state" c)
  assume b: literalFalse ww2 (elements M)

  have (∃ l el ((getF ?state") ! c) ∧ literalTrue l (elements M) ∧ elementLevel l M ≤ elementLevel (opposite ww2) M) ∨
    (∀ l l el (getF ?state" ! c) ∧ l ≠ ww1 ∧ l ≠ ww2 −→
      literalFalse l (elements M) ∧ elementLevel (opposite l) M ≤ elementLevel (opposite ww2) M)
  proof (cases c = clause)
    case False
    thus ?thesis
    using a and b
    using Cons(9)
    unfolding InvariantWatchCharacterization-def
    unfolding watchCharacterizationCondition-def
    unfolding setWatch2-def
    by auto
  next
  case True
  with a
  have ww1 = ?w1 and ww2 = l'
    using ⟨getWatch1 ?state' clause = Some ?w1⟩
    using ⟨getWatch2 ?state' clause = Some ?w2⟩[THEN sym]
  unfolding setWatch2-def
  by auto
  with (∼ literalFalse l' (elements (getM ?state'))): b
    Cons(2)
  have False
    unfolding setWatch2-def
    by simp
    thus ?thesis
    by simp
  qed
}
ultimately
show ?thesis
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
by blast
qed
moreover
have ∼ (∃ l. isUnitClause (nth (getF state) clause) l (elements (getM state)))
proof –
\{ 
assume \neg \ thesis
then obtain l
 where isUnitClause (nth (getF state) clause) l (elements (getM state))
  by auto
with (l' el (nth (getF ?state') clause); \neg literalFalse l'
 (elements (getM ?state')));
  have l = l'
  unfolding isUnitClause-def
  by auto
with (l' \neq ?w1) have
  literalFalse ?w1 (elements (getM ?state'))
  using isUnitClause (nth (getF state) clause) l (elements (getM state))
  using (?w1 el (nth (getF state) clause))
  unfolding isUnitClause-def
  by simp
with (?w1 \neq ?w2) (?w2 = literal)
Cons(2)
  have literalFalse ?w1 (elements M)
  by simp
from (isUnitClause (nth (getF state) clause) l (elements (getM state)));
Cons(6)
  have \neg (\exists l. (l el (nth (getF state) clause) \land literalTrue l
 (elements (getM state))))
  using containsTrueNotUnit[of - (nth (getF state) clause)
 elements (getM state)]
  unfolding InvariantConsistent-def
  by auto
from (InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) M)
 (\langle clause < length (getF state)\rangle)
 (\langle literalFalse ?w1 (elements M)\rangle)
 (\langle getWatch1 ?state' clause = Some ?w1 \rangle \THEN sym)
 (\langle getWatch2 ?state' clause = Some ?w2 \rangle \THEN sym)
 have (\exists l. l el (getF state ! clause) \land literalTrue l (elements M) \land elementLevel l M \leq elementLevel (opposite ?w1) M) \lor
 (\forall l. l el (getF state ! clause) \land l \neq ?w1 \land l \neq ?w2 \rightarrow
 literalFalse l (elements M))
  unfolding InvariantWatchCharacterization-def
 unfolding watchCharacterizationCondition-def
 unfolding swapWatches-def
  by auto
with (\neg (\exists l. (l el (nth (getF state) clause) \land literalTrue l
 (elements (getM state))))))
\}
Cons(2)

have \( \forall \, l. \, \text{l el (getF state ! clause)} \land l \neq \text{?w1} \land l \neq \text{?w2} \rightarrow \text{literalFalse l (elements M)} \)
by auto

with \( l' \, \text{el (getF ?state' ! clause)} \) \( l' \neq \text{?w1} \) \( l' \neq \text{?w2} \) \( \neg \text{literalFalse l' (elements (getM ?state'))} \)
Cons(2)

have \text{False}

unfolding \text{swapWatches-def}
by simp

\}

thus \text{?thesis}
by auto

qed

ultimately
show \text{?thesis}
proof
  (cases \text{literalFalse ?w1 (elements (getM ?state'))})
  case \text{True}
  let \( \text{?state'} = \text{?state'}[\text{getConflictFlag := True, getConflict-Clause := clause}] \)

from Cons(3)

have \text{InvariantWatchesEl (getF ?state')} (getWatch1 ?state') (getWatch2 ?state')

unfolding \text{InvariantWatchesEl-def}
by auto

moreover
from Cons(4)

have \text{InvariantWatchesDiffer (getF ?state')} (getWatch1 ?state') (getWatch2 ?state')

unfolding \text{InvariantWatchesDiffer-def}
by auto

moreover
from Cons(6)
have InvariantConsistent (getM ?state"")
  unfolding setWatch2-def
  by simp
moreover
have getM ?state"") = getM state ∧
  getF ?state" = getF state ∧
  getSATFlag ?state"") = getSATFlag state
  by simp
moreover
have ∀ c ∈ set Wl → Some literal = getWatch1 ?state"") c
  using Cons(7)
  using ⟨clause ∉ set Wl⟩
  unfolding setWatch2-def
  by auto
moreover
have InvariantWatchCharacterization (getF ?state"") (getWatch1 ?state"") (getWatch2 ?state"") M
  using Cons(9)
  unfolding InvariantWatchCharacterization-def
  by auto
moreover
have clauseFalse (nth (getF state) clause) (elements (getM state))
  using ∀ l. l el (nth (getF ?state') clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM ?state'))
  using (literalFalse ?w1 (elements (getM ?state')))
  using (literalFalse ?w2 (elements (getM state))
  unfolding swapWatches-def
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)

hence ¬ (∃ l. isUnitClause (nth (getF state) clause) l (elements (getM state)))
  unfolding isUnitClause-def
  by (simp add: clauseFalseIffAllLiteralsAreFalse)
ultimately
show ?thesis
  using Cons(1)[of ?state" clause ≠ newWI]
  using Cons(2) Cons(5) Cons(7)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (∀ Some literal = getWatch1 state clause)
  using (∀ literalTrue ?w1 (elements (getM ?state')))
  using None
  using (literalFalse ?w1 (elements (getM ?state')))
  using (uniq WI)
  by (simp add: Let-def)
next
  case False
let ?state'" = setReason ?w1 clause (?state'"(getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])))

from Cons(3)
have InvariantWatchesEl (getF ?state") (getWatch1 ?state")
  unfolding InvariantWatchesEl-def
  unfolding setReason-def
  by auto
moreover
from Cons(4)
have InvariantWatchesDiffer (getF ?state") (getWatch1 ?state") (getWatch2 ?state")
  unfolding InvariantWatchesDiffer-def
  unfolding setReason-def
  by auto
moreover
from Cons(6)
have InvariantConsistent (getM ?state")
  unfolding setReason-def
  by simp
moreover
have getM ?state" = getM state ∧
  getF ?state" = getF state ∧
  getSATFlag ?state" = getSATFlag state
  unfolding setReason-def
  by simp
moreover
have ∀ c. c ∈ set Wl' → Some literal = getWatch1 ?state" c
  using Cons(7)
  using ⟨clause ⋄ set Wl'⟩
  unfolding setReason-def
  by auto
moreover
have InvariantWatchCharacterization (getF ?state") (getWatch1 ?state") (getWatch2 ?state") M
  using Cons(9)
  unfolding InvariantWatchCharacterization-def
  unfolding setReason-def
  by auto
ultimately
have let state' = notifyWatches-loop literal Wl' (clause # newWl) ?state'' in
  ?Cond1 state' ?state" ∧ ?Cond2 Wl' state' ?state"
moreover

have notifyWatches-loop literal Wl' (clause # newWl) ?state'' = notifyWatches-loop literal (clause # Wl') newWl state

using (getWatch1 ?state' clause = Some ?w1)
using (getWatch2 ?state' clause = Some ?w2)
using (Some literal = getWatch1 state clause)
using (literalTrue ?w1 (elements (getM ?state')))
using None
using (literalFalse ?w1 (elements (getM ?state')))

ultimately

have let state' = notifyWatches-loop literal (clause # Wl') newWl state in

?Cond1 state' ?state'' ∧ ?Cond2 Wl' state' ?state''

by simp

have isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))

using (∀ l. l el (nth (getF ?state') clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM ?state')))

using (?w1 el (nth (getF state) clause))

using (?w2 el (nth (getF state) clause))

using (literalFalse ?w2 (elements (getM state)))

using (literalTrue ?w1 (elements (getM ?state')))
with ⟨let state′ = notifyWatches-loop literal (clause # Wl') newWl state in
?Cond1 state′ ?state'" ∧ ?Cond2 Wl' state' ?state"
⟩

show ?thesis

unfolding setReason-def

unfolding swapWatches-def

by (simp add: Let-def del: notifyWatches-loop.simps)

next

case False

thus ?thesis

using ⟨nth (getF state) clause⟩ el (getF state)
⟨isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))⟩
⟨?w2 = literal⟩
⟨?w2 el (nth (getF state) clause)⟩

by (auto simp add: Let-def)

qed

hence let state′ = notifyWatches-loop literal (clause # Wl')
newWl state in

?Cond1 state′ state

by simp

moreover

{ fix c

assume c ∈ set (clause # Wl')

have let state′ = notifyWatches-loop literal (clause # Wl')
newWl state in
∀ l. isUnitClause (nth (getF state) c) l (elements (getM state)) ⟷ l ∈ set (getQ state')

proof (cases c = clause)

case True

{ fix l::Literal

assume isUnitClause (nth (getF state) c) l (elements (getM state))

with ⟨isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))⟩ ⟨c = clause⟩

have l = ?w1

unfolding isUnitClause-def

by auto

have isPrefix (getQ ?state") (getQ (notifyWatches-loop literal Wl' (clause # newWl) ?state'"))

using ⟨InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state")⟩

using notifyWatchesLoopPreservedVariables[of ?state"
Wl' literal clause # newWl]

using Cons(5)

unfolding swapWatches-def

unfolding setReason-def

455
by (simp add: Let-def)

hence \( \text{set} (\text{getQ} \ ?\text{state}''') \subseteq \text{set} (\text{getQ} (\text{notifyWatches-loop literal} \ \text{Wl}' (\text{clause} \# \ \text{newWl}) ?\text{state}'')) \)

using prefixIsSubset[of getQ ?state''' getQ (notifyWatches-loop literal \ \text{Wl}' (\text{clause} \# \ \text{newWl}) ?\text{state}'')]

by auto

hence \( l \in \text{set} (\text{getQ} (\text{notifyWatches-loop literal} \ \text{Wl}' ?\text{state}''')) \)

using \( \langle \text{getQ} \ ?\text{state}'' \rangle \)

by auto

\}

thus \( ?\text{thesis} \)

using \( \langle \text{notifyWatches-loop literal} \ \text{Wl}' (\text{clause} \# \ \text{newWl}) \text{?state}'''' = \text{notifyWatches-loop literal} (\text{clause} \# \ \text{Wl}') \text{newWl state} \rangle \)

by (simp add:Let-def)

next

case \text{False}

hence \( c \in \text{set} \ \text{Wl}' \)

using \( \langle c \in \text{set} (\text{clause} \# \ \text{Wl}') \rangle \)

by simp

\{

fix \( l::\text{Literal} \)

assume isUnitClause \( \langle \text{nth} (\text{getF state}) c \rangle \) \( l \) \( (\text{elements} \ (\text{getM state})) \)

hence isUnitClause \( \langle \text{nth} (\text{getF ?state'''}) c \rangle \) \( l \) \( (\text{elements} \ (\text{getM ?state'''})) \)

unfolding setReason-def

unfolding swapWatches-def

by simp

with \( \langle \text{let state}''' = \text{notifyWatches-loop literal} (\text{clause} \# \ \text{Wl}') \text{newWl state in} \rangle \)

\( ?\text{Cond1 state'} ?\text{state}''' \wedge ?\text{Cond2 Wl'} state' ?\text{state}''' \)

\( c \in \text{set} \ \text{Wl}' \)

have \( \langle \text{let state}''' = \text{notifyWatches-loop literal} (\text{clause} \# \ \text{Wl}') \text{newWl state in} \rangle \)

\( l \in \text{set} (\text{getQ state'}) \)

by (simp add:Let-def)

\}

thus \( ?\text{thesis} \)

by (simp add:Let-def)

qed

\}

hence \( ?\text{Cond2} (\text{clause} \# \ \text{Wl}') (\text{notifyWatches-loop literal} (\text{clause} \# \ \text{Wl}') \text{newWl state}) \)

by (simp add: Let-def)

ultimately

show \( ?\text{thesis} \)

by (simp add:Let-def)

456
lemma InvariantUniqQAfterNotifyWatchesLoop:
fixes literal :: Literal and \( Wl :: \text{nat list} \) and \( \text{newWl} :: \text{nat list} \) and state :: State
assumes
\( \text{InvariantWatchesEl} (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state}) \)
and
\( \forall (c :: \text{nat}). \ c \in \text{set} \ Wl \rightarrow 0 \leq c \land c < \text{length} (\text{getF state}) \) and
\( \text{InvariantUniqQ} (\text{getQ state}) \)
shows
let state' = notifyWatches-loop literal Wl \( \text{newWl} \) state in
\( \text{InvariantUniqQ} (\text{getQ state'}) \)
using assms
proof (induct \( Wl \) arbitrary: \( \text{newWl} \) state)
case Nil
thus ?case
by simp
next
case (Cons clause \( Wl' \))
from \( \forall (c :: \text{nat}). \ c \in \text{set} \ \text{clause} \# \ Wl' \rightarrow 0 \leq c \land c < \text{length} (\text{getF state}) \)
have \( 0 \leq \text{clause} \land \text{clause} < \text{length} (\text{getF state}) \)
by auto
then obtain wa::Literal and wb::Literal
where getWatch1 state clause = Some wa and getWatch2 state clause = Some wb
using Cons
unfolding InvariantWatchesEl-def
by auto
show ?case
proof (cases Some literal = getWatch1 state clause)
case True
let ?state' = swapWatches clause state
let ?w1 = wb
have getWatch1 ?state' clause = Some ?w1
  using (getWatch2 state clause = Some wb)
  unfolding swapWatches-def
  by auto
let ?w2 = wa
have getWatch2 ?state' clause = Some ?w2
  using (getWatch1 state clause = Some wa)
  unfolding swapWatches-def
qed
by auto
show thesis
proof (cases literal True ?w1 (elements (getM ?state')))
case True
from Cons(2)
  have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto
moreover
have getM ?state' = getM state ∧
  getF ?state' = getF state ∧
  getQ ?state' = getQ state
unfolding swapWatches-def
by simp
ultimately
show thesis
  using Cons(1)[of ?state' clause # newWl]
  using Cons(3) Cons(4)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (literal True ?w1 (elements (getM ?state')))
  by (simp add: Let-def)
next
case False
show thesis
proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
  clause) ?w1 ?w2 (getM ?state'))
case (Some l')
hence l' el (nth (getF ?state') clause)
  using getNonWatchedUnfalsifiedLiteralSomeCharacterization
  by simp
let ?state'' = setWatch2 clause l' ?state'
from Cons(2)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
    using l' el (nth (getF ?state') clause):
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def
    unfolding setWatch2-def
    by auto
moreover
have getM ?state'' = getM state ∧
getF state'' = getF state ∧
getQ state'' = getQ state

unfolding swapWatches-def
unfolding setWatch2-def
by simp
ultimately
show ?thesis
  using Cons(1)[of ?state'' new]  
  using Cons(3) Cons(4)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (~ literalTrue ?w1 (elements (getM ?state')))
  using Some
  by (simp add: Let-def)

next
  case None
  show ?thesis
  proof (cases literalFalse ?w1 (elements (getM ?state')))
  case True
  let ?state'' = ?state'(getConflictFlag := True, getConflict-Clause := clause)

  from Cons(2)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
       (getWatch2 ?state'')
   unfolding InvariantWatchesEl-def
   unfolding swapWatches-def
   by auto
moreover
have getM ?state'' = getM state ∧
getF ?state'' = getF state ∧
getQ ?state'' = getQ state
    unfolding swapWatches-def
    by simp
ultimately
show ?thesis
  using Cons(1)[of ?state'' clause ≠ new]
  using Cons(3) Cons(4)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (~ literalTrue ?w1 (elements (getM ?state')))
  using None
  using (literalFalse ?w1 (elements (getM ?state')))
  by (simp add: Let-def)

next
  case False
  let ?state'' = setReason ?w1 clause (?state'' getQ := (if ?w1
el (getQ ?state) then (getQ ?state') else (getQ ?state') @ [?w1])
from Cons(2)
have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
  (getWatch2 ?state')
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  unfolding setReason-def
  by auto
moreover
have getM ?state'' = getM state
  getF ?state'' = getF state
  getQ ?state'' = (if ?w1 el (getQ state) then (getQ state) else
  (getQ state) @ [?w1])
  unfolding swapWatches-def
  unfolding setReason-def
  by auto
moreover
have uniq (getQ ?state'')
  using Cons(4)
  using (getQ ?state'' = (if ?w1 el (getQ state) then (getQ state) else
  (getQ state) @ [?w1]))
  unfolding InvariantUniqQ-def
  by (simp add: uniqAppendIff)
ultimately
show ?thesis
  using Cons(1)[of ?state''] clause # newWl
  using Cons(3)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (¬ literalTrue ?w1 (elements (getM ?state')))
  using None
  using (¬ literalFalse ?w1 (elements (getM ?state')))
  unfolding isPrefix-def
  unfolding InvariantUniqQ-def
  by (simp add: Let-def split: split-if-asm)
qed
qed
next
  case False
  let ?state' = state
  let ?w1 = wa
  have getWatch1 ?state' clause = Some ?w1
    using (getWatch1 state clause = Some wa)
    by auto
  let ?w2 = wb
  have getWatch2 ?state' clause = Some ?w2
    using (getWatch2 state clause = Some wb)
by auto

show \?thesis

proof (cases literalTrue \?w1 (elements (getM ?state'))
  case True
  thus \?thesis
    using Cons
    using (\neg Some literal = getWatch1 state clause)
    using (getWatch1 ?state' clause = Some \?w1)
    using (getWatch2 ?state' clause = Some \?w2)
    using (literalTrue \?w1 (elements (getM ?state')))
    by (simp add: Let-def)

next

  case False
  show \?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) \?w1 \?w2 (getM ?state')
      case (Some l')
      hence l' el (nth (getF ?state') clause)
      using getNonWatchedUnfalsifiedLiteralSomeCharacterization
      by simp

      let \?state'' = setWatch2 clause l' ?state'

      from Cons(2)
      have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
        using (l' el (nth (getF ?state') clause)
        unfolding InvariantWatchesEl-def
        unfolding setWatch2-def
        by auto

      moreover
      have getM ?state'' = getM state ∧
        getF ?state'' = getF state ∧
        getQ ?state'' = getQ state
        unfolding setWatch2-def
        by simp

      ultimately
      show \?thesis
        using Cons(1)[of \?state'']
        using Cons(3) Cons(4)
        using (getWatch1 ?state' clause = Some \?w1)
        using (getWatch2 ?state' clause = Some \?w2)
        using (\neg Some literal = getWatch1 state clause)
        using (\neg literalTrue \?w1 (elements (getM ?state')))
        using Some
        by (simp add: Let-def)

next

  case None
  show \?thesis
\textbf{proof} (cases literalFalse w1 (elements (getM ?state')))

case True
  let ?state'' = ?state'\langle getConflictFlag := True, getConflict-Clause := clause \rangle

from Cons(2) have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
  unfolding InvariantWatchesEl-def by auto
moreover have getM ?state'' = getM state \land
  getF ?state'' = getF state \land
  getQ ?state'' = getQ state
  by simp
ultimately show ?thesis
  using Cons(1)[of ?state'']
  using Cons(3) Cons(4)
  using (getWatch1 ?state' clause = Some w1)
  using (getWatch2 ?state' clause = Some w2)
  using (\neg Some literal = getWatch1 state clause)
  using (\neg literalTrue w1 (elements (getM ?state')))
  using None
  using (literalFalse w1 (elements (getM ?state')))
  by (simp add: Let-def)
next
case False
  let ?state'' = setReason w1 clause (?state'\langle getQ := (if \neg w1 el (getQ ?state') then (getQ ?state') else (getQ ?state) \@ [w1]) \rangle)
from Cons(2) have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
  unfolding InvariantWatchesEl-def
  unfolding setReason-def by auto
moreover have getM ?state'' = getM state
  getF ?state'' = getF state
  getQ ?state'' = (if \neg w1 el (getQ state) then (getQ state) else (getQ state) \@ [w1])
  unfolding setReason-def
  by auto
moreover have uniq (getQ ?state'')
  using Cons(4)
  using (getQ ?state'' = (if \neg w1 el (getQ state) then (getQ state) else (getQ state) \@ [w1]))
  unfolding InvariantUniqQ-def

462
by (simp add: uniqAppendIff)
ultimately
show \( ?\)thesis
  using Cons(1)[of \(?\)state’]
  using Cons(3)
  using (getWatch1 \(?\)state’ clause = Some \(?\)w1)
  using (getWatch2 \(?\)state’ clause = Some \(?\)w2)
  using (\(\neg\) Some literal = getWatch1 \(?\)state clause)
  using (\(\neg\) literalTrue \(?\)w1 (elements (getM \(?\)state’)))
  using None
  using (\(\neg\) literalFalse \(?\)w1 (elements (getM \(?\)state’)))
unfolding isPrefix-def
unfolding InvariantUniqQ-def
by (simp add: Let-def split: split-if-asm)
qed
qed
qed
qed

lemma InvariantConflictClauseCharacterizationAfterNotifyWatches:
  assumes
    (getM state) = M @ [(opposite literal, decision)] and
    InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  and
    \(\forall\) (c::nat). c \(\in\) set \(\text{Wl}\) \(\rightarrow\) \(0 \leq c \land c < \text{length}\) (getF state) and
    \(\forall\) (c::nat). c \(\in\) set \(\text{Wl}\) \(\rightarrow\) Some literal = (getWatch1 state c) \(\lor\)
  Some literal = (getWatch2 state c) and
  InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
unq \(\text{Wl}\)
shows
  let state’ = (notifyWatches-loop literal \(\text{Wl}\) newWl state) in
  InvariantConflictClauseCharacterization (getConflictFlag state’) (getConflictClause state’) (getF state’) (getM state’)
using assms
proof (induct \(\text{Wl}\) arbitrary: newWl state)
  case Nil
  thus \(?\)case
    by simp
  next
  case (Cons clause \(\text{Wl}\’)

from (uniq (clause \# \(\text{Wl}\’))
have clause \(\notin\) set \(\text{Wl}\’ uniq \(\text{Wl}\’
  by (auto simp add:uniqAppendIff))

from (\(\forall\) (c::nat). c \(\in\) set (clause \# \(\text{Wl}\’) \(\rightarrow\) \(0 \leq c \land c < \text{length}\)
  (getF state))
have \(0 \leq \text{clause} \land \text{clause} < \text{length} \,(\text{getF}\,\text{state})\)  
by auto  
then obtain \(wa::\text{Literal}\) and \(wb::\text{Literal}\)  
where \(\text{getWatch1}\,\text{state}\,\text{clause} = \text{Some}\,wa\) and \(\text{getWatch2}\,\text{state}\,\text{clause} = \text{Some}\,wb\)  
using Cons  
unfolding InvariantWatchesEl-def  
by auto  
show ?case  
proof (cases \(\text{Some}\,\text{literal} = \text{getWatch1}\,\text{state}\,\text{clause}\))  
  case True  
  let \(?\text{state}' = \text{swapWatches}\,\text{clause}\,\text{state}\)  
  let \(?w1 = wb\)  
  have \(\text{getWatch1}\,\text{?state}'\,\text{clause} = \text{Some}\,?w1\)  
    using \(\text{getWatch2}\,\text{state}\,\text{clause} = \text{Some}\,wb\)  
    unfolding swapWatches-def  
    by auto  
  let \(?w2 = wa\)  
  have \(\text{getWatch2}\,\text{?state}'\,\text{clause} = \text{Some}\,?w2\)  
    using \(\text{getWatch1}\,\text{state}\,\text{clause} = \text{Some}\,wa\)  
    unfolding swapWatches-def  
    by auto  
with True have \(?w2 = \text{literal}\)  
unfolding swapWatches-def  
by simp  
hence \(\text{literalFalse}\,?w2\,(\text{elements}\,(\text{getM}\,\text{state}))\)  
using Cons\((2)\)  
by simp  
show ?thesis  
proof (cases \(\text{literalTrue}\,?w1\,(\text{elements}\,(\text{getM}\,?\text{state}'))\))  
  case True  
  from Cons\((3)\)  
  have InvariantWatchesEl\,(\text{getF}\,?\text{state}')(\text{getWatch1}\,?\text{state}')\)  
    (\text{getWatch2}\,?\text{state}'))  
    unfolding InvariantWatchesEl-def  
    unfolding swapWatches-def  
    by auto  
  moreover  
  have \(\forall\,c.\,c \in \text{set}\,\text{Wl}' \to \text{Some}\,\text{literal} = \text{getWatch1}\,?\text{state}'\,c\ \lor\ \text{Some}\,\text{literal} = \text{getWatch2}\,?\text{state}'\,c\)  
    using Cons\((5)\)  
    unfolding swapWatches-def  
    by auto  
  moreover  
  have \(\text{getM}\,?\text{state}' = \text{getM}\,\text{state}\ \land\)
getF $state\' = getF \ state \land$
getConflictFlag $state' = getConflictFlag \ state \land$
getConflictClause $state' = getConflictClause \ state

unfolding swapWatches-def
by simp
ultimately
show $\neg \thesis$
  using Cons(1)[of $state'$ clause ≠ newWI]
  using Cons(2) Cons(4) Cons(6) Cons(7)
  using (getWatch1 $state' \ clause = Some \ ?w1)
  using (getWatch2 $state' \ clause = Some \ ?w2)
  using (Some literal = getWatch1 state clause)
  using (literalTrue \ ?w1 \ elements \ (getM \ $state'))
  using (uniq WI)
by (simp add:Let-def)
next
  case False
  show $\neg \thesis$
  proof (cases getNonWatchedUnfalsifiedLiteral \ (nth (getF $state' \ clause) \ ?w1 \ ?w2 \ (getM \ $state')))
  case (Some \ l')
  hence \ l' el \ (nth (getF \ $state' \ clause))
  using getNonWatchedUnfalsifiedLiteralSomeCharacterization
by simp

let $\state'' = setWatch2 \ clause \ l' \ \state'$
from Cons(3)
  have InvariantWatchesEl \ (getF \ $state'') \ (getWatch1 \ $state'')
\ (getWatch2 \ $state'')
using \ (l' el \ (nth \ (getF \ $state') \ clause))
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
unfolding setWatch2-def
by auto
moreover
have $\forall \ (c::nat). \ c \in set \ WI' \longrightarrow \ Some \ literal = (getWatch1 \ $state'' \ c) \ \lor \ Some \ literal = (getWatch2 \ $state'' \ c)$
using Cons(5)
using \ clause \notin \ set \ WI'
using swapWatchesEffect[of clause state]
unfolding setWatch2-def
by simp
moreover
have getM $state'' = getM \ state \land$
getF $state'' = getF \ state \land$
getConflictFlag $state'' = getConflictFlag \ state \land$
getConflictClause $state'' = getConflictClause \ state

465
unfolding \texttt{swapWatches-def}
unfolding \texttt{setWatch2-def}
by \texttt{simp}
ultimately

\textbf{show} \ ?thesis
  \textbf{using} \ Cons(1)[of \ ?state'' newWl]
  \textbf{using} Cons(2) Cons(4) Cons(6) Cons(7)
  \textbf{using} \ (getWatch1 \ ?state' \ clause = Some \ ?w1);
  \textbf{using} \ (getWatch2 \ ?state' \ clause = Some \ ?w2);
  \textbf{using} \ (Some \ literal = getWatch1 \ state \ clause)
  \textbf{using} \ (\neg \ literalTrue \ ?w1 \ (elements \ (getM \ ?state')));
  \textbf{using} \ Some
  \textbf{using} \ (uniq \ Wl)
by \ (simp add: \ Let-def)

\textbf{next}
\textbf{case} None
\textbf{show} \ ?thesis
\textbf{proof} \ (cases \ literalFalse \ ?w1 \ (elements \ (getM \ ?state')))
\textbf{case} True
  \textbf{let} \ ?state'' = ?state'[getConflictFlag := True, getConflict-Clause := clause]

from \ Cons(3)
\textbf{have} \ InvariantWatchesEl \ (getF \ ?state'') \ (getWatch1 \ ?state'') \ (getWatch2 \ ?state'')
  \textbf{unfolding} InvariantWatchesEl-def
  \textbf{unfolding} \ swapWatches-def
  \textbf{by} \ auto
moreover
\textbf{have} \ getM \ ?state'' = getM \ state \ ∧
  \textbf{getF} \ ?state'' = getF \ state \ ∧
  \textbf{getConflictFlag} \ ?state'' \ ∧
  \textbf{getConflictClause} \ ?state'' = clause
  \textbf{unfolding} \ swapWatches-def
  \textbf{by} \ simp
moreover
\textbf{have} \ ∀ \ (c::nat). \ c ∈ set \ Wl' \ → \ Some \ literal = (getWatch1 \ ?state'' c) \ ∨ \ Some \ literal = (getWatch2 \ ?state'' c)
  \textbf{using} \ Cons(5)
  \textbf{using} \ (clause \ \notin \ set \ Wl')
  \textbf{using} \ swapWatchesEffect[of \ clause \ state]
  \textbf{by} \ simp
moreover
\textbf{have} \ ∀ \ l. \ l \ el \ (nth \ (getF \ ?state'') \ clause) \ ∧ \ l \ \neq \ ?w1 \ ∧ \ l \ \neq \ ?w2 \ → \ literalFalse \ l \ (elements \ (getM \ ?state''))
  \textbf{using} \ None
  \textbf{using} \ (getWatch1 \ ?state' \ clause = Some \ ?w1)
  \textbf{using} \ (getWatch2 \ ?state' \ clause = Some \ ?w2)
  \textbf{using} \ getNonWatchedUnfalsifiedLiteralNoneCharacteriza-
tion[of nth (getF ?state) clause ?w1 ?w2 getM ?state]  
  unfolding setReason-def  
  unfolding swapWatches-def  
  by auto

hence clauseFalse (nth (getF state) clause) (elements (getM state))
  using (literalFalse ?w1 (elements (getM ?state)))
  using (literalFalse ?w2 (elements (getM state)))
  unfolding swapWatches-def  
  by (auto simp add: clauseFalseIffAll_literals_AreFalse)
moreover
have (nth (getF state) clause) el (getF state)  
  using (0 ≤ clause ∧ clause < length (getF state))  
  using nth-mem[of clause getF state]  
  by simp
ultimately
show ?thesis
  using Cons(1)[of ?state" clause ≠ newWl]
  using Cons(2) Cons(4) Cons(6) Cons(7)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (¬ literalTrue ?w1 (elements (getM ?state')))
  using None
  using (literalFalse ?w1 (elements (getM ?state')))
  using (uniq Wl')  
  using (0 ≤ clause ∧ clause < length (getF state))
  unfolding InvariantConflictClauseCharacterization-def
  by (simp add: Let-def)
next
  case False
  let ?state" = setReason ?w1 clause (??state"{getQ := (if ?w1 el (getQ ?state') then (getQ ?state') else (getQ ?state') @ [?w1])})
  from Cons(3)
  have InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state")
    unfolding InvariantWatchesEl-def
    unfolding swapWatches-def  
    unfolding setReason-def
    by auto
moreover
have getM ?state" = getM state
  getF ?state" = getF state
  getConflictFlag ?state" = getConflictFlag state
  getConflictClause ?state" = getConflictClause state
  unfolding swapWatches-def
  unfolding setReason-def
  by auto

467
moreover

have \( \forall \) \((c :: \text{nat}). c \in \text{set} Wl' \rightarrow \text{Some literal} = (\text{getWatch1} \ ?\text{state}'' c) \lor \text{Some literal} = (\text{getWatch2} \ ?\text{state}'' c)\)

using Cons(5)
using \((\text{clause} \notin \text{set} Wl')\)
using swapWatchesEffect[of clause state]

unfolding setReason-def
by simp

ultimately

show \(?\text{thesis}\)
using Cons(1)[of \(?\text{state}'' \text{ clause} \# \text{newWl}\)]
using Cons(2) Cons(4) Cons(6) Cons(7)
using \((\text{getWatch1} \ ?\text{state}' \text{ clause} = \text{Some} \ ?w1)\)
using \((\text{getWatch2} \ ?\text{state}' \text{ clause} = \text{Some} \ ?w2)\)
using \((\text{Some literal} = \text{getWatch1} \text{ state clause})\)
using None
using \((\neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}')))\)
using \((\text{uniq} Wl')\)
by (simp add: Let-def)

qed
qed
qed

next

case False

let \(?\text{state}' = \text{state}\)

let \(?w1 = wa\)

have \(\text{getWatch1} \ ?\text{state}' \text{ clause} = \text{Some} \ ?w1\)
using \((\text{getWatch1} \text{ state clause} = \text{Some} wa)\)
by auto

let \(?w2 = wb\)

have \(\text{getWatch2} \ ?\text{state}' \text{ clause} = \text{Some} \ ?w2\)
using \((\text{getWatch2} \text{ state clause} = \text{Some} wb)\)
by auto

from \((\neg \text{Some literal} = \text{getWatch1} \text{ state clause})\)
\(\forall \) \((c :: \text{nat}). c \in \text{set} (\text{clause} \# Wl') \rightarrow \text{Some literal} = (\text{getWatch1} \text{ state} c) \lor \text{Some literal} = (\text{getWatch2} \text{ state} c)\)

have \(\text{Some literal} = \text{getWatch2} \text{ state clause} \)
by auto

hence \(?w2 = \text{literal}\)
using \((\text{getWatch2} \ ?\text{state}' \text{ clause} = \text{Some} \ ?w2)\)
by simp

hence \(\text{literalFalse} \ ?w2 \ (\text{elements} \ (\text{getM} \text{ state}))\)
using Cons(2)
by simp

show \(?\text{thesis}\)

proof (cases literalTrue \(?w1 \ (\text{elements} \ (\text{getM} \ ?\text{state}'))))
case True
thus ?thesis
  using Cons(1)[of ?state' clause # newWl]
  using Cons(2) Cons(3) Cons(4) Cons(5) Cons(6) Cons(7)
  using (∨ Some literal = getWatch1 state clause)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (literalTrue ?w1 (elements (getM ?state')))
  using uniq Wl'
  by (simp add: Let-def)
next
  case False
  show ?thesis
  proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state') clause) ?w1 ?w2 (getM ?state'))
    case (Some l')
    hence l' el (nth (getF ?state')) clause
      using getNonWatchedUnfalsifiedLiteralSomeCharacterization
      by simp
    let ?state'' = setWatch2 clause l' ?state'
    from Cons(3)
    have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    using l' el (nth (getF ?state')) clause:
    unfolding InvariantWatchesEl-def
    unfolding setWatch2-def
    by auto
    moreover
    have getM ?state'' = getM state ∧
      getF ?state'' = getF state ∧
      getQ ?state'' = getQ state ∧
      getConflictFlag ?state'' = getConflictFlag state ∧
      getConflictClause ?state'' = getConflictClause state
    unfolding setWatch2-def
    by simp
    moreover
    have ∀ (c::nat). c ∈ set Wl' → Some literal = (getWatch1 ?state'' c) ∨ Some literal = (getWatch2 ?state'' c)
    using Cons(5)
    using clause / set Wl'
    unfolding setWatch2-def
    by simp
    ultimately
    show ?thesis
    using Cons(1)[of ?state'' newWl]
    using Cons(2) Cons(4) Cons(6) Cons(7)
    using (getWatch1 ?state' clause = Some ?w1)
using \langle getWatch2 ?state' clause = Some ?w2 \rangle
using \langle \neg Some literal = getWatch1 state clause \rangle
using \langle \neg literalTrue ?w1 (elements (getM ?state')) \rangle
using Some
using \langle uniq WI' \rangle
by (simp add: Let-def)

next
case None
show \( ?thesis \)
proof (cases literalFalse ?w1 (elements (getM ?state')))
case True
  let ?state'' = ?state'\{getConflictFlag := True, getConflictClause := clause\}
from Cons(3)
  have InvariantWatchesEl (getF ?state'') (getWatch1 ?state'')
  (getWatch2 ?state'')
  unfolding InvariantWatchesEl-def
  by auto
moreover
  have getM ?state'' = getM state ∧
    getF ?state'' = getF state ∧
    getQ ?state'' = getQ state ∧
    getConflictFlag ?state'' ∧
    getConflictClause ?state'' = clause
    by simp
moreover
  have \( \forall \ c :: nat. \ c \in set WI' \longrightarrow \ Some \ literal = (getWatch1 ?state'' c) \lor Some \ literal = (getWatch2 ?state'' c) \)
    using Cons(5)
    using \langle clause \notin set WI' \rangle
    by simp
moreover
  have \( \forall \ l. \ l \in (nth (getF ?state'') clause) \land \ l \neq ?w1 \land \ l \neq ?w2 \longrightarrow literalFalse l (elements (getM ?state'')) \)
    using None
    using \langle getWatch1 ?state' clause = Some ?w1 \rangle
    using \langle getWatch2 ?state' clause = Some ?w2 \rangle
    using getNonWatchedUnfalsifiedLiteralNoneCharacterization[of nth (getF ?state') clause ?w1 ?w2 getM ?state']
  unfolding setReason-def
  by auto
  hence \( clauseFalse \ (nth (getF state) clause) (elements (getM state)) \)
    using \langle literalFalse ?w1 (elements (getM ?state')) \rangle
    using \langle literalFalse ?w2 (elements (getM state)) \rangle
    by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
moreover
  have \( \nth (getF state) clause \in (getF state) \)
using \(0 \leq \text{clause} \land \text{clause} < \text{length} (\text{getF state})\)

using \(\text{nth-mem}[\text{of clause} \text{getF state}]\)

by simp

ultimately

show \(?\text{thesis}\)

using \(\text{Cons}(1)[\text{of ?state}]\)

using \(\text{Cons}(2) \text{Cons}(4) \text{Cons}(6) \text{Cons}(7)\)

using \((\text{getWatch1 ?state}\) clause = Some \(?w1)\)

using \((\text{getWatch2 ?state}\) clause = Some \(?w2)\)

using \((\neg \text{Some literal} = \text{getWatch1 state clause})\)

using \((\neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM ?state})))\)

using None

using \(\neg \text{literalFalse} \ ?w1 \ (\text{elements} \ (\text{getM ?state})))\)

using \(\text{uniq WI}\)

using \((0 \leq \text{clause} \land \text{clause} < \text{length} (\text{getF state}))\)

unfolding \(\text{InvariantConflictsClauseCharacterization-def}\)

by (simp add: \(\text{Let-def}\))

next

case False

let \(?\text{state}'' = \text{setReason} \ ?w1 \text{clause} \ (\text{?state}'\) \((\text{getQ} := (\text{if} \ ?w1 \ else \ (\text{getQ}\ ?\text{state}'\) \else \ (\text{getQ}\ ?\text{state}'\) \@ \[\?w1])))\)

from \(\text{Cons}(3)\)

have \(\text{InvariantWatchesEl} (\text{getF ?state}''\) (\text{getWatch1 ?state}''\)

\((\text{getWatch2 ?state}''\)

unfolding \(\text{InvariantWatchesEl-def}\)

unfolding \(\text{setReason-def}\)

by auto

moreover

have \(\text{getM ?state}'' = \text{getM state}\)

\(\text{getF ?state}'' = \text{getF state}\)

\(\text{getConflictFlag ?state}'' = \text{getConflictFlag state}\)

\(\text{getConflictClause ?state}'' = \text{getConflictClause state}\)

unfolding \(\text{setReason-def}\)

by auto

moreover

have \(\forall \ (c::\text{nat}) \ c \in \text{set WI'} \rightarrow \text{Some literal} = (\text{getWatch1 ?state}''\ c) \lor \text{Some literal} = (\text{getWatch2 ?state}''\ c)\)

using \(\text{Cons}(5)\)

using \((\text{clause} \notin \text{set WI'})\)

unfolding \(\text{setReason-def}\)

by simp

ultimately

show \(?\text{thesis}\)

using \(\text{Cons}(1)[\text{of ?state}]\)

using \(\text{Cons}(2) \text{Cons}(4) \text{Cons}(6) \text{Cons}(7)\)

using \((\text{getWatch1 ?state}\) clause = Some \(?w1)\)

using \((\text{getWatch2 ?state}\) clause = Some \(?w2)\)

using \((\neg \text{Some literal} = \text{getWatch1 state clause})\)

using \((\neg \text{literalTrue} \ ?w1 \ (\text{elements} \ (\text{getM ?state})))\)

471
using None
using (∼ literalFalse ?w1 (elements (getM ?state')))
using (uniq Wl'
by (simp add: Let-def)
qed
ded
qed

lemma InvariantGetReasonIsReasonQSubset:
assumes Q ⊆ Q’ and
InvariantGetReasonIsReason GetReason F M Q’
sows
InvariantGetReasonIsReason GetReason F M Q
using assms
unfolding InvariantGetReasonIsReason-def
by auto

lemma InvariantGetReasonIsReasonAfterNotifyWatches:
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
∀ (c::nat). c ∈ set Wl → 0 ≤ c ∧ c < length (getF state) and
∀ (c::nat). c ∈ set Wl → Some literal = (getWatch1 state c) ∨
Some literal = (getWatch2 state c) and
uniq Wl
getM state = M @ [(opposite literal, decision)]
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) Q
sows
let state’ = notifyWatches-loop literal Wl newWl state in
let Q’ = Q ∪ (set (getQ state’) − set (getQ state)) in
InvariantGetReasonIsReason (getReason state’) (getF state’) (getM state’) Q’
using assms
proof (induct Wl arbitrary: newWl state Q)
case Nil
thus ?case
by simp
next
case (Cons clause Wl')
from (uniq (clause # Wl'))
have clause ∉ set Wl' uniq Wl'
by (auto simp add:uniqAppendIff)
from (∀ (c::nat). c ∈ set (clause # Wl') → 0 ≤ c ∧ c < length (getF state))

472
have \(0 \leq \text{clause} \land \text{clause} < \text{length}(\text{getF state})\)
by auto
then obtain \(wa::\text{Literal}\) and \(wb::\text{Literal}\)
where \(\text{getWatch1 state clause} = \text{Some wa}\) and \(\text{getWatch2 state clause} = \text{Some wb}\)
using Cons
unfolding InvariantWatchesEl-def
by auto
show \(?\text{case}\)
proof (cases \(\text{Some literal} = \text{getWatch1 state clause}\))
case True
let \(?\text{state}^{'} = \text{swapWatches clause state}\)
let \(?w1 = wb\)
have \(\text{getWatch1 ?state}^{'} \text{ clause} = \text{Some ?w1}\)
using \(\langle \text{getWatch2 state clause} = \text{Some wb}\rangle\)
unfolding swapWatches-def
by auto
let \(?w2 = wa\)
have \(\text{getWatch2 ?state}^{'} \text{ clause} = \text{Some ?w2}\)
using \(\langle \text{getWatch1 state clause} = \text{Some wa}\rangle\)
unfolding swapWatches-def
by auto
with True have
\(?w2 = \text{literal}\)
unfolding swapWatches-def
by simp
hence \(\text{literalFalse ?w2 (elements (getM state))}\)
using Cons(6)
by simp
from \(\text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}\)
have \(?w1 el (\text{nth (getF state) clause}) ?w2 el (\text{nth (getF state) clause})\)
using \(\langle \text{getWatch1 ?state}^{'} \text{ clause} = \text{Some ?w1}\rangle\)
using \(\langle \text{getWatch2 ?state}^{'} \text{ clause} = \text{Some ?w2}\rangle\)
using \(\langle 0 \leq \text{clause} \land \text{clause} < \text{length}(\text{getF state})\rangle\)
unfolding InvariantWatchesEl-def
unfolding swapWatches-def
by auto
show \(?\text{thesis}\)
proof (cases \(\text{literalTrue ?w1 (elements (getM ?state'))}\))
case True
from Cons(2)
have \(\text{InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state')}\)
unfolding InvariantWatchesEl-def
unfolding swapWatches-def by auto
moreover
have ∀ c  . c ∈ set Wl' → Some literal = getWatch1 ?state' c ∨
  Some literal = getWatch2 ?state' c
  using Cons(4)
unfolding swapWatches-def by auto
moreover
have getM ?state' = getM state ∧
  getF ?state' = getF state ∧
  getQ ?state' = getQ state ∧
  getReason ?state' = getReason state

unfolding swapWatches-def by simp
ultimately
show ?thesis
  using Cons(1)[of ?state' Q clause # newWl]
  using Cons(3) Cons(6) Cons(7)
  using (getWatch1 ?state' clause = Some ?w1)
  using (getWatch2 ?state' clause = Some ?w2)
  using (Some literal = getWatch1 state clause)
  using (literalTrue ?w1 {elements (getM ?state')})
  using (uniq Wl)
  by (simp add:Let-def)
next
case False
  show ?thesis
    proof (cases getNonWatchedUnfalsifiedLiteral (nth (getF ?state')
      clause) ?w1 ?w2 (getM ?state'))
      case (Some l')
      hence l' el (nth (getF ?state') clause)
        using getNonWatchedUnfalsifiedLiteralSomeCharacterization
        by simp
    let ?state'' = setWatch2 clause l' ?state'
    from Cons(2)
    have InvariantWatchesEl (getF ?state') (getWatch1 ?state'')
      (getWatch2 ?state'')
        using l' el (nth (getF ?state') clause):
          unfolding InvariantWatchesEl-def
          unfolding swapWatches-def
          unfolding setWatch2-def
          by auto
moreover
have ∀ (c::nat). c ∈ set Wl' → Some literal = (getWatch1 ?state'' c) ∨ Some literal = (getWatch2 ?state'' c)
using Cons(4)
using ⟨clause ∉ set Wl⟩
using swapWatchesEffect[of clause state]
unfolding setWatch2-def
by simp
moreover
have getM ?state" = getM state ∧
  getF ?state" = getF state ∧
  getQ ?state" = getQ state ∧
  getReason ?state" = getReason state
unfolding swapWatches-def
unfolding setWatch2-def
by simp
ultimately
show ?thesis
using Cons(1)[of ?state" Q newWl]
using Cons(3) Cons(6) Cons(7)
using ⟨getWatch1 ?state’ clause = Some ?w1⟩
using ⟨getWatch2 ?state’ clause = Some ?w2⟩
using ⟨Some literal = getWatch1 state clause⟩
using ⟨¬ literalTrue ?w1 (elements (getM ?state’))⟩
using Some
using ⟨uniq Wl⟩
by (simp add: Let-def)

next
case None
   hence ∀ l. l el (nth (getF ?state’) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM ?state’))
   using getNonWatchedUnfalsifiedLiteralNoneCharacterization
   by simp
show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state’)))
case True
   let ?state" = ?state[getConflictFlag := True, getConflict-Clause := clause]

from Cons(2)
have InvariantWatchesEl (getF ?state") (getWatch1 ?state’)
  (getWatch2 ?state’)
  unfolding InvariantWatchesEl-def
  unfolding swapWatches-def
  by auto
moreover
have ∀ c. c ∈ set Wl’ → Some literal = getWatch1 ?state" ∧ Some literal = getWatch2 ?state" c
  using Cons(4)
  unfolding swapWatches-def
  by auto
moreover
have \( getM \ ?state'' = getM \ state \land \\
getF \ ?state'' = getF \ state \land \\
getQ \ ?state'' = getQ \ state \land \\
getReason \ ?state'' = getReason \ state \\
\) unfolding swapWatches-def \\
by simp \\
ultimately \\
show \(?thesis\) \\
using Cons(1)\[ ?state'' Qclause \# \ newWl\] \\
using Cons(3) Cons(6) Cons(7) \\
using \(getWatch1 \ ?state'\) clause = Some \(?w1\) \\
using \(getWatch2 \ ?state'\) clause = Some \(?w2\) \\
using \(\text{Some literal} = \text{getWatch1 state clause}\) \\
using \(\neg \ \text{literalTrue} \ ?w1 \ \text{(elements} \ \text{getM} \ ?state')\) \\
using None \\
using \(\text{literalFalse} \ ?w1 \ \text{(elements} \ \text{getM} \ ?state')\) \\
using \(\text{uniq} \ Wl\) \\
by (simp add: Let-def) \\
next \\
case False \\
let \(?state'' = setReason \ ?w1 \ \text{clause} \ (\ ?state' (if \ ?w1 \ \text{el} \ (getQ \ ?state') \ \text{then} \ (getQ \ ?state') \ \text{else} \ (getQ \ ?state') @ [\?w1]))) \\
let \(?state0 = notifyWatches-loop \ \text{literal} \ Wl' \ \text{(clause} \ # \ \text{newWl}) \ ?state'' \\
from Cons(2) \\
have InvariantWatchesEl \((getF \ ?state'') \ (getWatch1 \ ?state'')\) \\
\(\) unfolding InvariantWatchesEl-def \\
unfolding swapWatches-def \\
unfolding setReason-def \\
by auto \\
moreover \\
have \( getM \ ?state'' = getM \ state \land \\
getF \ ?state'' = getF \ state \land \\
getQ \ ?state'' = (if \ ?w1 \ \text{el} \ (getQ \ ?state) \ \text{then} \ (getQ \ ?state) \ \text{else} \ (getQ \ state) @ [\?w1]) \\
getReason \ ?state'' = (getReason \ state)(?w1 := \text{Some} \ \text{clause}) \\
\) unfolding swapWatches-def \\
unfolding setReason-def \\
by auto \\
moreover \\
hence \forall \ (c::nat). \ c \ \text{in} \ Wl' \ \text{---} \ \text{Some literal} = (getWatch1 \ ?state'' c) \ \text{or} \ \text{Some literal} = (getWatch2 \ ?state'' c) \\
using Cons(4) \\
using \(\text{clause} \ \notin \ \text{set} \ Wl\) \\
using swapWatchesEffect[of clause state] \\
unfolding setReason-def 

476
by simp
moreover
have isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))
  using (∀ l. l el (nth (getF state) clause) ∧ l ≠ ?w1 ∧ l ≠ ?w2 → literalFalse l (elements (getM state))):
  using (∀ el (nth (getF state) clause))
  using (∀ el (nth (getF state) clause))
  using (¬ literalTrue ?w1 (elements (getM state))):
  using (¬ literalFalse ?w2 (elements (getM state))):
unfolding swapWatches-def
unfolding isUnitClause-def
by auto

hence InvariantGetReasonIsReason (getReason ?state′′) (getF ?state′′) (getM ?state′′) (Q ∪ {?w1})
  using Cons(7)
  using (getM state′′ = getM state)
  using (getF state′′ = getF state)
  using (getQ state′′ = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])):
  using (getReason state′′ = (getReason state)(?w1 := Some clause)):
  using (0 ≤ clause ∧ clause < length (getF state))
  using (¬ literalTrue ?w1 (elements (getM state))):
  using (isUnitClause (nth (getF state) clause) ?w1 (elements (getM state))):
unfolding swapWatches-def
unfolding InvariantGetReasonIsReason-def
by auto
moreover
have (λa. if a = ?w1 then Some clause else getReason state a) = getReason ?state′′
unfolding setReason-def
unfolding swapWatches-def
by (auto simp add: fun-upd-def)
ultimately
have InvariantGetReasonIsReason (getReason ?state0) (getF ?state0) (getM ?state0)
  (Q ∪ (set (getQ state0) − set (getQ state′′)) ∪ {?w1})
  using Cons(1)[of state′′ Q ∪ {?w1} clause # newWl]
  using Cons(3) Cons(6) Cons(7)
  using [uniq Wl]
  by (simp add: Let-def split: split-if-asm)
moreover
have (Q ∪ (set (getQ state0) − set (getQ state))) ⊆ (Q ∪ (set (getQ state0) − set (getQ state′′)) ∪ {?w1})
  using (getQ state′′ = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])):
unfolding \texttt{swapWatches-def}

by auto

ultimately

have \texttt{InvariantGetReasonIsReason (getReason ?state0) (getF ?state0) (getM ?state0)}

\( (Q \cup \{ \text{set (getQ ?state0)} - \text{set (getQ state)} \}) \)

using \texttt{InvariantGetReasonIsReasonQSubset} \( Q \cup \{ \text{set (getQ state)} \} \)

getReason ?state0 getF ?state0 getM ?state0

by simp

moreover

have \texttt{notifyWatches-loop literal \( \text{clause \# Wl'} \) newWl state = ?state0}

using \( \{ \text{getWatch1 ?state'} \text{ clause = Some \(?w1\)} \}

using \( \{ \text{getWatch2 ?state'} \text{ clause = Some \(?w2\)} \}

using \( \{ \text{Some literal = getWatch1 state clause} \}

using \( \{ \neg \text{literalTrue \(?w1\) (elements (getM ?state'))} \}

using \( \{ \neg \text{literalFalse \(?w1\) (elements (getM ?state'))} \}

using \( \{ \text{uniq Wl'} \}

by (simp add: Let-def)

ultimately

show \(?thesis\)

by simp

qed

qed

next

case False

let ?state' = state

let ?w1 = wa

have \texttt{getWatch1 ?state' clause = Some \(?w1\)}

using \( \{ \text{getWatch1 state clause = Some \(?w1\)} \}

by auto

let ?w2 = wb

have \texttt{getWatch2 ?state' clause = Some \(?w2\)}

using \( \{ \text{getWatch2 state clause = Some \(?w2\)} \}

by auto

have \(?w2 = \text{literal}\)

using \( \{ 0 \le \text{clause} \land \text{clause < length (getF state)} \}

using \( \{ \text{getWatch1 ?state' clause = Some \(?w1\)} \}

using \( \{ \text{getWatch2 ?state' clause = Some \(?w2\)} \}

using \( \{ \text{Cons(4)} \}

using \( \{ \text{False} \}

by simp

hence \texttt{literalFalse \(?w2\) (elements (getM state))}
using \texttt{Cons}(6)
by \texttt{simp}

from \texttt{InvariantWatchesEl \{getF state\} \{getWatch1 state\} \{getWatch2 state\})
have \(\var{w1} \in \text{nth \{getF state\} clause}\) \(\var{w2} \in \text{nth \{getF state\} clause}\)
using \texttt{getWatch1 \{state\} clause = \text{Some \var{w1}\}}
using \texttt{getWatch2 \{state\} clause = \text{Some \var{w2}\}}
using \((0 \leq \text{clause} \land \text{clause} < \text{length \{getF state\}})\)
unfolding \texttt{InvariantWatchesEl-def}
unfolding \texttt{swapWatches-def}
by \texttt{auto}

show \(?thesis\)
proof (cases literalTrue \var{w1} \{elements \{getM \{state\}\}\})
case True
thus \(?thesis\)
using \texttt{Cons}(1)\[\text{of \{state Q clause \# newWl\}}
using \texttt{Cons}(2) \texttt{Cons}(3) \texttt{Cons}(4) \texttt{Cons}(5) \texttt{Cons}(6) \texttt{Cons}(7)
using \((\neg \text{Some literal} = \text{getWatch1 state clause})\)
using \texttt{getWatch1 \{state\} clause = \text{Some \var{w1}\}}
using \texttt{getWatch2 \{state\} clause = \text{Some \var{w2}\}}
using \texttt{literalTrue \var{w1} \{elements \{getM \{state\}\}\}}
using \texttt{uniq Wl'}
by (\texttt{simp add:Let-def})
next
case False
show \(?thesis\)
proof (cases getNonWatchedUnfalsifiedLiteral \{nth \{getF ?state\} clause\} \var{w1} \var{w2} \{getM ?state\}\})
case \((\text{Some \var{l}'})\)
hence \(\var{l}' \in \text{nth \{getF ?state\} clause}\)
using \texttt{getNonWatchedUnfalsifiedLiteralSomeCharacterization}
by \texttt{simp}

let \(?state''' = \text{setWatch2 clause \var{l}' ?state'}\)

from \texttt{Cons}(2)
have \texttt{InvariantWatchesEl \{getF \var{state'''}\} \{getWatch1 \var{state'''}\} \{getWatch2 \var{state'''}\}}
using \texttt{l' el \{nth \{getF \var{state'''}\} clause\}}
unfolding \texttt{InvariantWatchesEl-def}
unfolding \texttt{setWatch2-def}
by \texttt{auto}
moreover
have \((\forall \text{c. c \in set Wl'} \rightarrow \text{Some literal} = \text{getWatch1 \var{state'''} c}) \lor \text{Some literal} = \text{getWatch2 \var{state'''} c})\)
using \texttt{Cons}(4)
using ⟨clause ∉ set Wl′⟩

unfolding setWatch2-def
by simp

moreover
have getM ?state" = getM state ∧
  getF ?state" = getF state ∧
  getQ ?state" = getQ state ∧
  getReason ?state" = getReason state
unfolding setWatch2-def
by simp

ultimately
show ?thesis
proof (cases literalFalse ?w1 (elements (getM ?state')))
case True
  let ?state" = ?state"[getConflictFlag := True, getConflict-Clause := clause]

from Cons(2)
  have InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state")
  unfolding InvariantWatchesEl-def
  by auto

moreover
have ∀c. c ∈ set Wl' → Some literal = getWatch1 ?state" c
  using Cons(4)
  using ⟨clause ∉ set Wl′⟩
  unfolding setWatch2-def
  by simp
moreover
have getM ?state" = getM state ∧
  getF ?state" = getF state ∧
getQ ?state'' = getQ state ∧
getReason ?state'' = getReason state
by simp
ultimately
show ?thesis
using Cons(1)[of ?state’’]
using Cons(3) Cons(6) Cons(7)
using (getWatch1 ?state’’ clause = Some ?w1)
using (getWatch2 ?state’’ clause = Some ?w2)
using (∼ Some literal = getWatch1 state clause)
using (∼ literalTrue ?w1 (elements (getM ?state’’))
using None
using (literalFalse ?w1 (elements (getM ?state’’))
using (uniq WI’’)
by (simp add: Let-def)

next
case False
let ?state'' = setReason ?w1 clause (?state’’(getQ := (if ?w1
el (getQ ?state’’) then (getQ ?state’’) else (getQ ?state’’) @ [?w1])))
let ?state0 = notifyWatches-loop literal WI’’ (clause # newWI)
?state’’

from Cons(2)
have InvariantWatchesEl (getF ?state’’) (getWatch1 ?state’’)
(getWatch2 ?state’’)
unfolding InvariantWatchesEl-def
unfolding setReason-def
by auto
moreover
have getM ?state’’ = getM state
getF ?state’’ = getF state
getQ ?state’’ = (if ?w1 el (getQ state) then (getQ state) else
(getQ state) @ [?w1])
getReason ?state’’ = (getReason state)(?w1 := Some clause)
unfolding setReason-def
by auto
moreover
hence ∀ (c::nat). c ∈ set WI’ → Some literal = (getWatch1
?state’’ c) ∨ Some literal = (getWatch2 ?state’’ c)
using Cons(4)
using (clause ∉ set WI’)
unfolding setReason-def
by simp
moreover
have isUnitClause (nth (getF state) clause) ?w1 (elements
(getM state))
using (∓ l. l el (nth (getF ?state’’) clause) ∧ l ≠ ?w1 ∧ l ≠
?w2 → literalFalse l (elements (getM ?state’’)))

481
using (?w1 el (nth (getF state) clause))
using (?w2 el (nth (getF state) clause))
using (¬ literalTrue ?w1 (elements (getM ?state)) +)
using (¬ literalFalse ?w1 (elements (getM ?state)) +)
using (literalFalse ?w2 (elements (getM state)) +)

unfolding isUnitClause-def
by auto

hence InvariantGetReasonIsReason (getReason ?state") (getF ?state") (getM ?state") (Q ∪ {?w1})
  using Cons(7)
  using (getM ?state" = getM state)
  using (getF ?state" = getF state)
  using (getQ ?state" = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])):
  using (getReason ?state" = (getReason state)(?w1 := Some clause)):
  using (0 ≤ clause ∧ clause < length (getF state))
  using (¬ literalTrue ?w1 (elements (getM ?state)) +)
  using (isUnitClause (nth (getF state) clause) ?w1 (elements (getM state)) +)

unfolding InvariantGetReasonIsReason-def
by auto

moreover have (λa. if a = ?w1 then Some clause else getReason state
a) = getReason ?state"
  unfolding setReason-def
  by (auto simp add: fun-upd-def)

ultimately have InvariantGetReasonIsReason (getReason ?state0) (getF ?state0) (getM ?state0) (Q ∪ (set (getQ ?state0) − set (getQ ?state")) ∪ {?w1})
  using Cons(1)of ?state" Q ∪ {?w1} clause # new Wl
  using Cons(3) Cons(6) Cons(7)
  using (uniq Wl)
  by (simp add: Let-def split: split-if-asm)

moreover have (Q ∪ (set (getQ ?state0) − set (getQ state))) ⊆ (Q ∪ (set (getQ ?state0) − set (getQ ?state")) ∪ {?w1})
  using (getQ ?state" = (if ?w1 el (getQ state) then (getQ state) else (getQ state) @ [?w1])):
  by auto

ultimately have InvariantGetReasonIsReason (getReason ?state0) (getF ?state0) (getM ?state0)
  (Q ∪ (set (getQ ?state0) − set (getQ state)))
  using InvariantGetReasonIsReasonQSubset[of Q ∪ (set (getQ ?state0) − set (getQ state))]
  using InvariantGetReasonIsReasonQSubset[of Q ∪ (set (getQ ?state0) − set (getQ ?state")) ∪ {?w1}]
  getReason ?state0 getF ?state0 getM ?state0]
by simp
moreover
have notifyWatches-loop literal (clause # Wl′) newWl state = ?state0
  using (getWatch1 ?state′ clause = Some ?w1)
  using (getWatch2 ?state′ clause = Some ?w2)
  using (∼ Some literal = getWatch1 state clause)
  using None
  using (∼ literalTrue ?w1 (elements (getM ?state′))
  using (uniq Wl′)
  by (simp add: Let-def)
ultimately
  show ?thesis
  by simp
qed
qed
qed
qed

lemma assertLiteralEffect:
fixes state::State and l::Literal and d::bool
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
shows
  (getM (assertLiteral l d state)) = (getM state) @ [(l, d)] and
  (getF (assertLiteral l d state)) = (getF state) and
  (getSATFlag (assertLiteral l d state)) = (getSATFlag state) and
  isPrefix (getQ state) (getQ (assertLiteral l d state))
  using assms
  unfolding assertLiteral-def
  unfolding notifyWatches-def
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  using notifyWatchesLoopPreservedVariables[of (state[@ (getM := getM state @ [(l, d)]])) getWatchList (state[@ (getM := getM state @ [(l, d)])]) (opposite l)]
  by (auto simp add: Let-def)

lemma WatchInvariantsAfterAssertLiteral:
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
\( (\text{getF state}) \) \textbf{and} 
\( \text{InvariantWatchListsUniq} \ (\text{getWatchList state}) \) \textbf{and} 
\( \text{InvariantWatchListsCharacterization} \ (\text{getWatchList state}) \) \( (\text{getWatch1 state}) \) \textbf{and} 
\( \text{InvariantWatchesEl} \ (\text{getF state}) \) \( (\text{getWatch1 state}) \) \( (\text{getWatch2 state}) \)

and

\( \text{InvariantWatchesDiffer} \ (\text{getF state}) \) \( (\text{getWatch1 state}) \) \( (\text{getWatch2 state}) \)

shows

let \( \text{state}' = (\text{assertLiteral literal decision state}) \) in 
\( \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList state}') \) \( (\text{getF state}') \) \wedge 
\( \text{InvariantWatchListsUniq} \ (\text{getWatchList state}') \) \( (\text{getWatch1 state}') \) \( (\text{getWatch2 state}') \) \wedge 
\( \text{InvariantWatchesEl} \ (\text{getF state}') \) \( (\text{getWatch1 state}') \) \( (\text{getWatch2 state}') \) \wedge 
\( \text{InvariantWatchesDiffer} \ (\text{getF state}') \) \( (\text{getWatch1 state}') \) \( (\text{getWatch2 state}') \)

using \( \text{assms} \)

unfolding \( \text{assertLiteral-def} \)

unfolding \( \text{notifyWatches-def} \)

using \( \text{InvariantWatchesElNotifyWatchesLoop[of state][getM := getM state \@ \((\text{literal, decision})\)] getWatchList state \text{(opposite literal)} \text{opposite literal} \} \)

using \( \text{InvariantWatchesDifferNotifyWatchesLoop[of state][getM := getM state \@ \((\text{literal, decision})\)] getWatchList state \text{(opposite literal)} \text{opposite literal} \} \)

using \( \text{InvariantWatchListsContainOnlyClausesFromFNotifyWatchesLoop[of state][getM := getM state \@ \((\text{literal, decision})\)] getWatchList state \text{(opposite literal)} \text{opposite literal} \} \)

unfolding \( \text{InvariantWatchListsContainOnlyClausesFromF-def} \)

unfolding \( \text{InvariantWatchListsCharacterization-def} \)

unfolding \( \text{InvariantWatchListsUniq-def} \)

by \( (\text{auto simp add: Let-def}) \)

\begin{verbatim}

lemma \text{InvariantWatchCharacterizationAfterAssertLiteral:}
assumes
\( \text{InvariantConsistent} \ (\text{getM state} \@ \((\text{literal, decision})\)) \) \textbf{and} 
\( \text{InvariantUniq} \ (\text{getM state} \@ \((\text{literal, decision})\)) \) \textbf{and} 
\( \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList state}) \) \( (\text{getF state}) \) \textbf{and} 
\( \text{InvariantWatchListsUniq} \ (\text{getWatchList state}) \) \textbf{and}
\end{verbatim}

484
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesElt (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
shows
let state' = (assertLiteral literal decision state)
in
InvariantWatchCharacterization (getF state') (getWatch1 state') (getWatch2 state') (getM state')
proof
let ?state = state ![getM := getM state @ [(literal, decision)]]]
let ?state' = assertLiteral literal decision state
have *: ∀ c. c ∈ set (getWatchList ?state (opposite literal)) →
  (∀ w1 w2. Some w1 = getWatch1 ?state' c ∧ Some w2 = getWatch2 ?state' c →
  watchCharacterizationCondition w1 w2 (getM ?state') (getF ?state' ! c) ∧
  watchCharacterizationCondition w2 w1 (getM ?state') (getF ?state' ! c))
  using assms
  using NotifyWatchesLoopWatchCharacterizationEffect ![of ?state getM state getWatchList ?state (opposite literal) opposite literal decision ]]
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsCharacterization-def
unfolding assertLiteral-def
unfolding notifyWatches-def
by (simp add: Let-def)
{
  fix c
  assume 0 ≤ c and c < length (getF ?state')
  fix w1::Literal and w2::Literal
  assume Some w1 = getWatch1 ?state' c Some w2 = getWatch2 ?state' c
  have watchCharacterizationCondition w1 w2 (getM ?state') (getF ?state' ! c) ∧
    watchCharacterizationCondition w2 w1 (getM ?state') (getF ?state' ! c)
    proof (cases c ∈ set (getWatchList ?state (opposite literal)))
      case True
      thus *thesis
      using *
      using ⟨Some w1 = getWatch1 ?state' c⟩ ⟨Some w2 = getWatch2 ?state' c⟩
        by auto

next
    case False
    hence Some (opposite literal) ≠ getWatch1 state c and Some (opposite literal) ≠ getWatch2 state c
    using (InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state))
    unfolding InvariantWatchListsCharacterization-def
    by auto
    moreover
    from assms False
    have getWatch1 ?state' c = getWatch1 state c and getWatch2 ?state' c = getWatch2 state c
    using notifyWatchesLoopPreservedWatches[of ?state getWatchList ?state (opposite literal) opposite literal []]
    unfolding False
    unfolding assertLiteral-def
    unfolding notifyWatches-def
    unfolding InvariantWatchListsContainOnlyClausesFromF-def
    by (auto simp add: Let-def)
    ultimately
    have w1 ≠ opposite literal w2 ≠ opposite literal
    using (Some w1 = getWatch1 ?state' c) and (Some w2 = getWatch2 ?state' c)
    by auto

    have watchCharacterizationCondition w1 w2 (getM state) (getF state ! c) and
        watchCharacterizationCondition w2 w1 (getM state) (getF state ! c)
    using (InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state))
    unfolding (Some w1 = getWatch1 ?state' c) and (Some w2 = getWatch2 ?state' c)
    using (getWatch1 ?state' c = getWatch1 state c) and (getWatch2 ?state' c = getWatch2 state c)
    unfolding InvariantWatchCharacterization-def
    using (c < length (getF ?state'))
    using assms
    using assertLiteralEffect[of state literal decision]
    by auto

    have watchCharacterizationCondition w1 w2 (getM ?state') ((getF ?state') ! c)
    proof-
    { assume literalFalse w1 (elements (getM ?state'))
        with (w1 ≠ opposite literal)
        have literalFalse w1 (elements (getM state))
        using assms

        486
using assertLiteralEffect[of state literal decision]
by simp
with (watchCharacterizationCondition w1 w2 (getM state)
  (getF state ! c))
  have (\exists l. l el ((getF state) ! c) \land literalTrue l (elements (getM state))
      \land elementLevel l (getM state) \leq elementLevel (opposite w1)
  (getM state)) \lor
      (\forall l. l el ((getF state) ! c) \land l \neq w1 \land l \neq w2 \rightarrow
        literalFalse l (elements (getM state)) \land
        elementLevel (opposite l) (getM state) \leq elementLevel
        (opposite w1) (getM state)) (is ?a state \lor ?b state)
  unfolding watchCharacterizationCondition-def
  using assms
  using assertLiteralEffect[of state literal decision]
  using (w1 \neq opposite literal)
  by simp
  have ?a ?state' \lor ?b ?state'
  proof (cases ?b state)
   case True
   show ?thesis
   proof
    { fix l
      assume l el (nth (getF ?state') c) l \neq w1 l \neq w2
      have literalFalse l (elements (getM ?state')) \land
        elementLevel (opposite l) (getM ?state') \leq elementLevel
        (opposite w1) (getM ?state')
      proof
        from True l el (nth (getF ?state') c): l \neq w1 \land l \neq w2
        have literalFalse l (elements (getM state))
        elementLevel (opposite l) (getM state) \leq elementLevel
        (opposite w1) (getM state)
        using assms
        using assertLiteralEffect[of state literal decision]
        by auto
        thus ?thesis
        using literalFalse w1 (elements (getM state))
        using elementLevelAppend[of opposite w1 getM state
        [[(literal, decision)]]
        using elementLevelAppend[of opposite l getM state
        [[(literal, decision)]]
        using assms
        using assertLiteralEffect[of state literal decision]
        by auto
        qed
     } thus ?thesis

487
by simp
qed

next
case False
with (?a state ∨ ?b state)
obtain l::Literal
  where l el (getF state ! c) literalTrue l (elements (getM state))
  elementLevel l (getM state) ≤ elementLevel (opposite w1) (getM state)
  by auto

from (w1 ≠ opposite literal):
  (literalFalse w1 (elements (getM ?state')))
  have elementLevel (opposite w1) ((getM state) @ [(literal, decision)]) =
    elementLevel (opposite w1) (getM state)
    using assms
    using assertLiteralEffect[of state literal decision]
    unfolding elementLevel-def
    by (simp add: markedElementsToAppend)
moreover
from (literalTrue l (elements (getM state))):
  have elementLevel l ((getM state) @ [(literal, decision)]) =
    elementLevel l (getM state)
    unfolding elementLevel-def
    by (simp add: markedElementsToAppend)
ultimately
  have elementLevel l ((getM state) @ [(literal, decision)]) ≤
    elementLevel (opposite w1) ((getM state) @ [(literal, decision)])
    using elementLevel l (getM state) ≤ elementLevel (opposite w1) (getM state)
    by simp
  thus ?thesis
    using (l el (getF state ! c)) : (literalTrue l (elements (getM state))):
      using assms
      using assertLiteralEffect[of state literal decision]
      by auto
qed

thus ?thesis
  unfolding watchCharacterizationCondition-def
  by auto
qed

moreover
  have watchCharacterizationCondition w2 w1 (getM ?state') ((getF ?state') ! c)
    proof
      {
**assume** literalFalse \(w_2\) (elements (getM \(?\text{state}'\)))

**with** \((w_2 \neq \text{opposite literal})\)

**have** literalFalse \(w_3\) (elements (getM state))

**using** assms

**using** assertLiteralEffect[of state literal decision]

by simp

**with** \((\text{watchCharacterizationCondition } w_2 \ w_1\) (getM state) \(\text{(getF state)} \ c\))

**have** \((\exists \ l. \ l \in ((\text{getF state}) \ ! c) \land \text{literalTrue } l \) (elements (getM state)))

\(\land \) elementLevel \(l\) (getM state) \(\leq\) elementLevel (opposite \(w_2\)) (getM state) \(\lor\)

\((\forall \ l. \ l \in ((\text{getF state}) \ c) \land l \neq w_2 \land l \neq w_1 \rightarrow\) literalFalse \(l\) (elements (getM state)) \(\land\)

elementLevel (opposite \(l\)) (getM state) \(\leq\) elementLevel (opposite \(w_2\)) (getM state)) (is \(?a\) state \(\lor\) \(?b\) state)

unfolding watchCharacterizationCondition-def

**using** assms

**using** assertLiteralEffect[of state literal decision]

**using** \((w_2 \neq \text{opposite literal})\)

by simp

**have** \(?a\) \(?\text{state}'\) \(\lor\) \(?b\) \(?\text{state}'\)

**proof** (cases \(?b\) state)

**case** True

**show** \(?\text{thesis}\)

**proof**−

{ fix \(l\)

**assume** \(l \in ((\text{getF} \ ?\text{state}'\) \ c) \land l \neq w_1 \land l \neq w_2\)

**have** literalFalse \(l\) (elements (getM \(?\text{state}'\))) \(\land\)

elementLevel (opposite \(l\)) (getM \(?\text{state}'\)) \(\leq\) elementLevel (opposite \(w_2\)) (getM \(?\text{state}'\))

**proof**−

from True \(\land l \in ((\text{getF} \ ?\text{state}') \ c) : (l \neq w_1) \land (l \neq w_2)\)

**have** literalFalse \(l\) (elements (getM state))

elementLevel (opposite \(l\)) (getM state) \(\leq\) elementLevel (opposite \(w_2\)) (getM state)

**using** assms

**using** assertLiteralEffect[of state literal decision]

by auto

thus \(?\text{thesis}\)

**using** \(\langle\text{literalFalse } w_2\) (elements (getM state))\)

**using** elementLevelAppend[of opposite \(w_2\) getM state [(\text{literal, decision})]]

**using** elementLevelAppend[of opposite \(l\) getM state [(\text{literal, decision})]]

**using** assms

**using** assertLiteralEffect[of state literal decision]
by auto
qed
}
thus \textit{thesis}
  by simp
qed
next
case False
with (?a state \lor ?b state)
obtain l::Literal
  where l el (getF state ! c) literalTrue l (elements (getM state))
  elementLevel l (getM state) \leq elementLevel (opposite w2) (getM state)
  by auto
  from \langle w2 \neq opposite literal \rangle
  \langle literalFalse w2 (elements (getM ?state')) \rangle
  have elementLevel (opposite w2) ((getM state) @ ([literal, decision])) = elementLevel (opposite w2) (getM state)
  using assms
  using assertLiteralEffect[of state literal decision]
  unfolding elementLevel-def
  by (simp add: markedElementsToAppend)
moreover
from \langle literalTrue l (elements (getM state)) \rangle
have elementLevel l ((getM state) @ ([literal, decision])) = elementLevel l (getM state)
  unfolding elementLevel-def
  by (simp add: markedElementsToAppend)
ultimately
have elementLevel l ((getM state) @ ([literal, decision])) \leq elementLevel (opposite w2) ((getM state) @ ([literal, decision]))
  using elementLevel l (getM state) \leq elementLevel (opposite w2) (getM state)
  by simp
  thus \textit{thesis}
      using \langle l el (getF state ! c) \rangle \langle literalTrue l (elements (getM state)) \rangle
  using assms
  using assertLiteralEffect[of state literal decision]
  by auto
qed
}
thus \textit{thesis}
  unfolding watchCharacterizationCondition-def
  by auto
qed
ultimately
show \(\text{thesis}\)
  by simp
qed

thus \(\text{thesis}\)
  unfolding InvariantWatchCharacterization-def
  by (simp add: Let-def)
qed

lemma assertLiteralConflictFlagEffect:
assumes
  InvariantConsistent ((\text{getM} \text{state}) \&\& [(\text{literal}, \text{decision})])
  InvariantUniq ((\text{getM} \text{state}) \&\& [(\text{literal}, \text{decision})])
  InvariantWatchListsContainOnlyClausesFromF (\text{getWatchList} \text{state})
    (\text{getF} \text{state})
  InvariantWatchListsUniq (\text{getWatchList} \text{state}) (\text{getWatch1} \text{state})
    (\text{getWatch2} \text{state})
  InvariantWatchesEl (\text{getF} \text{state}) (\text{getWatch1} \text{state}) (\text{getWatch2} \text{state})
  InvariantWatchCharacterization (\text{getF} \text{state}) (\text{getWatch1} \text{state})
    (\text{getWatch2} \text{state}) (\text{getM} \text{state})
shows
let \text{state}' = assertLiteral \text{literal} \text{decision} \text{state} in
  getConflictFlag \text{state}' = (getConflictFlag \text{state} \vee
  (\exists \text{clause}. \text{clause \in el (getF state)} \land
    opposite literal \text{el clause} \land
    clauseFalse clause ((\text{elements (getM state)} @ [\text{literal}])))
proof–
let \(?\text{state} = \text{state}[(\text{getM} := \text{getM state} @ [(\text{literal}, \text{decision})])]\)
let \(?\text{state}' = \text{assertLiteral \text{literal} \text{decision} \text{state}}\)

have
  getConflictFlag \?\text{state}' = (getConflictFlag \?\text{state} \vee
  (\exists \text{clause}. \text{clause \in set (getWatchList \?\text{state} \text{state}) (opposite literal)})
  \land
  clauseFalse (\text{nth (getF \?\text{state} clause)} (\text{elements (getM \?\text{state}))}))
using
  NotifyWatchesLoopConflictFlagEffect[of \?\text{state}
  getWatchList \?\text{state} (opposite literal)
  opposite literal \[]]
using
  InvariantConsistent ((\text{getM} \text{state}) \&\& [(\text{literal}, \text{decision})])
using
  InvariantWatchListsContainOnlyClausesFromF (\text{getWatchList} \text{state})
    (\text{getF} \text{state})
using
  InvariantWatchListsUniq (\text{getWatchList} \text{state})
using
  InvariantWatchListsCharacterization (\text{getWatchList} \text{state})
    (\text{getWatch1} \text{state}) (\text{getWatch2} \text{state})
using
  InvariantWatchesEl (\text{getF} \text{state}) (\text{getWatch1} \text{state}) (\text{getWatch2} \text{state})

491
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding assertLiteral-def
unfolding notifyWatches-def
by (simp add: Let-def)
moreover
have (∃ clause. clause ∈ set (getWatchList ?state (opposite literal))
∧
clauseFalse (nth (getF ?state) clause) (elements (getM ?state))) =
(∃ clause. clause el (getF state) ∧
opposite literal el clause ∧
clauseFalse clause ((elements (getM state)) @ [literal]))
(is ?lhs = ?rhs)
proof
assume ?lhs
then obtain clause
where clause ∈ set (getWatchList ?state (opposite literal))
clauseFalse (nth (getF ?state) clause) (elements (getM ?state))
by auto

have getWatch1 ?state clause = Some (opposite literal) ∨ getWatch2 ?state clause = Some (opposite literal)
clause < length (getF ?state)
∃ w1 w2. getWatch1 ?state clause = Some w1 ∧ getWatch2 ?state clause = Some w2 ∧
w1 el (nth (getF ?state) clause) ∧ w2 el (nth (getF ?state) clause)
using ⟨clause ∈ set (getWatchList ?state (opposite literal))⟩:
using assms
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchesEl-def
unfolding InvariantWatchListsCharacterization-def
by auto
hence (nth (getF ?state) clause) el (getF ?state)
opposite literal el (nth (getF ?state) clause)
using nth-mem[of clause getF ?state]
by auto
thus ?rhs
using ⟨clauseFalse (nth (getF ?state) clause) (elements (getM ?state))⟩
by auto
next
assume ?rhs
then obtain clause
where clause el (getF ?state)
opposite literal el clause
clauseFalse clause ((elements (getM state)) @ [literal])
by auto

492
then obtain \( ci \)
  where \( \text{clause} = \left( \text{nth} \ (\text{getF} \ ?\text{state}) \ ci \right) \ \text{ci} < \text{length} \ (\text{getF} \ ?\text{state}) \)
  by (auto simp add: in-set-conv-nth)
moreover
from \( \text{ci} < \text{length} \ (\text{getF} \ ?\text{state}) \)
obtain \( w1 \ w2 \)
where \( \text{getWatch1 state ci} = \text{Some w1} \)
\( \text{getWatch2 state ci} = \text{Some w2} \)
  using assms
unfolding InvariantWatchesEl-def
by auto
have \( \text{getWatch1 state ci} = \text{Some} \ (\text{opposite literal}) \lor \text{getWatch2 state ci} = \text{Some} \ (\text{opposite literal}) \)
proof
  { 
    assume \( \neg \ ?\text{thesis} \)
    with \( \langle \text{clauseFalse \ clause} \ ((\text{elements} \ (\text{getM state})) \ @ \ [\text{literal}]) \rangle \)
    \( \langle \text{clause} = \left( \text{nth} \ (\text{getF} \ ?\text{state}) \ ci \right) \rangle \)
    \( \langle \text{getWatch1 state ci} = \text{Some w1} \rangle \)
    \( \langle \text{getWatch2 state ci} = \text{Some w2} \rangle \)
    have \( \text{literalFalse w1} \ (\text{elements} \ (\text{getM state})) \)
    \( \langle \text{literalFalse w2} \ (\text{elements} \ (\text{getM state})) \rangle \)
    by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
  }
  from \( \langle \text{InvariantConsistent} \ ((\text{getM state}) \ @ \ [\text{lit}]) \rangle \)
  \( \langle \text{clauseFalse \ clause} \ ((\text{elements} \ (\text{getM state})) \ @ \ [\text{literal}]) \rangle \)
  have \( \neg (\exists \ l. \ l \ \text{el} \ \text{clause} \land \text{literalTrue l} \ (\text{elements} \ (\text{getM state}))) \)
  unfolding InvariantConsistent-def
  by (auto simp add: inconsistentCharacterization clauseFalseIffAllLiteralsAreFalse)
  from \( \langle \text{InvariantUniq} \ ((\text{getM state}) \ @ \ [\text{lit}]) \rangle \)
  have \( \neg \text{literalTrue} \ \text{literal} \ (\text{elements} \ (\text{getM state})) \)
  unfolding InvariantUniq-def
  by (auto simp add: uniqAppendIff)
  from \( \langle \text{InvariantWatchCharacterization} \ (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \ (\text{getM state}) \rangle \)
  \( \langle \text{literalFalse w1} \ (\text{elements} \ (\text{getM state})) \rangle \)
  \( \langle \text{literalFalse w2} \ (\text{elements} \ (\text{getM state})) \rangle \)
  \( \neg (\exists \ l. \ l \ \text{el} \ \text{clause} \land \text{literalTrue l} \ (\text{elements} \ (\text{getM state}))) \)
  \( \langle \text{getWatch1 state ci} = \text{Some w1} \rangle \)
  \( \langle \text{getWatch2 state ci} = \text{Some w2} \rangle \)
  \( \langle \text{ci} < \text{length} \ (\text{getF} \ ?\text{state}) \rangle \)
493
\(\text{clause} = (\text{nth} \ (\text{getF} \ ?\text{state}) \ \text{ci})\)

have\( \ \forall \ l, l \in \text{clause} \land l \neq w_1 \land l \neq w_2 \rightarrow \text{literalFalse} \ l\)

(elements \ (\text{getM} \ \text{state}))

unfolding \text{InvariantWatchCharacterization-def}

unfolding \text{watchCharacterizationCondition-def}

by auto

hence \text{literalTrue} \ \text{literal} \ (\text{elements} \ (\text{getM} \ \text{state}))

using \neg (\text{getWatch1} \ \text{state} \ \text{ci} = \text{Some} \ (\text{opposite literal}) \lor \text{getWatch2} \ \text{state} \ \text{ci} = \text{Some} \ (\text{opposite literal}))

using \langle \text{opposite literal} \ \text{el} \ \text{clause} \rangle

using \langle \text{getWatch1} \ \text{state} \ \text{ci} = \text{Some} \ \text{w_1} \rangle

using \langle \text{getWatch2} \ \text{state} \ \text{ci} = \text{Some} \ \text{w_2} \rangle

by auto

with \langle \neg \text{literalTrue} \ \text{literal} \ (\text{elements} \ (\text{getM} \ \text{state})) \rangle

have \text{False}

by simp

}\}

thus \ ?\text{thesis}

by auto

qed

ultimately

show \ ?\text{lhs}

using \text{assms}

using \langle \text{clauseFalse} \ \text{clause} \ (\text{elements} \ (\text{getM} \ \text{state})) \ @ \ [\text{literal}] \rangle

unfolding \text{InvariantWatchListsCharacterization-def}

by force

qed

ultimately

show \ ?\text{thesis}

by auto

qed

\textbf{lemma} \text{InvariantConflictFlagCharacterizationAfterAssertLiteral}: 

\textbf{assumes}

\text{InvariantConsistent} \ ((\text{getM} \ \text{state}) \ @ \ [(\text{literal}, \ \text{decision})])

\text{InvariantWatchesEl} \ (\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state})

and

\text{InvariantWatchesDiffer} \ (\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state}) \ \text{and}

\text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList} \ \text{state}) \ (\text{getF} \ \text{state}) \ \text{and}

\text{InvariantWatchListsUniq} \ (\text{getWatchList} \ \text{state}) \ \text{and}

\text{InvariantWatchListsCharacterization} \ (\text{getWatchList} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state})

\text{InvariantWatchesEl} \ (\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state})

and

\text{InvariantWatchCharacterization} \ (\text{getF} \ \text{state}) \ (\text{getWatch1} \ \text{state}) \ (\text{getWatch2} \ \text{state}) \ (\text{getM} \ \text{state})

\text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag} \ \text{state}) \ (\text{getF} \ \text{state})
state (getM state)

shows

let state' = (assertLiteral literal decision state) in
InvariantConflictFlagCharacterization (getConflictFlag state')
(getF state') (getM state')

proof

let ?state = state{getM := getM state @ [(l literal, d decision)]} in

have *: getConflictFlag ?state' = (getConflictFlag state ∨
(∃ clause. clause ∈ set (getWatchList ?state (opposite literal)))
 ∧
   clauseFalse (nth (getF ?state) clause) (elements (getM ?state)))
   using NotifyWatchesLoopConflictFlagEffect[of ?state
   getWatchList ?state (opposite literal)
   opposite literal []]
   using (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state))
   using (InvariantConsistent ((getM state) @ [(l literal, d decision)]))
   using (InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state))
   using (InvariantWatchListsUniq (getWatchList state))
   using (InvariantWatchListsCharacterization (getWatchList state)
   (getWatch1 state) (getWatch2 state))
   unfolding InvariantWatchListsUniq-def
   unfolding InvariantWatchListsCharacterization-def
   unfolding InvariantWatchListsContainOnlyClausesFromF-def
   unfolding assertLiteral-def
   unfolding notifyWatches-def
   by (simp add: Let-def)

hence getConflictFlag state → getConflictFlag ?state'
   by simp

show ?thesis
proof (cases getConflictFlag state)
case True
  thus ?thesis
  using :InvariantConflictFlagCharacterization (getConflictFlag state)
   (getF state) (getM state)
  using assertLiteralEffect[of state literal decision]
  using (getConflictFlag state → getConflictFlag ?state')
  using assms
  unfolding InvariantConflictFlagCharacterization-def
  by (auto simp add: Let-def formulaFalseAppendValiation)
next
case False
hence \( \neg \text{formulaFalse} \ (\text{getF state}) \ (\text{elements} \ (\text{getM state})) \)

using \( \text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag state}) \ (\text{getF state}) \ (\text{getM state}) \)

unfolding \( \text{InvariantConflictFlagCharacterization-def} \)
by simp

have \(*\): \( \forall \; \text{clause}. \ \text{clause} \notin \text{set} \ (\text{getWatchList ?state} \ (\text{opposite literal})) \wedge \)
\( 0 \leq \text{clause} \wedge \text{clause} < \text{length} \ (\text{getF ?state}) \rightarrow \neg \; \text{clauseFalse} \ (\text{nth} \ (\text{getF ?state}) \ \text{clause}) \ (\text{elements} \ (\text{getM ?state})) \)

proof -
{ fix \( \text{clause} \)
assume \( \text{clause} \notin \text{set} \ (\text{getWatchList ?state} \ (\text{opposite literal})) \)
and
\( 0 \leq \text{clause} \wedge \text{clause} < \text{length} \ (\text{getF ?state}) \)
from \( 0 \leq \text{clause} \wedge \text{clause} < \text{length} \ (\text{getF ?state}) \) obtain \( w1::\text{Literal} \) and \( w2::\text{Literal} \)
where \( \text{getWatch1 ?state} \ \text{clause} = \text{Some} \ w1 \) and
\( \text{getWatch2 ?state} \ \text{clause} = \text{Some} \ w2 \) and
\( w1 \ \text{el} \ (\text{nth} \ (\text{getF ?state}) \ \text{clause}) \) and
\( w2 \ \text{el} \ (\text{nth} \ (\text{getF ?state}) \ \text{clause}) \)
using \( \text{InvariantWatchesEl} \ (\text{getF state}) \ (\text{getWatch1 state}) \)
(\getWatch2 state)
unfolding \( \text{InvariantWatchesEl-def} \)
by auto

have \( \neg \; \text{clauseFalse} \ (\text{nth} \ (\text{getF ?state}) \ \text{clause}) \ (\text{elements} \ (\text{getM ?state})) \)
proof -
from \( \text{clause} \notin \text{set} \ (\text{getWatchList ?state} \ (\text{opposite literal})) \)
have \( w1 \neq \text{opposite literal} \) and
\( w2 \neq \text{opposite literal} \)
using \( \text{InvariantWatchListsCharacterization} \ (\text{getWatchList state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \)
using \( \text{getWatch1 ?state} \ \text{clause} = \text{Some} \ w1 \) and \( \text{getWatch2 ?state} \ \text{clause} = \text{Some} \ w2 \)
unfolding \( \text{InvariantWatchListsCharacterization-def} \)
by auto

from \( \neg \; \text{formulaFalse} \ (\text{getF state}) \ (\text{elements} \ (\text{getM state})) \)
have \( \neg \; \text{clauseFalse} \ (\text{nth} \ (\text{getF ?state}) \ \text{clause}) \ (\text{elements} \ (\text{getM state})) \)
using \( 0 \leq \; \text{clause} \wedge \; \text{clause} < \text{length} \ (\text{getF ?state}) \)
by (simp add: \( \text{formulaFalseIffContainsFalseClause} \))
show \(\text{?thesis}\)

proof (cases literalFalse \(w1\) (elements (getM state)) \(\lor\) literalFalse \(w2\) (elements (getM state)))

  case True

  with (InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state))
  have \(\exists l. l \in (\text{nth (getF state) clause}) \land \text{literalTrue} l \land (\text{elements (getM state)}) \lor \forall l. l \in (\text{nth (getF state) clause}) \land l \neq w1 \land l \neq w2 \rightarrow \text{literalFalse} l \land (\text{elements (getM state)})\)
    using \(\text{getWatch1 ?state clause = Some w1[THEN sym]}\)
    using \(\text{getWatch2 ?state clause = Some w2[THEN sym]}\)
    using \(\theta \leq \text{clause }\land\text{ clause} < \text{length (getF ?state)}\)
    unfolding InvariantWatchCharacterization-def
    unfolding watchCharacterizationCondition-def
    by auto

  thus \(\text{?thesis}\)

proof (cases \(\forall l. l \in (\text{nth (getF state) clause}) \land l \neq w1 \land l \neq w2 \rightarrow \text{literalFalse} l \land (\text{elements (getM state)})\))

  case True

  have \(\neg \text{literalFalse} w1 \land (\text{elements (getM state)}) \lor \neg \text{literalFalse} w2 \land (\text{elements (getM state)})\)
    proof
      from \(\neg \text{clauseFalse} (\text{nth (getF ?state) clause})\) (elements (getM state))
      obtain \(l::\text{Literal}\)
      where \(l \in (\text{nth (getF ?state) clause}) \land \neg \text{literalFalse} l \land (\text{elements (getM state)})\)
      by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
      with True
      show \(\text{?thesis}\)
      by auto
      qed
    hence \(\neg \text{literalFalse} w1 \land (\text{elements (getM ?state)}) \lor \neg \text{literalFalse} w2 \land (\text{elements (getM ?state)})\)
      using \(\neg \text{opposite literal}\) \(\land\) \(\neg \text{opposite literal}\)
      by auto
    thus \(\text{?thesis}\)
      using \(\neg \text{clauseFalse} (\text{nth (getF ?state) clause})\) \(\land\) \(\neg \text{clauseFalse} (\text{nth (getF ?state) clause})\)
      by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
next
  case False
  then obtain \(l::\text{Literal}\)
where \( l \in \text{el} (\text{nth} (\text{getF} \ ?\text{state}) \ \text{clause}) \) and \( \text{literalTrue} \ l \)
\( (\text{elements} (\text{getM} \ ?\text{state})) \)
using $\$
by auto
thus \(?\text{thesis}\)
using \langle \text{InvariantConsistent} ((\text{getM} \ ?\text{state}) @ [(\text{literal}, \text{decision})]) \rangle
unfolding \text{InvariantConsistent-def} 
by (auto simp add: \text{clauseFalseIffAllLiteralsAreFalse} \text{inconsistentCharacterization})
qed
next
case False
thus \(?\text{thesis}\)
using \langle \text{w1 el} (\text{nth} (\text{getF} \ ?\text{state}) \ \text{clause}) \rangle and 
\langle \text{w1} \neq \ \text{opposite literal} \rangle
by (auto simp add: \text{clauseFalseIffAllLiteralsAreFalse})
qed
qed

show \(?\text{thesis}\)
proof (cases \text{getConflictFlag} \ ?\text{state}‟)
case True
from \( \langle \neg \text{getConflictFlag} \ ?\text{state} \rangle \langle \text{getConflictFlag} \ ?\text{state}‟ \rangle \)
obtain \text{clause}:nat
where
\text{clause} \in \text{set} (\text{getWatchList} \ ?\text{state} (\text{opposite literal})) and 
\text{clauseFalse} (\text{nth} (\text{getF} \ ?\text{state}) \ \text{clause}) (\text{elements} (\text{getM} \ ?\text{state}))
using $\$
by auto
from \langle \text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList} \ ?\text{state} \ (\text{getF} \ ?\text{state})) \ (\text{getF} \ ?\text{state}) \rangle
\langle \text{clause} \in \text{set} (\text{getWatchList} \ ?\text{state} (\text{opposite literal})) \rangle
have \( \text{nth} (\text{getF} \ ?\text{state}) \ \text{clause} \in \text{el} (\text{getF} \ ?\text{state}) \)
unfolding \text{InvariantWatchListsContainOnlyClausesFromF-def} 
using \text{nth-mem}
by simp
with \langle \text{clauseFalse} (\text{nth} (\text{getF} \ ?\text{state}) \ \text{clause}) \ (\text{elements} (\text{getM} \ ?\text{state})) \rangle
have \text{formulaFalse} (\text{getF} \ ?\text{state}) \ (\text{elements} (\text{getM} \ ?\text{state}))
by (auto simp add: \text{Let-def formulaFalseIffContainsFalseClause})
thus \(?\text{thesis}\)
using \( \langle \neg \text{getConflictFlag} \ ?\text{state} \rangle \langle \text{getConflictFlag} \ ?\text{state}‟ \rangle \)
unfolding \text{InvariantConflictFlagCharacterization-def} 
using \text{assms}
using assertLiteralEffect[of state literal decision]
by (simp add: Let-def)

next

case False
hence ∀ clause::nat. clause ∈ set (getWatchList ?state (opposite literal)) −→
¬ clauseFalse (nth (getF ?state) clause) (elements (getM ?state))
using *
by auto

with **
have ∀ clause. 0 ≤ clause ∧ clause < length (getF ?state) −→
¬ clauseFalse (nth (getF ?state) clause) (elements (getM ?state))
by (auto simp add: set-conv-nth formulaFalseIffContainsFalse-Clause)

thus ?thesis
using (¬ getConflictFlag state) (¬ getConflictFlag ?state')
using assms
unfolding InvariantConflictFlagCharacterization-def
by (auto simp add: Let-def assertLiteralEffect)

qed

lemma InvariantConflictClauseCharacterizationAfterAssertLiteral:
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
InvariantWatchListsUniq (getWatchList state)
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)

shows
let state' = assertLiteral literal decision state in
InvariantConflictClauseCharacterization (getConflictFlag state') (getConflictClause state') (getF state') (getM state')

proof —
let ?state0 = state{ getM := getM state @ [(literal, decision)]}
show ?thesis
using assms
unfolding InvariantConflictClauseCharacterizationAfterNotifyWatches[of ?state0 getM state opposite literal decision
getWatchList ?state0 (opposite literal) []]
unfolding assertLiteral-def

499
lemma assertLiteralQEffect:
assumes
  InvariantConsistent ((getM state) @ [(literal, decision)])
  InvariantUniq ((getM state) @ [(literal, decision)])
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  InvariantWatchListsUniq (getWatchList state) (getWatch1 state) (getWatch2 state)
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
shows
let state' = assertLiteral literal decision state in
set (getQ state') = set (getQ state) ∪
  { ul. (∃ uc. uc el (getF state) ∧
          opposite literal el uc ∧
          isUnitClause uc ul ((elements (getM state)) @ [literal])) }
(is let state' = assertLiteral literal decision state in
set (getQ state') = set (getQ state) ∪ ?ulSet)
proof−
  let ?state' = state([getM := getM state @ [(literal, decision)]])
  let ?state'' = assertLiteral literal decision state
  have set (getQ ?state'') − set (getQ state) ⊆ ?ulSet
    unfolding assertLiteral-def
    unfolding notifyWatches-def
    using assms
    using NotifyWatchesLoopQEffect[of ?state' getM state opposite literal decision getWatchList ?state' (opposite literal) []]
    unfolding InvariantWatchListsCharacterization-def
    unfolding InvariantWatchListsUniq-def
    unfolding InvariantWatchListsContainOnlyClausesFromF-def
    using set-com-nth[of getF state]
    by (auto simp add: Let-def)
  moreover
  have ?ulSet ⊆ set (getQ ?state'')
  proof
  qed
fix \( ul \)
assume \( ul \in \mathcal{ulSet} \)
then obtain \( uc \)
  where \( uc \in (\text{getF state}) \) opposite literal el uc isUnitClause uc
  \( ul \) ((\( (\text{elements} (\text{getM state})) \) @ [\( \text{l literal} \)])
    by auto
then obtain \( uci \)
  where \( uci = (\text{nth} (\text{getF state}) uci) \) uci < length (\( \text{getF state} \))
  using set-conv-nth[of getF state]
  by auto
let \( ?w1 = \text{getWatch1 state} \) uci
let \( ?w2 = \text{getWatch2 state} \) uci

have \( ?w1 = \text{Some (opposite literal)} \lor ?w2 = \text{Some (opposite literal)} \)
proof-
{
  assume \( \neg \) ?thesis

  from (\( \text{InvariantWatchesEl} \) (\( \text{getF state} \)) (\( \text{getWatch1 state} \))
    (\( \text{getWatch2 state} \))
  obtain \( \text{wl1} \) \( \text{wl2} \)
    where \( ?w1 = \text{Some \text{wl1}} \) \( ?w2 = \text{Some \text{wl2}} \) \( \text{el (\text{getF state}} \)
      \( \!\) uci) \( \text{wl2} \) el (\( \text{getF state} \) ! uci)
    unfolding \( \text{InvariantWatchesEl-def} \)
    using (\( \text{uci < length (getF state)} \))
    by force
with (\( \text{InvariantWatchCharacterization} \) (\( \text{getF state} \)) (\( \text{getWatch1 state} \))
    (\( \text{getWatch2 state} \)) (\( \text{getM state} \))
  have watchCharacterizationCondition \( \text{wl1} \) \( \text{wl2} \) (\( \text{getM state} \))
    (\( \text{getF state} \) ! uci)
    watchCharacterizationCondition \( \text{wl2} \) \( \text{wl1} \) (\( \text{getM state} \)) (\( \text{getF state} \) ! uci)
    using (\( \text{uci < length (getF state)} \))
    unfolding \( \text{InvariantWatchCharacterization-def} \)
    by auto

  from (isUnitClause uc ul ((\( \text{elements (getM state)}) @ [\( \text{l literal} \)])
    have \( \neg (\exists \text{l. l el uc} \land (\text{literalTrue l ((\( \text{elements (getM state)})
      @ [\( \text{l literal} \)]))}
    using containsTrueNotUnit
    using (\( \text{InvariantConsistent (getM state) @ [\( \text{decision} \)]]})
    unfolding \( \text{InvariantConsistent-def} \)
    by auto
  from (\( \text{InvariantUniq (getM state) @ [\( \text{decision} \)]]})
  have \( \neg \) literal el (\( \text{elements (getM state)} \))
unfolding $\text{InvariantUniq-def}$
by (simp add: uniqAppendIff)

from ($\neg \ ?thesis$)
($?w1 = \text{Some } w1$) ($?w2 = \text{Some } w2$)
have $w1 \neq \text{ opposite literal } w2 \neq \text{ opposite literal}$
by auto

from $\langle \text{InvariantWatchesDiffer } (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \rangle$
have $w1 \neq w2$
using ($?w1 = \text{Some } w1$) ($?w2 = \text{Some } w2$)
unfolding $\text{InvariantWatchesDiffer-def}$
using ($\text{uci} < \text{length } (\text{getF state})$)
by auto

have $\text{literalFalse } w1 \ (\text{elements } (\text{getM state})) \lor \text{literalFalse } w2 \ (\text{elements } (\text{getM state}))$
proof (cases $ul = w1$)
case True
with ($?w1 \neq w2$)
have $ul \neq w2$
by simp
with $\langle \text{isUnitClause } uc \ ul \ ((\text{elements } (\text{getM state})) @ [\text{literal}]) \rangle$
$\langle w2 \neq \text{ opposite literal } w1 \ el \ (\text{getF state} ! \text{uci}) \rangle$
$\langle uc = (\text{getF state} ! \text{uci}) \rangle$
show $\ ?thesis$
unfolding $\text{isUnitClause-def}$
by auto
next
case False
with $\langle \text{isUnitClause } uc \ ul \ ((\text{elements } (\text{getM state})) @ [\text{literal}]) \rangle$
$\langle w1 \neq \text{ opposite literal } w1 \ el \ (\text{getF state} ! \text{uci}) \rangle$
$\langle uc = (\text{getF state} ! \text{uci}) \rangle$
show $\ ?thesis$
unfolding $\text{isUnitClause-def}$
by auto
qed

with $\langle \text{watchCharacterizationCondition } w1 \ w2 \ (\text{getM state}) \ (\text{getF state} ! \text{uci}) \rangle$
$\langle \text{watchCharacterizationCondition } w2 \ w1 \ (\text{getM state}) \ (\text{getF state} ! \text{uci}) \rangle$
$\langle \neg \ (\exists \ l. \ l \ el \ uc \ \land \ (\text{literalTrue } l \ ((\text{elements } (\text{getM state})) @ [\text{literal}])))$
$\langle uc = (\text{getF state} ! \text{uci}) \rangle$
$\langle ?w1 = \text{Some } w1 \rangle$ ($?w2 = \text{Some } w2$)
have $\forall \ l. \ l \ el \ uc \ \land \ l \neq w1 \ \land \ l \neq w2 \rightarrow \text{literalFalse } l \ (\text{elements } (\text{getM state}))$
unfolding \textit{watchCharacterizationCondition-def}
\begin{verbatim}
by auto
with \langle \forall i \neq \textit{opposite literal}, \forall j \neq \textit{opposite literal} \rangle
\langle \textit{opposite literal el uc} \rangle
have \textit{literalTrue literal \langle elements (getM state) \rangle}
by auto
with \langle \neg \textit{literal el \langle elements (getM state) \rangle} \rangle
have \textit{False}
by simp
\end{verbatim}
\} thus \textit{?thesis}
by auto
qed

with \langle \textit{InvariantWatchListsCharacterization (getWatchList state)} \textit{(getWatch1 state)} \textit{(getWatch2 state)} \rangle
have \textit{aci} \in \textit{set (getWatchList state \langle opposite literal \rangle)}
unfolding \textit{InvariantWatchListsCharacterization-def}
by auto
thus \textit{ul} \in \textit{set (getQ ?state'')}
using \langle \textit{uc el (getF state)} \rangle
using \langle \textit{isUnitClause uc ul \langle \langle \textit{elements (getM state)} \rangle \ @ \langle \textit{literal} \rangle \rangle} \rangle
using \langle \textit{uc = \langle getF state \rangle \at \langle uci \rangle} \rangle
unfolding \textit{assertLiteral-def}
unfolding \textit{notifyWatches-def}
using \textit{assms}
using \textit{NotifyWatchesLoopQEffect[of state literal decision getWatchList ?state' (opposite literal)] []}
unfolding \textit{InvariantWatchListsCharacterization-def}
unfolding \textit{InvariantWatchListsUniq-def}
unfolding \textit{InvariantWatchListsContainOnlyClausesFromF-def}
\begin{verbatim}
by (auto simp add: Let-def)
\end{verbatim}
qed

moreover
have \textit{set (getQ state) \subseteq set (getQ ?state'')}
using \textit{assms}
using \textit{assertLiteralEffect[of state literal decision]}
using \textit{prefixIsSubset[of getQ state getQ ?state'']}
\textit{by simp}
ultimately
show \textit{?thesis}
by (auto simp add: Let-def)
qed

\textbf{lemma} \textit{InvariantQCharacterizationAfterAssertLiteral:}
\begin{verbatim}
assumes
    \textit{InvariantConsistent \langle \langle getM state \rangle \ @ \langle \langle \textit{literal, decision} \rangle \rangle} \rangle
    \textit{InvariantWatchListsContainOnlyClausesFromF \langle getWatchList state \rangle}
    \langle getF state \rangle \textit{and}
\end{verbatim
InvariantWatchListsUniq \( \text{getWatchList state} \) and
InvariantWatchListsCharacterization \( \text{getWatchList state} \) \( \text{getWatch1 state} \) \( \text{getWatch2 state} \)
and
InvariantWatchesEl \( \text{getF state} \) \( \text{getWatch1 state} \) \( \text{getWatch2 state} \)
and
InvariantWatchesDiffer \( \text{getF state} \) \( \text{getWatch1 state} \) \( \text{getWatch2 state} \)
and
InvariantConflictFlagCharacterization \( \text{getConflictFlag state} \) \( \text{getF state} \) \( \text{getM state} \)
InvariantQCharacterization \( \text{getConflictFlag state} \) \( \text{getQ state} \) \( \text{getF state} \) \( \text{getM state} \)
shows
let state' = (assertLiteral literal decision state)
in
InvariantQCharacterization \( \text{getConflictFlag state'} \) \( \text{getF state'} \) \( \text{getM state'} \)

proof—
let ?state = state[getM := getM state @ [(literal, decision)]]
let ?state' = assertLiteral literal decision state

have \( \forall l. l \in \text{set (getQ ?state')} \rightarrow \text{set (getQ ?state)} \rightarrow \)
(\( \exists \text{clause. clause el (getF ?state) \& isUnitClause clause l (elements (getM ?state))} \))
using NotifyWatchesLoopQEffect[of ?state getM state opposite literal decision getWatchList ?state (opposite literal)]
using assms
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchCharacterization-def
unfolding assertLiteral-def
unfolding notifyWatches-def
by (auto simp add: Let-def)

have \( \forall \text{clause. clause } \in \text{set (getWatchList ?state (opposite literal))} \rightarrow \)
\( \forall l. \text{(isUnitClause (nth (getF ?state) clause) l (elements (getM ?state)))} \rightarrow l \in \text{set (getQ ?state')}) \)
using NotifyWatchesLoopQEffect[of ?state getM state opposite literal decision getWatchList ?state (opposite literal)]
using assms
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchCharacterization-def
unfolding assertLiteral-def
unfolding notifyWatches-def

504
by (simp add: Let-def)

have getConflictFlag state \rightarrow getConflictFlag ?state'
proof -
  have getConflictFlag ?state' = (getConflictFlag state \lor
  (\exists \text{ clause}. \text{ clause} \in \text{ set} \text{ (getWatchList ?state \text{ (opposite literal)})})
  \land
  \text{ clauseFalse (nth (getF ?state) clause) (elements (getM ?state))})
  using NotifyWatchesLoopConflictFlagEffect[of ?state
  getWatchList ?state (opposite literal)
  opposite literal []]
  using assms
  unfolding InvariantWatchListsUniq-def
  unfolding InvariantWatchListsCharacterization-def
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  unfolding assertLiteral-def
  unfolding notifyWatches-def
  by (simp add: Let-def)
thus ?thesis
by simp
qed

{ assume \neg \text{ getConflictFlag ?state'
  with (\text{ getConflictFlag state \rightarrow getConflictFlag ?state'})
  have \neg \text{ getConflictFlag state}
    by simp

  have \forall l. l el (removeAll literal (getQ ?state')) =
    (\exists c. c el (getF ?state') \land isUnitClause c l (elements (getM ?state')))
    proof
    fix l ::\text{Literal}
    show l el (removeAll literal (getQ ?state')) =
      (\exists c. c el (getF ?state') \land isUnitClause c l (elements (getM ?state')))
      proof
      assume l el (removeAll literal (getQ ?state'))
      hence l el (getQ ?state') l \neq literal
        by auto
      show \exists c. c el (getF ?state') \land isUnitClause c l (elements (getM ?state'))
        proof (cases l el (getQ state))
        case True

    from (\neg \text{ getConflictFlag state)
        :InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)}

505
\[ l \in (\text{getQ state}) \]

obtain \( c \in \text{Clause} \)
where \( c \in (\text{getF state}) \) isUnitClause \( c \) \( l \) \( (\text{elements} \ (\text{getM state})) \)

unfolding InvariantQCharacterization-def
by auto

show \( \text{?thesis} \)
proof (cases \( l \neq \) opposite literal)
  case True
  hence opposite \( l \neq \) literal
  by auto

  from (isUnitClause \( c \) \( l \) \( (\text{elements} \ (\text{getM state})) \))
    \( (\text{opposite} \ l \neq \) literal) \( (l \neq \) literal)
  have isUnitClause \( c \) \( l \) \( ((\text{elements} \ (\text{getM state})) @ [\text{literal}]) \)
    using isUnitClauseAppendValuation\( [c \ l \ \text{elements} \ (\text{getM state}) \ \text{l literal}] \)
    by simp
  thus \( \text{?thesis} \)
    using assms
    using assertLiteralEffect\( [\text{of state literal decision}] \)
    by auto

next
  case False
  hence opposite \( l = \) literal
  by simp

  from (isUnitClause \( c \) \( l \) \( (\text{elements} \ (\text{getM state})) \))
  have clauseFalse \( c \) \( (\text{elements} \ (\text{getM ?state'}) \))
    using assms
    using assertLiteralEffect\( [\text{of state literal decision}] \)
    using unitBecomesFalse\( [c \ l \ \text{elements} \ (\text{getM state})] \)
    by simp
  with \( (c \ l \ (\text{getF state})) \)
  have formulaFalse \( (\text{getF state}) \) \( (\text{elements} \ (\text{getM ?state'}) \))
    by (auto simp add: formulaFalseIfContainsFalseClause)

  from assms
  have InvariantConflictFlagCharacterization \( (\text{getConflictFlag ?state'}) \) \( (\text{getF ?state'}) \) \( (\text{getM ?state'}) \)
    using InvariantConflictFlagCharacterizationAfterAssertLiteral
    by (simp add: Let-def)
  with \( \) \( \) \( \) \( formulaFalse \) \( (\text{getF state}) \) \( (\text{elements} \ (\text{getM ?state'}) \))
  have getConflictFlag \( ?\text{state'} \)
    using assms
    using assertLiteralEffect\( [\text{of state literal decision}] \)

506
unfold invariantConflictedFlagCharacterization-def
by auto
with \( \neg \text{getConflictFlag} \ ?\text{state}' \)
show \(?\text{thesis}\)
by simp
qed
next
case False
then obtain \(c::\text{Clause}\)
where \(c \ \text{el} \ \text{getF} \ ?\text{state}' \land \text{isUnitClause} \ c \ l \ \text{elements} \ \text{getM} \ ?\text{state}'\)
using *
using \(l \ \text{el} \ \text{getQ} \ ?\text{state}'\)
using assms
using assertLiteralEffect[of state literal decision]
by auto
thus \(?\text{thesis}\)
using formulaEntailsItsClauses[of \ c \ \text{getF} \ ?\text{state}']
by auto
qed
next
assume \(\exists \ c. \ c \ \text{el} \ \text{getF} \ ?\text{state}' \land \text{isUnitClause} \ c \ l \ \text{elements} \ \text{getM} \ ?\text{state}'\)
then obtain \(c::\text{Clause}\)
where \(c \ \text{el} \ \text{getF} \ ?\text{state}' \land \text{isUnitClause} \ c \ l \ \text{elements} \ \text{getM} \ ?\text{state}'\)
by auto
then obtain \(ci::\text{nat}\)
where \(0 \leq ci < \text{length} \ \text{getF} \ ?\text{state}' \ c = (\text{nth} \ \text{getF} \ ?\text{state}') \ ci\)
using set-conv-nth[of \ getF \ ?\text{state}']
by auto
then obtain \(w1::\text{Literal} \ \text{and} \ w2::\text{Literal}\)
where \(\text{getWatch1} \ ?\text{state} \ ci = \text{Some} \ w1 \ \text{and} \ \text{getWatch2} \ ?\text{state} \)
\(ci = \text{Some} \ w2 \ \text{and} \ w1 \ \text{el} \ c \ \text{and} \ w2 \ \text{el} \ c\)
using \(\text{InvariantWatchesEl} \ \text{(getF} \ ?\text{state}) \ \text{(getWatch1} \ ?\text{state}) \ \text{(getWatch2} \ ?\text{state})\)
using \(c = (\text{nth} \ \text{getF} \ ?\text{state}') \ ci\)
unfolding InvariantWatchesEl-def
using assms
using assertLiteralEffect[of state literal decision]
by auto
hence \(w1 \neq w2\)
using \(ci < \text{length} \ \text{getF} \ ?\text{state}'\)
using \(\text{InvariantWatchesDiffer} \ \text{(getF} \ ?\text{state}) \ \text{(getWatch1} \ ?\text{state}) \ \text{(getWatch2} \ ?\text{state})\)
unfolding InvariantWatchesDiffer-def
using assms
using assertLiteralEffect[of state literal decision]
by auto

show l el (removeAll literal (getQ ?state'))
proof (cases isUnitClause c l (elements (getM state)))
case True
with InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
(¬ getConflictFlag state)
⟨c el (getF ?state')⟩
have l el (getQ state)
using assms
using assertLiteralEffect[of state literal decision]
unfolding InvariantQCharacterization-def
by auto
have isPrefix (getQ state) (getQ ?state')
using assms
using assertLiteralEffect[of state literal decision]
by simp
then obtain Q'
where (getQ state) @ Q' = (getQ ?state')
unfolding isPrefix-def
by auto
have l el (getQ ?state')
using (l el (getQ state))
((getQ state) @ Q' = (getQ ?state')[THEN sym])
by simp
moreover
have l ≠ literal
using (isUnitClause c l (elements (getM ?state')))
using assms
using assertLiteralEffect[of state literal decision]
unfolding isUnitClause-def
by simp
ultimately
show ?thesis
by auto
next
case False

thus ?thesis
proof (cases ci ∈ set (getWatchList ?state (opposite literal)))
case True
with **
⟨isUnitClause c l (elements (getM ?state'))⟩
⟨c = (nth (getF ?state') ci)⟩
have l ∈ set (getQ ?state')
using assms
using assertLiteralEffect[of state literal decision]
by simp moreover
have \( l \neq \text{ literal} \)
using \( \text{ isUnitClause } c \ l \ (\text{ elements } \ (\text{ getM } \ ?\text{state}')) \)
unfolding isUnitClause-def
using assms
using assertLiteralEffect[of state literal decision]
by simp
ultimately
show \( ?\text{thesis} \)
by simp

next
case \( \text{False} \)
with \( \text{InvariantWatchListsCharacterization } (\text{ getWatchList state }) (\text{ getWatch1 state }) (\text{ getWatch2 state }) \)
have \( w1 \neq \text{ opposite literal} \ w2 \neq \text{ opposite literal} \)
using \( (\text{ getWatch1 state } \ ci = \text{ Some } w1) \ \text{ and } (\text{ getWatch2 state } \ ci = \text{ Some } w2) \)
unfolding InvariantWatchListsCharacterization-def
by auto
have \( \text{ literalFalse } w1 \ (\text{ elements } \ (\text{ getM state })) \lor \text{ literalFalse } w2 \ (\text{ elements } \ (\text{ getM state })) \)
proof−
{ assume \( \neg \ ?\text{thesis} \)
    hence \( \neg \ \text{ literalFalse } w1 \ (\text{ elements } \ (\text{ getM state }')) \lor \neg \ \text{ literalFalse } w2 \ (\text{ elements } \ (\text{ getM state }')) \)
    using \( (w1 \neq \text{ opposite literal}) \ \text{ and } (w2 \neq \text{ opposite literal}) \)
    using assms
    using assertLiteralEffect[of state literal decision]
    by auto
    with \( (w1 \neq w2) \ (w1 \ \text{ el } c) \ (w2 \ \text{ el } c) \)
    have \( \neg \ \text{ isUnitClause } c \ l \ (\text{ elements } \ (\text{ getM state }')) \)
    unfolding isUnitClause-def
    by auto
}
with \( \text{ isUnitClause } c \ l \ (\text{ elements } \ (\text{ getM state }')) \)
show \( ?\text{thesis} \)
by auto
qed

with \( \text{InvariantWatchCharacterization } (\text{ getF state }) (\text{ getWatch1 state }) (\text{ getWatch2 state }) (\text{ getM state }) \)
have \( \$: (\exists \ l. \ l \ \text{ el } c \ \land \ \text{ literalTrue } l \ (\text{ elements } \ (\text{ getM state }))) \lor \\
(\forall \ l. \ l \ \text{ el } c \ \land \ l \neq w1 \ \land \ l \neq w2 \ \rightarrow \ \text{ literalFalse } l \ (\text{ elements } \ (\text{ getM state }))) \)
using $\langle \text{ci} < \text{length (getF ?state')} \rangle$:
using $\langle \text{c} = (\text{nth (getF ?state')} \text{ ci}) \rangle$
using $\langle \text{getWatch1 state ci = Some w1} \rangle$ [THEN sym] and
$\langle \text{getWatch2 state ci = Some w2} \rangle$ [THEN sym]
using assms
using assertLiteralEffect [of state literal decision]
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
by auto
thus ?thesis
proof (cases $\forall \ l. \ l \in c \land l \neq w1 \land l \neq w2 \longrightarrow \text{literalFalse} l$ (elements (getM state))):
case True
with $\langle \text{isUnitClause c l (elements (getM ?state')} \rangle$)
have \text{literalFalse w1 (elements (getM state)) $\longrightarrow \neg \text{literalFalse w2 (elements (getM state)) \land \neg \text{literalTrue w2 (elements (getM state)) \land l = w2}$
 \text{literalFalse w2 (elements (getM state)) $\longrightarrow \neg \text{literalFalse w1 (elements (getM state)) \land \neg \text{literalTrue w1 (elements (getM state)) \land l = w1}$
unfolding isUnitClause-def
using assms
using assertLiteralEffect [of state literal decision]
by auto

with $\langle \text{literalFalse w1 (elements (getM state)) \lor \text{literalFalse w2 (elements (getM state))} \rangle$
have $\langle \text{literalFalse w1 (elements (getM state)) \land \neg \text{literalFalse w2 (elements (getM state)) \land \neg \text{literalTrue w2 (elements (getM state)) \land l = w2} \lor \text{literalFalse w2 (elements (getM state)) \land \neg \text{literalFalse w1 (elements (getM state)) \land \neg \text{literalTrue w1 (elements (getM state)) \land l = w1}$
by blast
hence $\langle \text{isUnitClause c l (elements (getM state))} \rangle$
using $\langle \text{w1 el c \land w2 el c \land True} \rangle$
unfolding isUnitClause-def
by auto
thus ?thesis
using $\langle \neg \text{isUnitClause c l (elements (getM state))} \rangle$
by simp
next
case False
then obtain $l' :: \text{Literal where}$
$\langle l' el c \land \text{literalTrue l' (elements (getM state))} \rangle$
using $\langle c \rangle$
by auto
hence $\langle \text{literalTrue l' (elements (getM ?state')} \rangle$
using assms

510
using \text{assertLiteralEffect}\{\text{of state literal decision}\}
by auto

from \langle \text{InvariantConsistent}\ ((\text{getM}\ \text{state}) @ [(\text{literal, decision})])\rangle:
\langle l'\ el c\ \text{literalTrue} l'\ (\text{elements}\ (\text{getM}\ ?\text{state}'))\rangle
show \ ?\text{thesis}
using \text{containsTrueNotUnit}\{of l'\ c\ \text{elements}\ (\text{getM}\ ?\text{state}')\}
using \text{isUnitClause}\ c\ l\ (\text{elements}\ (\text{getM}\ ?\text{state}'))
using \text{assms}
using \text{assertLiteralEffect}\{\text{of state literal decision}\}
unfolding \text{InvariantConsistent-def}
by auto
qed

\}
thus \ ?\text{thesis}
unfolding \text{InvariantQCharacterization-def}
by simp
qed

\text{lemma}\ \text{AssertLiteralStartQIrelevant}:\
\text{fixes}\ \text{literal}::\ \text{Literal}\ \text{and}\ \text{Wl}::\ \text{nat list}\ \text{and}\ \text{newWl}::\ \text{nat list}\ \text{and}\ \text{state}::\ \text{State}\n\text{assumes}\ \text{InvariantWatchesEl}\ (\text{getF}\ \text{state})\ (\text{getWatch1}\ \text{state})\ (\text{getWatch2}\ \text{state})\n\text{and}\ \text{InvariantWatchListsContainOnlyClausesFromF}\ (\text{getWatchList}\ \text{state})\ (\text{getF}\ \text{state})\n\text{shows}\ \text{let}\ \text{state}' = (\text{assertLiteral}\ \text{literal}\ \text{decision}\ (\text{state}[\text{getQ} := Q' \}])))\ \text{in}\n\text{let}\ \text{state}'' = (\text{assertLiteral}\ \text{literal}\ \text{decision}\ (\text{state}[\text{getQ} := Q'' \}])))\ \text{in}\n(\text{getM}\ \text{state}') = (\text{getM}\ \text{state}'')\ \land\n(\text{getF}\ \text{state}') = (\text{getF}\ \text{state}'')\ \land\n(\text{getSATFlag}\ \text{state}') = (\text{getSATFlag}\ \text{state}'')\ \land\n(\text{getConflictFlag}\ \text{state}') = (\text{getConflictFlag}\ \text{state}'')\n
\text{using}\ \text{assms}
\text{unfolding}\ \text{assertLiteral-def}
\text{unfolding}\ \text{notifyWatches-def}
\text{unfolding}\ \text{InvariantWatchListsContainOnlyClausesFromF-def}
\text{using}\ \text{notifyWatchesStartQIrelevant}\{\text{of}\ \text{state}[\text{getQ} := Q',\ \text{getM} := \text{getM}\ \text{state} @ [(\text{literal, decision})] \} \text{getWatchList}\ \text{state}[\text{getM} := \text{getM}\ \text{state} @ [(\text{literal, decision})]])\ (\text{opposite}\ \text{literal})\n\text{state}[\text{getQ} := Q'',\ \text{getM} := \text{getM}\ \text{state} @ [(\text{literal, decision})] \}
opposite literal \[
\]
by (simp add: Let-def)

lemma assertedLiteralIsNotUnit:
assumes

InvariantConsistent \(((\text{get}M \text{ state}) \odot [(\text{literal}, \text{decision})])\)
InvariantWatchListsContainOnlyClausesFromF \((\text{getWatchList state})\) and
InvariantWatchListsUniq \((\text{getWatchList state})\) and
InvariantWatchListsCharacterization \((\text{getWatchList state})\) \((\text{getWatch1 state})\) \((\text{getWatch2 state})\)
InvariantWatchesEl \((\text{getF state})\) \((\text{getWatch1 state})\) \((\text{getWatch2 state})\)
and
InvariantWatchesDiffer \((\text{getF state})\) \((\text{getWatch1 state})\) \((\text{getWatch2 state})\)
and
InvariantWatchCharacterization \((\text{getF state})\) \((\text{getWatch1 state})\) \((\text{getWatch2 state})\) \((\text{getM state})\)
shows

let state’ = assertLiteral literal decision state in
\(- \text{literal} \in (\text{set (getQ state’)} - \text{set(getQ state)})\)

proof-

\{ 
let \(?\text{state} = \text{state}(\text{getM := getM state @ [(\text{literal}, \text{decision})]})\)  
let \(?\text{state’} = \text{assertLiteral literal decision state}\) 
assume \(- \?\text{thesis}\)

have \(*\forall l. l \in \text{set (getQ ?state')} - \text{set (getQ ?state)} \rightarrow (\exists \text{clause. clause el (getF ?state) \& isUnitClause clause l})\) \((\text{elements (getM ?state)})\))
using NotifyWatchesLoopQEffect[of \(?\text{state} \text{getM state opposite literal decision getWatchList ?state (opposite literal) []}])
using \(?\text{assms}\)
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchCharacterization-def
unfolding assertLiteral-def
unfolding notifyWatches-def
by (auto simp add: Let-def)
with \(- \?\text{thesis}\)
obtain clause
where isUnitClause clause literal \((\text{elements (getM ?state)})\))
by (auto simp add: Let-def)
hence False
unfolding isUnitClause-def
by simp
\}
thus \(?\text{thesis}\)
by auto
qed

lemma InvariantQCharacterizationAfterAssertLiteralNotInQ:
assumes
  InvariantConsistent ((getM state) @ [(literal, decision)])
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
and
  InvariantWatchListsUniq (getWatchList state) and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
  InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
¬ literal el (getQ state)
shows
let state' = (assertLiteral literal decision state) in
  InvariantQCharacterization (getConflictFlag ?state') (getQ state') (getF state') (getM state')
proof –
  let ?state' = assertLiteral literal decision state
  have InvariantQCharacterization (getConflictFlag ?state') (removeAll literal (getQ ?state')) (getF ?state') (getM ?state')
    using assms
    using InvariantQCharacterizationAfterAssertLiteral
    by (simp add: Let-def)
    moreover
    have ¬ literal el (getQ ?state')
      using assms
      using assertedLiteralIsNotUnit[of state literal decision]
      by (simp add: Let-def)
    hence removeAll literal (getQ ?state') = getQ ?state'
      using removeAll-id[of literal getQ ?state']
      by simp
    ultimately
    show ?thesis
      by (simp add: Let-def)
qed

lemma InvariantUniqQAfterAssertLiteral:
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)

513
\begin{itemize}
  \item \textbf{(getF state) and} \textbf{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}
  \item \textbf{InvariantUniqQ (getQ state)}
\end{itemize}

\textbf{shows}
\begin{itemize}
  \item let \texttt{state′} = assertLiteral literal decision state in
    \textbf{InvariantUniqQ (getQ state')}
\end{itemize}

\textbf{using} \texttt{assms}
\textbf{using} \texttt{InvariantUniqQAfterNotifyWatchesLoop[of state|getM := getM state @ [(literal, decision)]]}
\texttt{getWatchList (state|getM := getM state @ [(literal, decision)])} (opposite literal)
\textbf{unfolding} \texttt{assertLiteral-def}
\textbf{unfolding} \texttt{notifyWatches-def}
\textbf{unfolding} \texttt{InvariantWatchListsContainOnlyClausesFromF-def}
\textbf{by} (auto simp add: \texttt{Let-def})

\textbf{lemma} \texttt{InvariantsNoDecisionsWhenConflictNorUnitAfterAssertLiteral:}
\textbf{assumes}
\begin{itemize}
  \item \texttt{InvariantWatchListsContainOnlyClausesFromF (getWatchList state)} (getF state) \textbf{and}
  \item \texttt{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}
\end{itemize}
\textbf{and}
\begin{itemize}
  \item \texttt{InvariantConflictFlagCharacterization (getConflictFlag state)} (getF state) (getM state)
  \item \texttt{InvariantQCharacterization (getConflictFlag state)} (getQ state) (getF state) (getM state)
  \item \texttt{InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))}
  \item \texttt{InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))}
\end{itemize}
\textbf{shows}
\begin{itemize}
  \item let \texttt{state′} = assertLiteral literal decision state in
    \texttt{InvariantNoDecisionsWhenConflict (getF state') (getM state')} (currentLevel (getM state')) \wedge
    \texttt{InvariantNoDecisionsWhenUnit (getF state') (getM state')} (currentLevel (getM state'))
\end{itemize}
\textbf{proof}−
\begin{itemize}
  \item \texttt{let ?state' = assertLiteral literal decision state}
  \item \texttt{fix level}
  \item \texttt{assume level < currentLevel (getM ?state')}
  \item \texttt{have \neg \texttt{formulaFalse (getF ?state')} (elements (prefixToLevel level (getM ?state')))} \wedge
    \begin{itemize}
      \item \texttt{\neg (\exists \texttt{clause literal. clause el (getF ?state')} \wedge}
      \texttt{\neg \texttt{isUnitClause clause literal (elements (prefixToLevel level (getM ?state'))))}}
    \end{itemize}
\end{itemize}
proof (cases level < currentLevel (getM state))
  case True
    hence prefixToLevel level (getM ?state′) = prefixToLevel level (getM state)
      using assms
      using assertLiteralEffect[of state literal decision]
      by (auto simp add: prefixToLevelAppend)
  moreover
    have ¬ formulaFalse (getF state) (elements (prefixToLevel level (getM state)))
      using (InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state)))
      using (level < currentLevel (getM state))
      unfolding InvariantNoDecisionsWhenConflict-def
      by simp
  moreover
    have ¬ (∃ clause literal. clause el (getF state) ∧ isUnitClause clause literal (elements (prefixToLevel level (getM state))))
      using (InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state)))
      using (level < currentLevel (getM state))
      unfolding InvariantNoDecisionsWhenUnit-def
      by simp
  ultimately
  show ?thesis
    using assms
    using assertLiteralEffect[of state literal decision]
    by auto
next
  case False
  thus ?thesis
proof (cases decision)
  case False
  hence currentLevel (getM ?state′) = currentLevel (getM state)
    using assms
    using assertLiteralEffect[of state literal decision]
    unfolding currentLevel-def
    by (auto simp add: markedElementsAppend)
  thus ?thesis
    using (¬ (level < currentLevel (getM state)))
    using (level < currentLevel (getM ?state′))
    by simp
next
  case True
  hence currentLevel (getM ?state′) = currentLevel (getM state)
    + 1
    using assms
    using assertLiteralEffect[of state literal decision]

515
unfolding currentLevel-def
by (auto simp add: markedElementsAppend)

hence level = currentLevel (getM state)
  using (~ (level < currentLevel (getM state)))
  using (level < currentLevel (getM ?state'))
  by simp

hence prefixToLevel level (getM ?state') = (getM state)
  using ⟨decision⟩
  using assms
  using assertLiteralEffect[of state literal decision]
  using prefixToLevelAppend[of currentLevel (getM state) getM state [(literal, True)]]
  by auto
  thus ?thesis
  using ⟨decision⟩
  using ⟨decision ⟧ → ¬ (getConflictFlag state) ∧ (getQ state)

= []
  using (InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state))
  using (InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state))
  unfolding InvariantConflictFlagCharacterization-def
  unfolding InvariantQCharacterization-def
  using assms
  using assertLiteralEffect[of state literal decision]
  by simp
  qed

qed

thus ?thesis
unfolding InvariantNoDecisionsWhenConflict-def
unfolding InvariantNoDecisionsWhenUnit-def
  by auto

qed


lemma InvariantVarsQAfterAssertLiteral:
assumes
  InvariantConsistent ((getM state) @ [(literal, decision)])
  InvariantUniq ((getM state) @ [(literal, decision)])
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  InvariantWatchListsUniq (getWatchList state)
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state)
shown

let state' = assertLiteral literal decision state in
InvariantVarsQ (getQ state') F0 Vbl

proof
let \( \forall \exists \) uc uc el (getF state) \land
(getM state) \oplus \{ \text{literal} \}
let ?state' = assertLiteral literal decision state
have vars ?Q' \subseteq \text{vars} (getF state)
proof
fix vbl::Variable
assume vbl \in \text{vars} ?Q'
then obtain ul::Literal
  where ul \in ?Q' \land ul = vbl
  by auto
then obtain uc::Clause
  where uc el (getF state) isUnitClause uc ul (elements (getM state) \oplus \{ \text{literal} \})
by auto
hence vars uc \subseteq \text{vars} (getF state) var ul = vbl
using formulaContainsItsClausesVariables[of getF state]
using clauseContainsItsLiteralsVariable[of ul uc]
unfolding isUnitClause-def
by auto
thus vbl \in \text{vars} (getF state)
using (var ul = vbl)
by auto
qed
thus ?thesis
using assms
using assertLiteralQEEffect[of state literal decision]
using varsClauseVarsSet[of getQ ?state']
using varsClauseVarsSet[of getQ state]
unfolding InvariantVarsQ-def
unfolding InvariantVarsF-def
by (auto simp add: Let-def)
qed

end
theory UnitPropagate
imports AssertLiteral
begin
lemma applyUnitPropagateEffect:
  assumes
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  and
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
  ¬ (getConflictFlag state)
  getQ state ≠ []
  shows
  let uLiteral = hd (getQ state) in
  let state' = applyUnitPropagate state in
  ∃ uClause. formulaEntailsClause (getF state) uClause ∧
  isUnitClause uClause uLiteral (elements (getM state)) ∧
  (getM state') = (getM state) @ [(uLiteral, False)]
proof−
  let ?uLiteral = hd (getQ state)
  obtain uClause
    where uClause el (getF state) isUnitClause uClause ?uLiteral (elements (getM state))
    using assms
    unfolding InvariantQCharacterization-def
    by force
  thus ?thesis
    using assms
    using assertLiteralEffect[of state ?uLiteral False]
    unfolding applyUnitPropagate-def
    using formulaEntailsItsClauses[of uClause getF state]
    by (auto simp add: Let-def )
qed

lemma InvariantConsistentAfterApplyUnitPropagate:
  assumes
  InvariantConsistent (getM state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  and
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
  getQ state ≠ []
  ¬ (getConflictFlag state)
  shows
  let state' = applyUnitPropagate state in
  InvariantConsistent (getM state')
proof−
let ?uLiteral = hd (getQ state)
let ?state' = applyUnitPropagate state
obtain uClause
  where isUnitClause uClause ?uLiteral (elements (getM state)) and
  (getM ?state') = (getM state) @ [?(uLiteral, False)]
  using assms
  using applyUnitPropagateEffect[of state]
  by (auto simp add: Let-def)
thus ?thesis
  using assms
  using InvariantConsistentAfterUnitPropagate[of getM state uClause ?uLiteral getM ?state']
  by (auto simp add: Let-def)
qed

lemma InvariantUniqAfterApplyUnitPropagate:
assumes
  InvariantUniq (getM state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
  getQ state ≠ []
  ¬ (getConflictFlag state)
shows
  let state' = applyUnitPropagate state in
  InvariantUniq (getM state')
proof –
  let ?uLiteral = hd (getQ state)
  let ?state' = applyUnitPropagate state
  obtain uClause
    where isUnitClause uClause ?uLiteral (elements (getM state)) and
    (getM ?state') = (getM state) @ [?(uLiteral, False)]
    using assms
    using applyUnitPropagateEffect[of state]
    by (auto simp add: Let-def)
  thus ?thesis
    using assms
    using InvariantUniqAfterUnitPropagate[of getM state uClause ?uLiteral getM ?state']
    by (auto simp add: Let-def)
qed

lemma InvariantWatchCharacterizationAfterApplyUnitPropagate:
assumes
  InvariantConsistent (getM state)
InvariantUniq \((\text{getM} \text{ state})\)
InvariantsWatchListsContainOnlyClausesFromF \((\text{getWatchList} \text{ state})\)

\((\text{getF} \text{ state})\) and
InvariantsWatchListsUniq \((\text{getWatchList} \text{ state})\) and
InvariantsWatchListsCharacterization \((\text{getWatchList} \text{ state})\) \((\text{getWatch1} \text{ state})\) \((\text{getWatch2} \text{ state})\)

and
InvariantsWatchesDiffer \((\text{getF} \text{ state})\) \((\text{getWatch1} \text{ state})\) \((\text{getWatch2} \text{ state})\)
InvariantsWatchesCharacterization \((\text{getF} \text{ state})\) \((\text{getWatch1} \text{ state})\) \((\text{getWatch2} \text{ state})\)

\((\text{getF} \text{ state})\) \((\text{getWatch1} \text{ state})\) \((\text{getWatch2} \text{ state})\)

\(\text{InvariantQCharacterization}\)

\((\text{getConflictFlag} \text{ state})\) \((\text{getQ} \text{ state})\) \((\text{getF} \text{ state})\) \((\text{getM} \text{ state})\)

\((\text{getQ} \text{ state})\) \(\neq []\)

\(\neg (\text{getConflictFlag} \text{ state})\)

shows
let state' = applyUnitPropagate state in

\(\text{InvariantWatchCharacterization}\) \((\text{getF} \text{ state}')\) \((\text{getWatch1} \text{ state}')\)
\((\text{getWatch2} \text{ state}')\) \((\text{getM} \text{ state}')\)

proof –

let \(?uLiteral = \text{hd} (\text{getQ} \text{ state})\)

let state' = assertLiteral ?uLiteral False state
let state'' = applyUnitPropagate state

have\(\text{InvariantConsistent}\) \((\text{getM} \text{ state}')\)
using\ assms

using\ \text{InvariantConsistentAfterApplyUnitPropagate}[\text{of state}]

unfolding\ applyUnitPropagate-def

by\ (auto simp add: \text{Let-def})

moreover

have\ \text{InvariantUniq} \((\text{getM} \text{ state}')\)
using\ assms

using\ \text{InvariantUniqAfterApplyUnitPropagate}[\text{of state}]

unfolding\ applyUnitPropagate-def

by\ (auto simp add: \text{Let-def})

ultimately

show\ \text{thesis}
using\ assms

using\ \text{InvariantWatchCharacterizationAfterAssertLiteral}[\text{of state}

\(\text{?uLiteral} \text{ False})\)

using\ \text{assertLiteralEffect}

unfolding\ applyUnitPropagate-def

by\ (simp add: \text{Let-def})

qed

lemma\ \text{InvariantConflictFlagCharacterizationAfterApplyUnitPropagate}:}

assumes

\(\text{InvariantConsistent} \ (\text{getM} \text{ state})\)

\(\text{InvariantUniq} \ (\text{getM} \text{ state})\)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
¬ getConflictFlag state
getQ state ≠ []
shows
let state’ = (applyUnitPropagate state) in
InvariantConflictFlagCharacterization (getConflictFlag state’) (getF state’) (getM state’)
proof –
let ?uLiteral = hd (getQ state)
let ?state’ = assertLiteral ?uLiteral False state
let ?state’’ = applyUnitPropagate state
have InvariantConsistent (getM ?state’)
using assms
using InvariantConsistentAfterApplyUnitPropagate[of state]
unfolding applyUnitPropagate-def
by (auto simp add: Let-def)
moreover
have InvariantUniq (getM ?state’)
using assms
using InvariantUniqAfterApplyUnitPropagate[of state]
unfolding applyUnitPropagate-def
by (auto simp add: Let-def)
ultimately
show ?thesis
using assms
using InvariantConflictFlagCharacterizationAfterAssertLiteral[of state ?uLiteral False]
using assertLiteralEffect
unfolding applyUnitPropagate-def
by (simp add: Let-def)
qed

lemma InvariantConflictClauseCharacterizationAfterApplyUnitPropagate:
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
InvariantWatchListsUniq (getWatchList state)
¬ getConflictFlag state
shows
let state' = applyUnitPropagate state in
InvariantConflictClauseCharacterization (getConflictFlag state') (getConflictClause state') (getF state') (getM state')
using assms
using InvariantConflictClauseCharacterizationAfterAssertLiteral[of state hd (getQ state) False]
unfolding applyUnitPropagate-def
unfolding InvariantWatchesEl-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchListsUniq-def
unfolding InvariantConflictClauseCharacterization-def
by (simp add: Let-def)

lemma InvariantQCharacterizationAfterApplyUnitPropagate:
assumes
InvariantConsistent (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantUniqQ (getQ state)
(getQ state) \neq []
¬ (getConflictFlag state)
shows
let state'' = applyUnitPropagate state in
InvariantQCharacterization (getConflictFlag state'') (getQ state'') (getF state'') (getM state'')
proof

let ?uLiteral = hd (getQ state)
let ?state' = assertLiteral ?uLiteral False state
let ?state'' = applyUnitPropagate state
have InvariantConsistent (getM ?state')
  using assms
  using InvariantConsistentAfterApplyUnitPropagate[of state]
  unfolding applyUnitPropagate-def
  by (auto simp add: Let-def)

hence InvariantQCharacterization (getConflictFlag ?state') (removeAll ?uLiteral (getQ ?state')) (getF ?state') (getM ?state')
  using assms
  using InvariantQCharacterizationAfterAssertLiteral[of state ?uLiteral False]
  unfolding assertLiteralEffect[of state ?uLiteral False]
  by (simp add: Let-def)

moreover
have InvariantUniqQ (getQ ?state')
  using assms
  using InvariantUniqQAfterAssertLiteral[of state ?uLiteral False]
  by (simp add: Let-def)

have ?uLiteral = (hd (getQ ?state'))
proof -
  obtain s
    where (getQ state) @ s = getQ ?state'
    using assms
    using assertLiteralEffect[of state ?uLiteral False]
    unfolding isPrefix-def
    by auto
  hence getQ ?state' = (getQ state) @ s
    by (rule sym)
  thus ?thesis
    using (getQ state) ≠ []
    using hd-append[of getQ state s]
    by auto
qed

hence set (getQ ?state'') = set (removeAll ?uLiteral (getQ ?state'))
  using assms
  using InvariantUniqQ (getQ ?state')
  unfolding InvariantUniqQ-def
  using uniqHeadTailSet[of getQ ?state']
  unfolding applyUnitPropagate-def
  by (simp add: Let-def)

ultimately
show ?thesis
  unfolding InvariantQCharacterization-def
  unfolding applyUnitPropagate-def

523
lemma InvariantUniqQAfterApplyUnitPropagate:
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
InvariantUniqQ (getQ state)
getQ state ≠ []
shows
let state'' = applyUnitPropagate state in
InvariantUniqQ (getQ state'')
proof−
let ?uLiteral = hd (getQ state)
let ?state' = assertLiteral ?uLiteral False state
let ?state'' = applyUnitPropagate state
have InvariantUniqQ (getQ ?state')
using assms
using InvariantUniqQAfterAssertLiteral[of state ?uLiteral False]
by (simp add: Let-def)
moreover
obtain s
where getQ state @ s = getQ ?state'
using assms
using assertLiteralEffect[of state ?uLiteral False]
unfolding isPrefix-def
by auto
hence getQ ?state' = getQ state @ s
by (rule sgm)
with 'getQ state ≠ []'
have getQ ?state' ≠ []
by simp
ultimately
show ?thesis
using (getQ state ≠ [])
unfolding InvariantUniqQ-def
unfolding applyUnitPropagate-def
using hd-Cons-tl[of getQ ?state']
using uniqAppendIff[of [hd (getQ ?state')] tl (getQ ?state')]
by (simp add: Let-def)
qed

lemma InvariantNoDecisionsWhenConflictNorUnitAfterUnitPropagate:
assumes
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
(getF state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))
InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))

shows

let state' = applyUnitPropagate state in
InvariantNoDecisionsWhenConflict (getF state') (getM state') (currentLevel (getM state')) ∧
InvariantNoDecisionsWhenUnit (getF state') (getM state') (currentLevel (getM state'))

using assms

unfolding applyUnitPropagate-def
using InvariantsNoDecisionsWhenConflictNorUnitAfterAssertLiteral[of state False hd (getQ state)]

unfolding InvariantNoDecisionsWhenConflict-def
by (simp add: Let-def)

lemma InvariantGetReasonIsReasonAfterApplyUnitPropagate:
assumes
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state) and
InvariantUniqQ (getQ state) and
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state)) and
getQ state ≠ [] and
¬ getConflictFlag state

shows

let state' = applyUnitPropagate state in
InvariantGetReasonIsReason (getReason state') (getF state') (getM state') (set (getQ state'))

proof

let ?state0 = state (getM := getM state @ [(hd (getQ state), False)])
let ?state' = assertLiteral (hd (getQ state)) False state
let ?state'' = applyUnitPropagate state

have InvariantGetReasonIsReason (getReason ?state0) (getF ?state0)
(getM ?state0) (set (removeAll (hd (getQ ?state0)) (getQ ?state0)))
proof
{
  fix l :: Literal
  assume *: l el (elements (getM ?state0)) ∧ ¬ l el (decisions (getM ?state0)) ∧ elementLevel l (getM ?state0) > 0
  hence ∃ reason. getReason ?state0 l = Some reason ∧ 0 ≤ reason ∧ reason < length (getF ?state0) ∧
    isReason (nth (getF ?state0) reason) l (elements (getM ?state0))
  proof (cases l el (elements (getM state)))
  case True
    from * have ¬ l el (decisions (getM state))
      by (auto simp add: markedElementsAppend)
    from * have elementLevel l (getM state) > 0
      using elementLevelAppend[of l getM state [(hd (getQ state), False)]
      using ⟨l el (elements (getM state))⟩ by simp
    show ?thesis
      using ⟨InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))⟩
      using ⟨l el (elements (getM state))⟩
      using ⟨¬ l el (decisions (getM state))⟩
      using ⟨elementLevel l (getM state) > 0⟩
      unfolding InvariantGetReasonIsReason-def
      by (auto simp add: isReasonAppend)
  next
  case False
    with * have l = hd (getQ state)
      by simp
    have currentLevel (getM ?state0) > 0
      using *
      using elementLevelLeqCurrentLevel[of l getM ?state0]
      by auto
    hence currentLevel (getM state) > 0
      unfolding currentLevel-def
      by (simp add: markedElementsAppend)
    moreover
    have hd (getQ ?state0) el (getQ state)
      using (getQ state ≠ [])
      by simp
    ultimately
    obtain reason
where getReason state (hd (getQ state)) = Some reason 0 ≤ reason ∧ reason < length (getF state)
  isUnitClause (nth (getF state) reason) (hd (getQ state))
  (elements (getM state)) ∨ clauseFalse (nth (getF state) reason) (elements (getM state))

using ⟨InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))⟩
unfolding InvariantGetReasonIsReason-def
by auto
hence isUnitClause (nth (getF state) reason) (hd (getQ state))
  (elements (getM state))
using (~ getConflictFlag state)
using ⟨InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)⟩
unfolding InvariantConflictFlagCharacterization-def
using nth-mem[of reason getF state]
using ⟨formulaFalseIfContainsFalseClause[of getF state elements (getM state)]⟩
by simp
thus ?thesis
using (getReason state (hd (getQ state))) = Some reason: 0 ≤ reason ∧ reason < length (getF state):
  using isUnitClauseIsReason[of reason getF state]
  unfolding currentLevel (getM state)[hd (getQ state)]
  unfolding currentLevel-def
  by (simp add: markedElementsAppend)

assume literal el removeAll (hd (getQ ?state0)) (getQ ?state0)
hence literal ≠ hd (getQ state) literal el getQ state
by auto

then obtain reason
where getReason state literal = Some reason 0 ≤ reason ∧ reason < length (getF state)
and
  isUnitClause (nth (getF state) reason) literal (elements (getM state)) ∨
  clauseFalse (nth (getF state) reason) (elements (getM state))
using ⟨currentLevel (getM state) > 0, getF state (currentLevel (getM state))⟩
using ⟨InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))⟩
unfolding InvariantGetReasonIsReason-def
by auto
qed
state) (getM state) (set (getQ state))

unfolding InvariantGetReasonIsReason-def
by auto

hence ∃ reason. getReason ?state0 literal = Some reason ∧ 0 ≤ reason ∧ reason < length (getF ?state0) ∧
(isUnitClause (nth (getF ?state0) reason) literal (elements (getM ?state0))) ∨
clauseFalse (nth (getF ?state0) reason) (elements (getM ?state0)))

proof (cases isUnitClause (nth (getF state) reason) literal (elements (getM state)))
case True
  show ?thesis
  proof (cases opposite literal = hd (getQ state))
    case True
    thus ?thesis
    using (isUnitClause (nth (getF state) reason) literal (elements (getM state))):
      using (getReason state literal = Some reason):
      using (literal ≠ hd (getQ state)):
      using (0 ≤ reason ∧ reason < length (getF state)):
      unfolding isUnitClause-def
      by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
    next
    case False
    thus ?thesis
    using (isUnitClause (nth (getF state) reason) literal (elements (getM state))):
      using (getReason state literal = Some reason):
      using (literal ≠ hd (getQ state)):
      using (0 ≤ reason ∧ reason < length (getF state)):
      unfolding isUnitClause-def
      by auto
  qed
next
case False
with *
  have clauseFalse (nth (getF state) reason) (elements (getM state))
    by simp
  thus ?thesis
  using (getReason state literal = Some reason):
  using (0 ≤ reason ∧ reason < length (getF state)):
  using clauseFalseAppendValuation[of nth (getF state) reason
    elements (getM state) [hd (getQ state)]]
    by auto
  qed
}
ultimately

528
show ?thesis
  unfolding InvariantGetReasonIsReason-def
  by auto
qed

hence InvariantGetReasonIsReason (getReason ?state′) (getF ?state′) (getM ?state′) (set (removeAll (hd (getQ state)) (getQ state)) ∪ (set (getQ ?state′) – set (getQ state)))
  using assms
  unfolding assertLiteral-def
  unfolding notifyWatches-def
  using InvariantGetReasonIsReasonAfterNotifyWatches[of ?state0 getWatchList ?state0 (opposite (hd (getQ state))) opposite (hd (getQ state)) getM state False set (removeAll (hd (getQ ?state0)) (getQ ?state0)) []]
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  unfolding InvariantWatchListsCharacterization-def
  unfolding InvariantWatchListsUniq-def
  by (auto simp add: Let-def)

obtain s
  where getQ state @ s = getQ ?state'
  using assms
  unfolding isPrefix-def
  by auto
hence getQ ?state' = getQ state @ s
  by simp
hence hd (getQ ?state') = hd (getQ state)
  using hd-append2[of getQ state s]
  using (getQ state ≠ [])
  by simp

have set (removeAll (hd (getQ state)) (getQ state)) ∪ (set (getQ ?state′) – set (getQ state)) =
  set (removeAll (hd (getQ state)) (getQ ?state′))
  using (getQ ?state' = getQ state @ s)
  using (getQ state ≠ []]
  by auto

have uniq (getQ ?state')
  using assms
  using InvariantUniqQAfterAssertLiteral[of state hd (getQ state) False]
  unfolding InvariantUniqQ-def
  by (simp add: Let-def)

have set (getQ ?state'') = set (removeAll (hd (getQ state)) (getQ ?state′))
using `uniq (getQ ?state')`
using `hd (getQ ?state') = hd (getQ state)`
using `uniqHeadTailSet[of getQ ?state']`
unfolding `applyUnitPropagate-def`
by (simp add: Let-def)

thus ?thesis
using `InvariantGetReasonIsReason (getReason ?state') (getF ?state') (getM ?state') (set (removeAll (hd (getQ state)) (getQ state)) ∪ (set (getQ ?state') − set (getQ state)))`
using `set (getQ ?state') = set (removeAll (hd (getQ state)) (getQ ?state'))`
using `set (removeAll (hd (getQ state)) (getQ state)) ∪ (set (getQ ?state') − set (getQ state)) = set (removeAll (hd (getQ state)) (getQ ?state'))`
unfolding `applyUnitPropagate-def`
by (simp add: Let-def)

qed

lemma `Invariant Equivalent ZL After Apply Unit Propagate`:
assumes `Invariant Equivalent ZL (getF state) (getM state) Phi`
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) (getM state)
and
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)

¬ (getConflictFlag state)
getQ state ≠ []

shows
let state' = applyUnitPropagate state in
Invariant Equivalent ZL (getF state') (getM state') Phi

proof–
let ?uLiteral = hd (getQ state)
let ?state' = applyUnitPropagate state
let ?FM = getF state @ val2form (elements (prefixToLevel 0 (getM state)))
let ?FM' = getF ?state' @ val2form (elements (prefixToLevel 0 (getM ?state')))

obtain uClause
  where formulaEntailsClause (getF state) uClause and
  isUnitClause uClause ?uLiteral (elements (getM state)) and
  (getM ?state') = (getM state) @ [('uLiteral, False)]
  (getF ?state') = (getF state)
using assms
using applyUnitPropagateEffect[of state]
unfolding applyUnitPropagate-def
using assertLiteralEffect
by (auto simp add: Let-def)

note * = this

show ?thesis
proof (cases currentLevel (getM state) = 0)
  case True
  hence getM state = prefixToLevel 0 (getM state)
    by (rule currentLevelZeroTrailEqualsItsPrefixToLevelZero)

  have ?FM' = ?FM @ [[?uLiteral]]
    using *
    using (getM state') = (getM state) @ [[?uLiteral, False]];
    using prefixToLevelAppend[of 0 getM state [[?uLiteral, False]]]
    using (currentLevel (getM state) = 0);
    using (getM state = prefixToLevel 0 (getM state));
    by (auto simp add: val2formAppend)

  have formulaEntailsLiteral ?FM ?uLiteral
    using *
    using unitLiteralIsEntailed[of uClause ?uLiteral elements (getM state) (getF state)]
    using (InvariantEquivalentZL (getF state) (getM state) Phi);
    unfolding InvariantEquivalentZL-def
    by simp

  hence formulaEntailsClause ?FM [?uLiteral]
    unfolding formulaEntailsLiteral-def
    unfolding formulaEntailsClause-def
    by (auto simp add: clauseTrueIffContainsTrueLiteral)

  show ?thesis
    using (InvariantEquivalentZL (getF state) (getM state) Phi)
    using (?FM' = ?FM @ [[?uLiteral]]);
    using (formulaEntailsClause ?FM [?uLiteral])
    unfolding InvariantEquivalentZL-def
    using extendEquivalentFormulaWithEntailedClause[of Phi ?FM [?uLiteral]]
    by (simp add: equivalentFormulaeSymmetry)

  next
  case False
  hence !FM = ?FM'
    using *
    using prefixToLevelAppend[of 0 getM state [[?uLiteral, False]]]
    by (simp add: Let-def)

  show ?thesis
    using (InvariantEquivalentZL (getF state) (getM state) Phi)
    using (?FM' = ?FM @ [[?uLiteral]]);
    using (formulaEntailsClause ?FM [?uLiteral])
    unfolding InvariantEquivalentZL-def
    using extendEquivalentFormulaWithEntailedClause[of Phi ?FM [?uLiteral]]
    by (simp add: equivalentFormulaeSymmetry)

next
  case False
  hence ?thesis
thus \( ?\)thesis
using \((\text{Invariant}\text{EquivalentZL}\ (\text{getF}\ state)\ (\text{getM}\ state)\ \Phi)\)
unfolding \(\text{Invariant}\text{EquivalentZL}\text{-def}\)
by \((\text{simp add: Let-def})\)
qed

qed

lemma \text{InvariantVarsQTl}:
assumes
\(\text{InvariantVarsQ}\ Q\ F0\ Vbl\)
\(Q \neq []\)
shows
\(\text{InvariantVarsQ}\ (\text{tl}\ Q)\ F0\ Vbl\)
proof—
  have \(\text{InvariantVarsQ}\ ((\text{hd}\ Q) \# (\text{tl}\ Q))\ F0\ Vbl\)
    using \(\text{assms}\)
    by \(\text{simp}\)
  hence \(\{\text{var}\ (\text{hd}\ Q)\} \cup \text{vars} (\text{tl}\ Q) \subseteq \text{vars} F0 \cup Vbl\)
    unfolding \(\text{InvariantVarsQ-def}\)
    by \(\text{simp}\)
  thus \(\ ?\)thesis
    unfolding \(\text{InvariantVarsQ-def}\)
    by \(\text{simp}\)
qed

lemma \text{InvariantsVarsAfterApplyUnitPropagate}:
assumes
\(\text{InvariantConsistent}\ (\text{getM}\ state)\)
\(\text{InvariantUniq}\ (\text{getM}\ state)\)
\(\text{InvariantWatchesEl}\ (\text{getF}\ state)\ (\text{getWatch1}\ state)\ (\text{getWatch2}\ state)\)
and
\(\text{InvariantWatchListsContainOnlyClausesFromF}\ (\text{getWatchList}\ state)\ (\text{getF}\ state)\) and
\(\text{InvariantWatchListsCharacterization}\ (\text{getWatchList}\ state)\ (\text{getWatch1}\ state)\ (\text{getWatch2}\ state)\) and
\(\text{InvariantWatchListsUniq}\ (\text{getWatchList}\ state)\) and
\(\text{InvariantWatchesDiffer}\ (\text{getF}\ state)\ (\text{getWatch1}\ state)\ (\text{getWatch2}\ state)\) and
\(\text{InvariantWatchCharacterization}\ (\text{getF}\ state)\ (\text{getWatch1}\ state)\ (\text{getWatch2}\ state)\ (\text{getM}\ state)\) and
\(\text{InvariantQCharacterization}\ False\ (\text{getQ}\ state)\ (\text{getF}\ state)\ (\text{getM}\ state)\) and
\(\text{getQ}\ state \neq []\)
\(\neg \text{getConflictFlag}\ state\)
\(\text{InvariantVarsM}\ (\text{getM}\ state)\ F0\ Vbl\) and
\(\text{InvariantVarsQ}\ (\text{getQ}\ state)\ F0\ Vbl\) and
\(\text{InvariantVarsF}\ (\text{getF}\ state)\ F0\ Vbl\)
shows

532
let state' = applyUnitPropagate state in
InvariantVarsM (getM state') F0 Vbl ∧
InvariantVarsQ (getQ state') F0 Vbl

proof
let ?state' = assertLiteral (hd (getQ state)) False state
let ?state'' = applyUnitPropagate state
have InvariantVarsQ (getQ ?state') F0 Vbl
  using assms
  using InvariantConsistentAfterApplyUnitPropagate[of state]
  using InvariantUniqAfterApplyUnitPropagate[of state]
  using InvariantVarsQAfterAssertLiteral[of state hd (getQ state) False F0 Vbl]
  unfolding applyUnitPropagate-def
  by (simp add: Let-def)
moreover
have (getQ ?state') ≠ []
  using assms
  using assertLiteralEffect[of state hd (getQ state) False]
  unfolding isPrefix-def
  by auto
ultimately
have InvariantVarsQ (getQ ?state'') F0 Vbl
  unfolding applyUnitPropagate-def
  using InvariantVarsQTl[of getQ ?state' F0 Vbl]
  by (simp add: Let-def)
moreover
have var (hd (getQ state)) ∈ vars F0 ∪ Vbl
  using (getQ state) ≠ []
  using (InvariantVarsQ (getQ state) F0 Vbl)
  using hd-in-set[of getQ state]
  using clauseContainsItsLiteralsVariable[of hd (getQ state) getQ state]
  unfolding InvariantVarsQ-def
  by auto
hence InvariantVarsM (getM ?state'') F0 Vbl
  using assms
  using assertLiteralEffect[of state hd (getQ state) False]
  using varsAppendValuation[of elements (getM state) [hd (getQ state)]]
  unfolding applyUnitPropagate-def
  unfolding InvariantVarsM-def
  by (simp add: Let-def)
ultimately
show ?thesis
  by (simp add: Let-def)
qed
**definition** \( \text{lexLessState} \ (\text{Vbl}::\text{Variable set}) \equiv \{(\text{state1}, \text{state2}). (\text{getM state1}, \text{getM state2}) \in \text{lexLessRestricted Vbl}\} \)

**lemma** \( \text{exhaustiveUnitPropagateTermination} \):

**fixes**

\( \text{state}::\text{State} \text{ and } \text{Vbl}::\text{Variable set} \)

**assumes**

- \( \text{InvariantUniq} \ (\text{getM state}) \)
- \( \text{InvariantConsistent} \ (\text{getM state}) \)
- \( \text{InvariantWatchListsContainOnlyClausesFromF} \ (\text{getWatchList state}) \)
  \( \text{and} \)
- \( \text{InvariantWatchListsUniq} \ (\text{getWatchList state}) \)
- \( \text{InvariantWatchListsCharacterization} \ (\text{getWatchList state}) \)
- \( \text{InvariantWatchCharacterization} \ (\text{getF state}) \)
- \( \text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag state}) \)
- \( \text{InvariantQCharacterization} \ (\text{getConflictFlag state}) \)
- \( \text{InvariantUniqQ} \ (\text{getQ state}) \)
- \( \text{InvariantVarsM} \ (\text{getM state}) \)
- \( \text{InvariantVarsQ} \ (\text{getQ state}) \)
- \( \text{InvariantVarsF} \ (\text{getF state}) \)
- finite \( \text{Vbl} \)

**shows**

\( \text{exhaustiveUnitPropagate-dom state} \)

**using** \( \text{assms} \)

**proof** (induct rule: \( \text{wf-induct[of lexLessState (vars F0 \cup Vbl)]} \))

**case 1**

**show** ?case

- unfolding \( \text{wf-eq-minimal} \)
  **proof**

  - show \( \forall Q \ (\text{state}::\text{State}). \text{state} \in Q \rightarrow (\exists \text{stateMin} \in Q. \forall \text{state}'. \text{state}', \text{stateMin}) \in \text{lexLessState (vars F0 \cup Vbl)} \rightarrow \text{state}' \notin Q) \)

  **proof**

  - { fix \( Q :: \text{State set} \text{ and } \text{state} :: \text{State} \)
    assume \( \text{state} \in Q \)
    let \( ?Q1 = \{\text{M::LiteralTrail}. \exists \text{state}. \text{state} \in Q \land (\text{getM state}) \land \)
from \( \langle \text{state} \in Q \rangle \)

have \( \text{getM state} \in \exists Q_1 \)

by auto

have \( \text{wf} \ (\text{lexLessRestricted} \ (\text{vars} \ F_0 \cup Vbl)) \)

using \( \langle \text{finite} Vbl \rangle \)

using \( \text{finiteVarsFormula[of} \ F_0 \rangle \)

using \( \text{wfLexLessRestricted[of} \ \text{vars} \ F_0 \cup Vbl \rangle \)

by simp

with \( \langle \text{getM state} \in \exists Q_1 \rangle \)

obtain \( M_{\text{min}} \) where \( M_{\text{min}} \in \exists Q_1 \forall \ M'. (M', M_{\text{min}}) \in \text{lexLessRestricted} \ (\text{vars} \ F_0 \cup Vbl) \rightarrow M' \notin \exists Q_1 \)

unfolding \( \text{wf-eq-minimal} \)

apply \( \langle \text{erule-tac} \ x=\exists Q_1 \ \text{in} \ \text{allE} \rangle \)

apply \( \langle \text{erule-tac} \ x=\text{getM state} \ \text{in} \ \text{allE} \rangle \)

by auto

from \( \langle M_{\text{min}} \in \exists Q_1 \rangle \)

obtain \( \text{stateMin} \)

where \( \text{stateMin} \in Q \) \( \text{(getM stateMin)} = M_{\text{min}} \)

by auto

have \( \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{lexLessState} \ (\text{vars} \ F_0 \cup Vbl) \rightarrow \text{state}' \notin Q \)

proof

fix \( \text{state}' \)

show \( (\text{state}', \text{stateMin}) \in \text{lexLessState} \ (\text{vars} \ F_0 \cup Vbl) \rightarrow \text{state}' \notin Q \)

proof

assume \( (\text{state}', \text{stateMin}) \in \text{lexLessState} \ (\text{vars} \ F_0 \cup Vbl) \)

hence \( (\text{getM state}', \text{getM stateMin}) \in \text{lexLessRestricted} \ (\text{vars} \ F_0 \cup Vbl) \)

unfolding \( \text{lexLessState-def} \)

by auto

from \( \forall M'. (M', M_{\text{min}}) \in \text{lexLessRestricted} \ (\text{vars} \ F_0 \cup Vbl) \rightarrow M' \notin \exists Q_1 \)

\( \langle (\text{getM state}', \text{getM stateMin}) \in \text{lexLessRestricted} \ (\text{vars} \ F_0 \\cup Vbl) \rangle \ (\text{getM stateMin} = M_{\text{min}}) \)

have \( \text{getM state}' \notin \exists Q_1 \)

by simp

with \( \langle \text{getM stateMin} = M_{\text{min}} \rangle \)

show \( \text{state}' \notin Q \)

by auto

qed

qed

with \( \langle \text{stateMin} \in Q \rangle \)

have \( \exists \text{stateMin} \in Q \forall \text{state}'. (\text{state}', \text{stateMin}) \in \text{lexLessState} \ (\text{vars} \ F_0 \cup Vbl) \rightarrow \text{state}' \notin Q \)

by auto

}\)

thus \( \exists \text{thesis} \)

by auto
qed
qed
next
  case (2 state')
  note ih = this
  show ?case
  proof (cases getQ state' = [] ∨ getConflictFlag state')
    case False
    let ?state" = applyUnitPropagate state'

    have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state") (getF ?state") and
      InvariantWatchListsUniq (getWatchList ?state") and
      InvariantWatchListsCharacterization (getWatchList ?state")
      (getWatch1 ?state") (getWatch2 ?state")
      InvariantWatchesEl (getF ?state") (getWatch1 ?state") (getWatch2 ?state")
      and
      InvariantWatchesDiffer (getF ?state") (getWatch1 ?state") (getWatch2 ?state")
      using ih
      using WatchInvariantsAfterAssertLiteral[of state' hd (getQ state') False]
      unfolding applyUnitPropagate-def
      by (auto simp add: Let-def)
    moreover
    have InvariantWatchCharacterization (getF ?state") (getWatch1 ?state") (getWatch2 ?state") (getM ?state")
      using ih
      using InvariantWatchCharacterizationAfterApplyUnitPropagate[of state']
      unfolding InvariantQCharacterization-def
      using False
      by (simp add: Let-def)
    moreover
    have InvariantQCharacterization (getConflictFlag ?state") (getQ ?state") (getF ?state") (getM ?state")
      using ih
      using InvariantQCharacterizationAfterApplyUnitPropagate[of state']
      unfolding InvariantConflictFlagCharacterization-def
      using False
      by (simp add: Let-def)
    moreover
    have InvariantConflictFlagCharacterization (getConflictFlag ?state") (getF ?state") (getM ?state")
      using ih
      using InvariantConflictFlagCharacterizationAfterApplyUnitPropagate[of state']
      unfolding InvariantConflictFlagCharacterization-def
      using False
      by (simp add: Let-def)
moreover
have InvariantUniqQ (getQ ?state"
  using ih
  using InvariantUniqQAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
movere
have InvariantConsistent (getM ?state"
  using ih
  using InvariantConsistentAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
movere
have InvariantUniq (getM ?state"
  using ih
  using InvariantUniqAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
movere
have InvariantVarsM (getM ?state"
  F0 Vbl InvariantVarsQ (getQ
  ?state"
  F0 Vbl
  using ih
  unfolding applyUnitPropagate-def
  using assertLiteralEffect[of state' hd (getQ state') False]
  using ih
  by (simp add: Let-def)
movere
have (?(state", state") \in lexLessState (vars F0 \cup Vbl)
proof
have getM ?state" = getM state' @ [(hd (getQ state'), False)]
  unfolding applyUnitPropagate-def
  using ih
  using assertLiteralEffect[of state' hd (getQ state') False]
  by (simp add: Let-def)
thus ?thesis
unfolding lexLessState-def
unfolding lexLessRestricted-def
using lexLessAppend[of [(hd (getQ state'), False)] getM state']
using (InvariantConsistent (getM ?state")
unfolding InvariantConsistent-def
using (InvariantConsistent (getM state'))
unfolding InvariantConsistent-def
using (InvariantUniq (getM ?state")
unfolding InvariantUniq-def
using \((\text{InvariantUniq } (\text{getM } \text{state}'))\)
unfolding \(\text{InvariantUniq-def}\)
using \((\text{InvariantVarsM } (\text{getM } ?\text{state}''))\) \(F0\) \(Vbl\)
using \((\text{InvariantVarsM } (\text{getM } \text{state}'))\) \(F0\) \(Vbl\)
unfolding \(\text{InvariantVarsM-def}\)
by simp
qed
ultimately
have \(\text{exhaustiveUnitPropagate-dom } ?\text{state}"\)
using \(\text{ih}\)
by auto
thus \(?\text{thesis}\)
using \(\text{exhaustiveUnitPropagate-dom.intros}[\text{of state}']\)
using \(\text{False}\)
by simp
next
case \(\text{True}\)
show \(?\text{thesis}\)
apply \((\text{rule exhaustiveUnitPropagate-dom.intros})\)
using \(\text{True}\)
by simp
qed
qed

lemma \(\text{exhaustiveUnitPropagatePreservedVariables:}\)
assumes
\(\text{exhaustiveUnitPropagate-dom } \text{state}\)
\(\text{InvariantWatchListsContainOnlyClausesFromF } (\text{getWatchList } \text{state})\)\(\text{getF } \text{state}\)\(\text{and}\)
\(\text{InvariantWatchListsUniq } (\text{getWatchList } \text{state})\)\(\text{and}\)
\(\text{InvariantWatchListsCharacterization } (\text{getWatchList } \text{state})\) \(\text{getWatch1 state}\) \(\text{getWatch1 state}\) \(\text{getWatch2 state}\)
\(\text{and}\)
\(\text{InvariantWatchesDiffer } (\text{getF } \text{state})\) \(\text{getWatch1 state}\) \(\text{getWatch2 state}\)
show
\(\text{let state}' = \text{exhaustiveUnitPropagate } \text{state}\) in
\((\text{getSATFlag state}') = (\text{getSATFlag } \text{state}))\)
using \(\text{assms}\)
proof \((\text{induct state rule: exhaustiveUnitPropagate-dom.induct})\)
case \(\text{step state}'\)
note \(\text{ih} = \text{this}\)
s how \(?\text{case}\)
proof \((\text{cases } (\text{getConflictFlag state}') \lor (\text{getQ state}') = [])\)
case \(\text{True}\)
with \(\text{exhaustiveUnitPropagate.simps}[\text{of state}']\)
have \(\text{exhaustiveUnitPropagate } \text{state}' = \text{state}'\)
by simp

538
thus \( ?\text{thesis} \)

by \( \text{(simp only: Let-def)} \)

next

case False

let \( ?\text{state}'' = \text{applyUnitPropagate state'} \)

have \( \text{exhaustiveUnitPropagate state'} = \text{exhaustiveUnitPropagate state''} \)

using \( \text{exhaustiveUnitPropagate.simps[of state']} \)

using False

by simp

moreover

have \( \text{InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'' ) (getF ?state'')} \) and

\( \text{InvariantWatchListsUniq (getWatchList ?state'') and} \)

\( \text{InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'') (getWatch2 ?state'') and} \)

\( \text{InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')} \)

using \( \text{ih} \)

using \( \text{WatchInvariantsAfterAssertLiteral[of state' hd (getQ state'')} \)

\( \text{False} \)

unfolding \( \text{applyUnitPropagate-def} \)

by \( \text{(auto simp add: Let-def)} \)

moreover

have \( \text{getSATFlag ?state'' = getSATFlag state'} \)

unfolding \( \text{applyUnitPropagate-def} \)

using \( \text{assertLiteralEffect[of state' hd (getQ state'')} \text{False} \)

using \( \text{ih} \)

by \( \text{(simp add: Let-def)} \)

ultimately

show \( ?\text{thesis} \)

using \( \text{ih} \)

using False

by \( \text{(simp add: Let-def)} \)

qed

qed

lemma \( \text{exhaustiveUnitPropagatePreservesCurrentLevel:} \)

assumes

\( \text{exhaustiveUnitPropagate-dom state} \)

\( \text{InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)} \) and

\( \text{InvariantWatchListsUniq (getWatchList state) and} \)

\( \text{InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)} \)

\( \text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)} \)
and

InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)

shows

let state' = exhaustiveUnitPropagate state in
  currentLevel (getM state') = currentLevel (getM state)

using assms

proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
  case (step state')
  note ih = this
  show ?case
  proof (cases (getConflictFlag state') ∨ (getQ state') = [])
    case True
    with exhaustiveUnitPropagate.simps[of state']
    have exhaustiveUnitPropagate state' = state'
      by simp
    thus ?thesis
      by (simp only: Let-def)
  next
  case False
  let ?state'' = applyUnitPropagate state'

  have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state''
    using exhaustiveUnitPropagate.simps[of state']
    using False
    by simp
  moreover
  have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'') and
    InvariantWatchListsUniq (getWatchList ?state'') and
    InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'') (getWatch2 ?state'') and
    InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    using ih
    unfolding WatchInvariantsAfterAssertLiteral[of state' hd (getQ state') False]
    unfolding applyUnitPropagate-def
    by (auto simp add: Let-def)
  moreover
  have currentLevel (getM state') = currentLevel (getM ?state'')
    unfolding applyUnitPropagate-def
    using assertLiteralEffect[of state' hd (getQ state') False]
    unfolding currentLevel-def
    by (simp add: Let-def markedElementsAppend)
ultimately

show ?thesis
  using ih
  using False
  by (simp add: Let-def)
qed
qed

lemma InvariantsAfterExhaustiveUnitPropagate:
assumes
  exhaustiveUnitPropagate-dom state
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
  (getF state) and

  InvariantWatchListsUniq (getWatchList state) and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state)
  (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state)
  (getM state)
  InvariantConflictFlagCharacterization (getConflictFlag state) (getF state)
  (getM state)
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state)
  (getM state)
  InvariantUniqQ (getQ state)
  InvariantVarsQ (getQ state) F0 Vbl
  InvariantVarsM (getM state) F0 Vbl
  InvariantVarsF (getF state) F0 Vbl
shows
  let state' = exhaustiveUnitPropagate state in
  InvariantConsistent (getM state') \land
  InvariantUniq (getM state') \land
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state') (getF state') \land
  InvariantWatchListsUniq (getWatchList state') \land
  InvariantWatchListsCharacterization (getWatchList state') (getWatch1 state') (getWatch2 state') \land
  InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state') \land
  InvariantWatchesDiffer (getF state') (getWatch1 state') (getWatch2 state') \land
  InvariantWatchCharacterization (getF state') (getWatch1 state') (getWatch2 state') (getM state') \land
  InvariantConflictFlagCharacterization (getConflictFlag state')
(getF state') (getM state') ∧
InvariantQCharacterization (getConflictFlag state') (getQ state')
(getF state') (getM state') ∧
InvariantUniqQ (getQ state') ∧
InvariantVarsQ (getQ state') F0 Vbl ∧
InvariantVarsM (getM state') F0 Vbl ∧
InvariantVarsF (getF state') F0 Vbl

using assms

proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
case (step state')
  note ih = this

show ?case
proof (cases (getConflictFlag state') ∨ (getQ state') = [])
case True
  with exhaustiveUnitPropagate.simps[of state']
  have exhaustiveUnitPropagate state' = state'
    by simp
  thus ?thesis
    using ih
    by (auto simp only: Let-def)
next
case False
  let ?state'' = applyUnitPropagate state'

  have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state''
    using exhaustiveUnitPropagate.simps[of state']
    using False
    by simp

moreover
  have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'') and
    InvariantWatchListsUniq (getWatchList ?state'') and
    InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state''') (getWatch2 ?state''')
  using ih
  using WatchInvariantsAfterAssertLiteral[of state' hd (getQ state')] False
  unfolding applyUnitPropagate-def
  by (auto simp add: Let-def)
moreover
  have InvariantWatchCharacterization (getF ?state''') (getWatch1 ?state''') (getWatch2 ?state''') (getM ?state''')
    using ih

542
using $\text{InvariantWatchCharacterizationAfterApplyUnitPropagate}$ of state

unfolding $\text{InvariantQCharacterization-def}$
using False
by (simp add: Let-def)

moreover have $\text{InvariantQCharacterization}$ ($\text{getConflictFlag ?state''}$$) ($\text{getQ ?state''}$$) ($\text{getF ?state''}$$) ($\text{getM ?state''}$$)$
  using ih
  using $\text{InvariantQCharacterizationAfterApplyUnitPropagate}$ of state

using False
by (simp add: Let-def)

moreover have $\text{InvariantConflictFlagCharacterization}$ ($\text{getConflictFlag ?state''}$$) ($\text{getF ?state''}$$) ($\text{getM ?state''}$$)$
  using ih
  using $\text{InvariantConflictFlagCharacterizationAfterApplyUnitPropagate}$ of state

using False
by (simp add: Let-def)

moreover have $\text{InvariantUniqQ}$ ($\text{getQ ?state''}$$)$
  using ih
  using $\text{InvariantUniqQAfterApplyUnitPropagate}$ of state

using False
by (simp add: Let-def)

moreover have $\text{InvariantConsistent}$ ($\text{getM ?state''}$$)$
  using ih
  using $\text{InvariantConsistentAfterApplyUnitPropagate}$ of state

using False
by (simp add: Let-def)

moreover have $\text{InvariantUniq}$ ($\text{getM ?state''}$$)$
  using ih
  using $\text{InvariantUniqAfterApplyUnitPropagate}$ of state

using False
by (simp add: Let-def)

moreover have $\text{InvariantVarsM}$ ($\text{getM ?state''}$$) F0 Vbl $\text{InvariantVarsQ}$ ($\text{getQ ?state''}$$) F0 Vbl$
  using ih
  using $\neg \text{getConflictFlag state'} \lor \text{getQ state'} = []$$)
  using $\text{InvariantVarsAfterApplyUnitPropagate}$ of state' F0 Vbl
  by (auto simp add: Let-def)

moreover have $\text{InvariantVarsF}$ ($\text{getF ?state''}$$) F0 Vbl
  unfolding $\text{applyUnitPropagate-def}$

543
using assertLiteralEffect[of state' hd (getQ state') False]
using ih
by (simp add: Let-def)
ultimately
show ?thesis
  using ih
  using False
  by (simp add: Let-def)
qed
qed

lemma InvariantConflictClauseCharacterizationAfterExhaustivePropagate:
assumes
  exhaustiveUnitPropagate-dom state
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
  InvariantWatchListsUniq (getWatchList state) and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
shows
  let state' = exhaustiveUnitPropagate state in
  InvariantConflictClauseCharacterization (getConflictFlag state') (getConflictClause state') (getF state') (getM state')
using assms
proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
case (step state')
note ih = this
show ?case
proof (cases (getConflictFlag state') ∨ (getQ state') = [])
case True
with exhaustiveUnitPropagate.simps[of state']
have exhaustiveUnitPropagate state' = state'
  by simp
thus ?thesis
  using ih
  by (auto simp only: Let-def)
next
case False
let ?state'' = applyUnitPropagate state'
  have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state''
    using exhaustiveUnitPropagate.simps[of state']
using \( \text{False} \)
by \( \text{simp} \)
moreover
have InvariantWatchListsContainOnlyClausesFromF \( (\text{getWatchList } ?\text{state}'') (\text{getF } ?\text{state}'') \) and
\( \text{InvariantWatchListsUniq } (\text{getWatchList } ?\text{state}'') \) and
\( \text{InvariantWatchListsCharacterization } (\text{getWatchList } ?\text{state}'') (\text{getWatch1 } ?\text{state}'') (\text{getWatch2 } ?\text{state}'') \)
\( \text{InvariantWatchesEl } (\text{getF } ?\text{state}'') (\text{getWatch1 } ?\text{state}'') (\text{getWatch2 } ?\text{state}'') \)
\( \text{InvariantWatchesDiffer } (\text{getF } ?\text{state}'') (\text{getWatch1 } ?\text{state}'') (\text{getWatch2 } ?\text{state}'') \)
using \( \text{ih}(2) \) \( \text{ih}(3) \) \( \text{ih}(4) \) \( \text{ih}(5) \) \( \text{ih}(6) \) \( \text{ih}(7) \)
using \( \text{WatchInvariantsAfterAssertLiteral}\left[\text{of state'} \right] \) \( \text{hd} \) \( \text{getQ state'} \) \( \text{False} \)
unfolding applyUnitPropagate-def
by \( \text{auto simp add: Let-def} \)
moreover
have InvariantConflictClauseCharacterization \( (\text{getConflictFlag } ?\text{state}'') \) \( (\text{getConflictClause } ?\text{state}'') \) \( (\text{getF } ?\text{state}'') \) \( (\text{getM } ?\text{state}'') \)
using \( \text{ih}(2) \) \( \text{ih}(3) \) \( \text{ih}(4) \) \( \text{ih}(5) \) \( \text{ih}(6) \)
using \( \neg (\text{getConflictFlag state'} \lor \text{getQ state'} = []) \).
using \( \text{InvariantConflictClauseCharacterizationAfterApplyUnitPropagate}\left[\text{of state'} \right] \)
by \( \text{auto simp add: Let-def} \)
ultimately
show \( \vdash \text{thesis} \)
using \( \text{ih}(1) \) \( \text{ih}(2) \)
using \( \text{False} \)
by \( \text{simp only: Let-def} \) \( \text{blast} \)
qed
def

lemma InvariantsNoDecisionsWhenConflictNorUnitAfterExhaustivePropagate:
assumes
\( \text{exhaustiveUnitPropagate-dom } \text{state} \)
\( \text{InvariantConsistent } (\text{getM } \text{state}) \)
\( \text{InvariantUniq } (\text{getM } \text{state}) \)
\( \text{InvariantWatchListsContainOnlyClausesFromF } (\text{getWatchList } \text{state}) \) \( (\text{getF } \text{state}) \) and
\( \text{InvariantWatchListsUniq } (\text{getWatchList } \text{state}) \) \( \text{and} \)
\( \text{InvariantWatchListsCharacterization } (\text{getWatchList } \text{state}) (\text{getWatch1 } \text{state}) (\text{getWatch2 } \text{state}) \)
\( \text{InvariantWatchesEl } (\text{getF } \text{state}) (\text{getWatch1 } \text{state}) (\text{getWatch2 } \text{state}) \) \( \text{and} \)
\( \text{InvariantWatchesDiffer } (\text{getF } \text{state}) (\text{getWatch1 } \text{state}) (\text{getWatch2 } \text{state}) \)
\( \text{InvariantWatchCharacterization } (\text{getF } \text{state}) (\text{getWatch1 } \text{state}) (\text{getWatch2 } \text{state}) \)
545
shows

let state' = exhaustiveUnitPropagate state in

InvariantNoDecisionsWhenConflict (getF state') (getM state') (currentLevel (getM state'))

using assms

proof (induct state rule: exhaustiveUnitPropagate-dom.induct)

case (step state')

  note ih = this

  show ?case

  proof (cases (getConflictFlag state') \lor (getQ state') = [])

    case True

    with exhaustiveUnitPropagate.simps[of state']

    have exhaustiveUnitPropagate state' = state'
      by simp

    thus ?thesis

    using ih

    by (auto simp only: Let-def)

next

case False

let ?state'' = applyUnitPropagate state'


have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state'' using exhaustiveUnitPropagate.simps[of state']

using False

by simp

moreover

have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'') and

InvariantWatchListsUniq (getWatchList ?state'') and

InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'') (getWatch2 ?state'')

InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') and

InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')

using ih(5) ih(6) ih(7) ih(8) ih(9)
using WatchInvariantsAfterAssertLiteral[of state'] hd (getQ state') False]

  unfolding applyUnitPropagate-def
  by (auto simp add: Let-def)

moreover
  have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getM ?state'')
    using ih
    using InvariantWatchCharacterizationAfterApplyUnitPropagate[of state']

  unfolding InvariantQCharacterization-def
  using False
  by (simp add: Let-def)

moreover
  have InvariantQCharacterization (getConflictFlag ?state'') (getQ ?state'') (getF ?state'') (getM ?state'')
    using ih
    using InvariantQCharacterizationAfterApplyUnitPropagate[of state']

  using False
  by (simp add: Let-def)

moreover
  have InvariantConflictFlagCharacterization (getConflictFlag ?state'') (getF ?state'') (getM ?state'')
    using ih
    using InvariantConflictFlagCharacterizationAfterApplyUnitPropagate[of state']

  using False
  by (simp add: Let-def)

moreover
  have InvariantUniqQ (getQ ?state'')
    using ih
    using InvariantUniqQAAfterApplyUnitPropagate[of state']

  using False
  by (simp add: Let-def)

moreover
  have InvariantConsistent (getM ?state'')
    using ih
    using InvariantConsistentAfterApplyUnitPropagate[of state']

  using False
  by (simp add: Let-def)

moreover
  have InvariantUniq (getM ?state'')
    using ih
    using InvariantUniqAfterApplyUnitPropagate[of state']

  using False
  by (simp add: Let-def)

moreover
  have InvariantNoDecisionsWhenUnit (getF ?state'') (getM ?state'')
(currentLevel (getM ?state'))
  InvariantNoDecisionsWhenConflict (getF ?state') (getM ?state') (currentLevel (getM ?state'))
  using ih(5) ih(8) ih(11) ih(12) ih(14) ih(15)
  using InvariantNoDecisionsWhenConflictNorUnitAfterUnitPropagate[of state']
    by (auto simp add: Let-def)
  ultimately
  show ?thesis
    using ih(1) ih(2)
    using False
    by (simp add: Let-def)
qed
qed

lemma InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate:
  assumes
    exhaustiveUnitPropagate-dom state
    InvariantConsistent (getM state)
    InvariantUniq (getM state)
    InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
    InvariantWatchListsUniq (getWatchList state) and
    InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
    InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
    InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
    InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
    InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
    InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
    InvariantUniqQ (getQ state) and
    InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
  shows
    let state' = exhaustiveUnitPropagate state in
    InvariantGetReasonIsReason (getReason state') (getF state') (getM state') (set (getQ state'))
  using assms
proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
case (step state')
  note ih = this
  show ?case
proof (cases (getConflictFlag state') ∨ (getQ state') = [])

548
case True
with exhaustiveUnitPropagate.simps[of state]
have exhaustiveUnitPropagate state' = state'
  by simp
thus ?thesis
  using ih
  by (auto simp only: Let-def)
next
case False
let ?state'' = applyUnitPropagate state'

  have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state''
    using exhaustiveUnitPropagate.simps[of state]
    using False
    by simp
  moreover
  have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'') and
    InvariantWatchListsUniq (getWatchList ?state'') and
    InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'') (getWatch2 ?state'') and
    InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    using ih
    using WatchInvariantsAfterAssertLiteral[of state' hd (getQ state')]
    False
    unfolding applyUnitPropagate-def
    by (auto simp add: Let-def)
moreover
  have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getM ?state'')
    using ih
    using InvariantWatchCharacterizationAfterApplyUnitPropagate[of state]
    unfolding InvariantQCharacterization-def
    using False
    by (simp add: Let-def)
moreover
  have InvariantQCharacterization (getConflictFlag ?state'') (getQ ?state'') (getF ?state'') (getM ?state'')
    using ih
    using InvariantQCharacterizationAfterApplyUnitPropagate[of state]
    using False
    by (simp add: Let-def)
moreover


have InvariantConflictFlagCharacterization (getConflictFlag ?state") (getF ?state") (getM ?state")
  using ih
  using InvariantConflictFlagCharacterizationAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
moreover
have InvariantUniqQ (getQ ?state")
  using ih
  using InvariantUniqQAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
moreover
have InvariantConsistent (getM ?state")
  using ih
  using InvariantConsistentAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
moreover
have InvariantUniq (getM ?state")
  using ih
  using InvariantUniqAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
moreover
have InvariantGetReasonIsReason (getReason ?state") (getF ?state") (getM ?state") (set (getQ ?state"))
  using ih
  using InvariantGetReasonIsReasonAfterApplyUnitPropagate[of state]
  using False
  by (simp add: Let-def)
ultimately
show ?thesis
  using ih
  using False
  by (simp add: Let-def)
qed
qed

lemma InvariantEquivalentZLAfterExhaustiveUnitPropagate:
assumes
exhaustiveUnitPropagate-dom state
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantEquivalentZL (getF state) (getM state) Phi
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
\[(getF \text{ state}) \text{ and} \]
- \text{InvariantWatchListsUniq (getWatchList state)} \text{ and}
- \text{InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)}
- \text{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)}

\text{and}
- \text{InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)}
- \text{InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)}
- \text{InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)}
- \text{InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)}
- \text{InvariantUniqQ (getQ state)}

\text{shows}
- \text{let state}' = exhaustiveUnitPropagate state in}
  - \text{InvariantEquivalentZL (getF state') (getM state') Phi}

\text{using assms}

\text{proof (induct state rule: exhaustiveUnitPropagate-dom.induct)}
\text{case (step state')}
\text{note ih = this}
\text{show ?case}
\text{proof (cases (getConflictFlag state') \lor (getQ state') = [])}
\text{case True}
\text{with exhaustiveUnitPropagate.simps[of state']}\text{ have exhaustiveUnitPropagate state' = state'}
\text{ by simp}
\text{thus ?thesis}
\text{ using ih}
\text{ by (simp only: Let-def)}
\text{next}
\text{case False}
\text{let ?state'' = applyUnitPropagate state'}

\text{have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state''}
\text{using exhaustiveUnitPropagate.simps[of state']}
\text{using False}
\text{by simp}
\text{moreover}
\text{have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'') and}
\text{InvariantWatchListsUniq (getWatchList ?state'') and}
\text{InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getM ?state'') and}
InvariantWatchesDiffer (getF ?state'\(') (getWatch1 ?state'\(') (getWatch2 ?state'\(')

using ih
using WatchInvariantsAfterAssertLiteral[of state' \(\text{hd (getQ state')}\) False]

unfolding applyUnitPropagate-def
by (auto simp add: Let-def)
moreover
have InvariantWatchCharacterization (getF ?state'\(') (getWatch1 ?state'\(') (getWatch2 ?state'\(') (getM ?state'\(')
using ih
using InvariantWatchCharacterizationAfterApplyUnitPropagate[of state']

unfolding InvariantQCharacterization-def
using False
by (simp add: Let-def)
moreover
have InvariantQCharacterization (getConflictFlag ?state'\(') (getQ ?state'\(') (getF ?state'\(') (getM ?state'\(')
using ih
using InvariantQCharacterizationAfterApplyUnitPropagate[of state']

using False
by (simp add: Let-def)
moreover
have InvariantConflictFlagCharacterization (getConflictFlag ?state'\(') (getF ?state'\(') (getM ?state'\(')
using ih
using InvariantConflictFlagCharacterizationAfterApplyUnitPropagate[of state']

using False
by (simp add: Let-def)
moreover
have InvariantUniqQ (getQ ?state'\(')
using ih
using InvariantUniqQAfterApplyUnitPropagate[of state']

using False
by (simp add: Let-def)
moreover
have InvariantConsistent (getM ?state'\(')
using ih
using InvariantConsistentAfterApplyUnitPropagate[of state']

using False
by (simp add: Let-def)
moreover
have InvariantUniq (getM ?state'\(')
using ih
using InvariantUniqAfterApplyUnitPropagate[of state']

using False
by (simp add: Let-def)

moreover

have InvariantEquivalentZL (getF ?state'') (getM ?state'') Phi
  using ih
  using InvariantEquivalentZLAfterApplyUnitPropagate[of state']

Phi
  using False
  by (simp add: Let-def)

moreover

have currentLevel (getM state') = currentLevel (getM ?state'')
  unfolding applyUnitPropagate-def
  using assertLiteralEffect[of state' hd (getQ state') False]
  using ih
  unfolding currentLevel-def
  by (simp add: Let-def markedElementsAppend)

ultimately

show ?thesis
  using ih
  using False
  by (auto simp only: Let-def)

qed

qed

lemma conflictFlagOrQEmptyAfterExhaustiveUnitPropagate:
  assumes exhaustiveUnitPropagate-dom state
  shows let state' = exhaustiveUnitPropagate state in
      (getConflictFlag state') ∨ (getQ state' = [])
  using assms
proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
  case (step state')
  note ih = this
  show ?case
  proof (cases (getConflictFlag state') ∨ (getQ state' = []))
    case True
    with exhaustiveUnitPropagate.simps[of state]
    have exhaustiveUnitPropagate state' = state'
      by simp
    thus ?thesis
      using True
      by (simp only: Let-def)

  next
  case False
  let ?state'' = applyUnitPropagate state'

    have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state''
      using exhaustiveUnitPropagate.simps[of state']
using False
by simp
thus ?thesis
using th
using False
by (simp add: Let-def)
qed
qed

end

theory Initialization
imports UnitPropagate
begin

lemma InvariantsAfterAddClause:
fixes state::State and clause :: Clause and Vbl :: Variable set
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantUniqQ (getQ state)
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
currentLevel (getM state) = 0
(getConflictFlag state) ∨ (getQ state) = []
InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl

554
finite Vbl
vars clause ⊆ vars F0

shows
let state' = (addClause clause state) in
  InvariantConsistent (getM state') ∧
  InvariantUniq (getM state') ∧
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state') (getF state') ∧
  InvariantWatchListsUniq (getWatchList state') (getWatch1 state') (getWatch2 state') ∧
  InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state') ∧
  InvariantWatchesDiffer (getF state') (getWatch1 state') (getWatch2 state') ∧
  InvariantWatchCharacterization (getF state') (getWatch1 state') (getWatch2 state') ∧
  InvariantConflictFlagCharacterization (getConflictFlag state') (getConflictClause state') (getF state') (getM state') ∧
  InvariantQCharacterization (getConflictFlag state') (getQ state') (getF state') (getM state') ∧
  InvariantGetReasonIsReason (getReason state') (getF state') (getM state') (set (getQ state')) ∧
  InvariantUniqQ (getQ state') ∧
  InvariantVarsQ (getQ state') F0 Vbl ∧
  InvariantVarsM (getM state') F0 Vbl ∧
  InvariantVarsF (getF state') F0 Vbl ∧
currentLevel (getM state') = 0 ∧
((getConflictFlag state') ∨ (getQ state') = [])

proof−
let ?clause' = remdups (removeFalseLiterals clause (elements (getM state)))

have *: ∀ l. l el ?clause' → ¬ literalFalse l (elements (getM state))
unfolding removeFalseLiterals-def
by auto

have vars ?clause' ⊆ vars clause
  using varsSubsetValuation[of ?clause' clause]
unfolding removeFalseLiterals-def
by auto

hence vars ?clause' ⊆ vars F0
  using (vars clause ⊆ vars F0)
by simp

show ?thesis
proof (cases clauseTrue ?clause' (elements (getM state)))
case True
thus ?thesis
  using assms
  unfolding addClause-def
  by simp
next
case False
show ?thesis
proof (cases ?clause' = [])
case True
thus ?thesis
  using assms
  using ⟨¬ clauseTrue ?clause' (elements (getM state))⟩
  unfolding addClause-def
  by simp
next
case False
thus ?thesis
proof (cases length ?clause' = 1)
case True
  let ?state' = assertLiteral (hd ?clause') False state
  have addClause clause state = exhaustiveUnitPropagate ?state'
    using ⟨¬ clauseTrue ?clause' (elements (getM state))⟩
    using ⟨¬ ?clause' = []⟩
    unfolding addClause-def
    by (simp add: Let-def)
moreover
from ⟨?clause' ≠ []⟩
  have hd ?clause' ∈ set ?clause'
    using hd-in-set[of ?clause']
    by simp
with *
  have ¬ literalFalse (hd ?clause') (elements (getM state))
    by simp
  hence consistent (elements ((getM state) @ [[hd ?clause', False]]))
    using assms
    unfolding InvariantConsistent-def
    using consistentAppendElement[of elements (getM state) hd ?clause']
    by simp
  hence consistent (elements (getM ?state'))
    using assms
    using assertLiteralEffect[of state hd ?clause' False]
    by simp
moreover
from ⟨¬ clauseTrue ?clause' (elements (getM state))⟩
have uniq (elements (getM ?state'))
    using assms
    using assertLiteralEffect[of state hd ?clause' False]
    using 'hd ?clause' ∈ set ?clause'
    unfolding InvariantUniq-def
    by (simp add: uniqAppendIff clauseTrueIffContainsTrueLiteral)
moreover
have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state') (getF ?state') and
  InvariantWatchListsUniq (getWatchList ?state') and
  InvariantWatchListsCharacterization (getWatchList ?state') (getWatch1 ?state') (getWatch2 ?state') and
  InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state')
    using assms
    using WatchInvariantsAfterAssertLiteral[of state hd ?clause' False]
    by (auto simp add: Let-def)
moreover
have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getM ?state')
    using assms
    using InvariantWatchCharacterizationAfterAssertLiteral[of state hd ?clause' False]
    unfolding InvariantUniq-def
    unfolding InvariantConsistent-def
    by (simp add: Let-def)
moreover
have InvariantConflictFlagCharacterization (getConflictFlag ?state') (getF ?state') (getM ?state')
    using assms
    using InvariantConflictFlagCharacterizationAfterAssertLiteral[of state hd ?clause' False]
    unfolding InvariantUniq-def
    unfolding InvariantConsistent-def
    by (simp add: Let-def)
moreover
have InvariantConflictClauseCharacterization (getConflictFlag ?state') (getConflictClause ?state') (getF ?state') (getM ?state')
    using assms
    using InvariantConflictClauseCharacterizationAfterAssertLiteral[of state hd ?clause' False]
    by (simp add: Let-def)

557
moreover
let \( \text{state}'' = \text{state}'[\text{getM} := (\text{getM} \\text{state'}) @ [(\text{hd} \\text{clause'}, False)]] \)

have InvariantQCharacterization (getConflictFlag \text{state'}) (getQ \text{state'}) (getF \text{state'}) (getM \text{state'})
proof (cases getConflictFlag \text{state'})
case True
hence getConflictFlag \text{state'}
  using assms
  using assertLiteralConflictFlagEffect[of state hd \text{clause'} False]
by (auto simp add: Let-def)
thus \( \text{thesis} \)
  using assms
  unfolding InvariantQCharacterization-def
by simp
next
case False
with \((\text{getConflictFlag} \text{state}) \lor (\text{getQ} \text{state}) = []\)
have getQ \text{state} = []
  by simp
thus \( \text{thesis} \)
  using InvariantQCharacterizationAfterAssertLiteralNotInQ[of state hd \text{clause'} False]
  using assms
  using (uniq (elements (getM \text{state'})))
  using (consistent (elements (getM \text{state'})))
  unfolding InvariantConsistent-def
  unfolding InvariantUniq-def
  using assertLiteralEffect[of state hd \text{clause'} False]
by (auto simp add: Let-def)
qed

moreover
have InvariantUniqQ (getQ \text{state'})
  using assms
  using InvariantUniqQAfterAssertLiteral[of state hd \text{clause'} False]
by (simp add: Let-def)
moreover
have currentLevel (getM \text{state'}) = 0
  using assms
  using (\(\neg\) clauseTrue \text{clause'} (elements (getM \text{state})))
  using (\(\neg\) \text{clause'} = [])
  using assertLiteralEffect[of state hd \text{clause'} False]
unfolding \texttt{addClause-def}
unfolding \texttt{currentLevel-def}
by (simp add: Let-def markedElementsAppend)

moreover

hence \texttt{InvariantGetReasonIsReason}\ (\texttt{getReason \ ?state'}) \ (\texttt{getF \ ?state'}) \ (\texttt{set \ (getQ \ ?state')})
unfolding \texttt{InvariantGetReasonIsReason-def}
using \texttt{elementLevelLeqCurrentLevel[of - getM \ ?state']}
by auto

moreover

have \texttt{var \ (hd \ ?clause') \in \ vars F0}
using \texttt{⟨?clause' ≠ []⟩}
using \texttt{hd-in-set[of \ ?clause']}
using \texttt{⟨vars \ ?clause' ⊆ vars F0⟩}

using \texttt{clauseContainsItsLiteralsVariable[of \ hd \ ?clause' \ ?clause']}
by auto

hence \texttt{InvariantVarsQ \ (getQ \ ?state')} \ F0 Vbl
\texttt{InvariantVarsM \ (getM \ ?state')} \ F0 Vbl
\texttt{InvariantVarsF \ (getF \ ?state')} \ F0 Vbl
using \texttt{InvariantWatchListsContainOnlyClausesFromF \ (getWatchList \ state)} \ (\texttt{getF \ state})
using \texttt{InvariantWatchesEl} \ (\texttt{getF \ state}) \ (\texttt{getWatch1 \ state})
\texttt{(getWatch2 \ state)}
using \texttt{InvariantWatchListsUniq} \ (\texttt{getWatchList \ state})
using \texttt{InvariantWatchListsCharacterization} \ (\texttt{getWatchList \ state}) \ (\texttt{getWatch1 \ state}) \ (\texttt{getWatch2 \ state})
using \texttt{InvariantWatchesDiffer} \ (\texttt{getF \ state}) \ (\texttt{getWatch1 \ state})
\texttt{(getWatch2 \ state)}
using \texttt{InvariantWatchCharacterization} \ (\texttt{getF \ state}) \ (\texttt{getWatch1 \ state})
\texttt{(getWatch2 \ state)} \ (\texttt{getM \ state})
using \texttt{InvariantVarsF \ (getF \ state)} \ F0 Vbl
using \texttt{InvariantVarsM \ (getM \ state)} \ F0 Vbl
using \texttt{InvariantVarsQ \ (getQ \ state)} \ F0 Vbl
using \texttt{⟨consistent \ (elements \ (getM \ ?state'))⟩}
using \texttt{uniq \ (elements \ (getM \ ?state'))}
using \texttt{assertLiteralEffect[of \ state \ hd \ ?clause' \ False]}
using \texttt{assertLiteralEffect[of \ state \ hd \ ?clause' \ False \ F0 Vbl]}

using \texttt{InvariantVarsQAfterAssertLiteral[of \ state \ hd \ ?clause']}

using \texttt{InvariantVarsM-def}
unfolding \texttt{InvariantConsistent-def}
unfolding \texttt{InvariantUniq-def}
by (auto simp add: Let-def)

moreover

have \texttt{exhaustiveUnitPropagate-dom \ ?state'}
using \texttt{exhaustiveUnitPropagateTermination[of \ ?state' \ F0 Vbl]}
using \texttt{InvariantUniqQ \ (getQ \ ?state')}
using \texttt{InvariantWatchListsContainOnlyClausesFromF \ (getWatchList \ state)}
\(?\text{state}') (\text{getF ?state}')

\text{using (InvariantWatchListsUniq (getWatchList ?state'))}
\text{using (InvariantWatchListsCharacterization (getWatchList ?state')) (getWatch1 ?state') (getWatch2 ?state')}
\text{using (InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state'))}
\text{using (InvariantWatchesDiffer (getF ?state') (getWatch1 ?state') (getWatch2 ?state'))}
\text{using (InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getM ?state'))}
\text{using (InvariantConflictFlagCharacterization (getConflictFlag ?state') (getF ?state') (getM ?state'))}
\text{using (InvariantConflictClauseCharacterizationAfterExhaustiveUnitPropagate[of ?state'])}
\text{using (exhaustiveUnitPropagatePreservesCurrentLevel[of ?state'])}
\text{using (InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate[of ?state'])}
\text{using (assms)}
\text{by (auto simp only: Let-def)}

next
\text{case False}
\text{thus ?thesis}
\text{proof (cases clauseTautology ?clause')}
\text{case True}
\text{thus ?thesis}
\text{using assms}
\text{using (\neg ?clause' = [])}
\text{using (\neg clauseTrue ?clause' (elements (getM state)))}
\text{using (length ?clause' \neq 1)}
unfolding addClause-def
by simp

next
  case False
  from (¬ ?clause' = []) (length ?clause' ≠ 1)
  have length ?clause' > 1
    by (induct (?clause')) auto

  hence nth ?clause' 0 ≠ nth ?clause' 1
    using distinct-remdups[of ?clause']
    using nth-eq-iff-index-eq[of ?clause' 0 1]
    using (¬ ?clause' = [])
    by auto

  let ?state' = let clauseIndex = length (getF state) in
    let state' = state() getF := (getF state) @
      [?clause'] in
    let state'' = setWatch1 clauseIndex (nth ?clause'
      0) state' in
    let state''' = setWatch2 clauseIndex (nth ?clause'
      1) state'' in

    have InvariantWatchesEl (getF ?state') (getWatch1 ?state')
      (getWatch2 ?state')
      using InvariantWatchesEl (getF state) (getWatch1 state)
      (getWatch2 state):
      using (length ?clause' > 1);
      using (¬ ?clause' ≠ []);
      using nth-mem[of 0 ?clause']
      using nth-mem[of 1 ?clause']
      unfolding InvariantWatchesEl-def
      unfolding setWatch1-def
      unfolding setWatch2-def
      by (auto simp add: Let-def nth-append)

  moreover
  have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state')
    (getWatch2 ?state')
    using InvariantWatchesDiffer (getF state) (getWatch1 state)
    (getWatch2 state):
    using (nth ?clause' 0 ≠ nth ?clause' 1);
    unfolding InvariantWatchesDiffer-def
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def)

  moreover
  have InvariantWatchListsContainOnlyClausesFromF (getWatchList
    ?state') (getF ?state')
    using InvariantWatchListsContainOnlyClausesFromF

561
(getWatchList state) (getF state):
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def) (force)+
mOREOVER
  have InvariantWatchListsCharacterization (getWatchList ?state) (getWatch1 ?state) (getWatch2 ?state)
  using :InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
  using :InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  unfolding InvariantWatchListsCharacterization-def
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
mOREOVER
  have InvariantWatchCharacterization (getF ?state) (getWatch1 ?state) (getWatch2 ?state) (getM ?state)
proof -
{  fix c
  assume 0 ≤ c ∧ c < length (getF ?state)
  fix www1 www2
  assume Some www1 = (getWatch1 ?state c) Some www2 = (getWatch2 ?state c)
  have watchCharacterizationCondition www1 www2 (getM ?state) (nth (getF ?state) c) ∧
    watchCharacterizationCondition www2 www1 (getM ?state) (nth (getF ?state) c)
  proof (cases c < length (getF state))
    case True
    hence (nth (getF ?state) c) = (nth (getF state) c)
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def nth-append)
    have Some www1 = (getWatch1 state c) Some www2 = (getWatch2 state c)
  using True
  using Some www1 = (getWatch1 ?state c) ;Some www2 = (getWatch2 ?state c)
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
  thus ?thesis
  using :InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)

562
unfolding InvariantWatchCharacterization-def
using \((\text{nth} \ (\text{getF} ?\text{state}') \ c) = (\text{nth} \ (\text{getF} \ \text{state}) \ c)\)
using True
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def)
next
case False
with \(0 \leq c \land c < \text{length} \ (\text{getF} ?\text{state}')\)
have \(c = \text{length} \ (\text{getF} \ \text{state})\)
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def)
from \((\text{InvariantWatchesEl} \ (\text{getF} ?\text{state}') \ (\text{getWatch1} ?\text{state}') \ (\text{getWatch2} ?\text{state}'))\)
obtain w1 w2
where
\(w1 \ \text{el} \ ?\text{clause}' \ w2 \ \text{el} \ ?\text{clause}'\)
\(\text{getWatch1} \ ?\text{state}' \ (\text{length} \ (\text{getF} \ \text{state})) = \text{Some} \ w1\)
\(\text{getWatch2} ?\text{state}' \ (\text{length} \ (\text{getF} \ \text{state})) = \text{Some} \ w2\)
unfolding InvariantWatchesEl-def
unfolding setWatch2-def
unfolding setWatch1-def
by (auto simp add: Let-def)

hence \(w1 = \text{www1} \ \text{and} \ w2 = \text{www2}\)

obtain \(\text{Some} \ \text{www1} = (\text{getWatch1} ?\text{state}' \ c) \cdot \text{Some} \ \text{www2} = (\text{getWatch2} ?\text{state}' \ c)\)
where
\(w1 \ \text{el} \ ?\text{clause}' \ w2 \ \text{el} \ ?\text{clause}'\)
\(\text{getWatch1} \ ?\text{state}' \ (\text{length} \ (\text{getF} \ \text{state})) = \text{Some} \ w1\)
\(\text{getWatch2} ?\text{state}' \ (\text{length} \ (\text{getF} \ \text{state})) = \text{Some} \ w2\)

unfolding InvariantWatchesEl-def
unfolding setWatch2-def
unfolding setWatch1-def
by (auto simp add: Let-def)

thus \(?\text{thesis}\)

unfolding watchCharacterizationCondition-def
unfolding setWatch2-def
unfolding setWatch1-def
by (auto simp add: Let-def)

qed
} thus \(?\text{thesis}\)
unfolding InvariantWatchCharacterization-def
by auto

qed

moreover
have \(\forall \ l. \ \text{length} \ (\text{getF} \ \text{state}) \ \notin \ \text{set} \ (\text{getWatchList} \ \text{state} \ l)\)
using :InvariantWatchListsContainOnlyClausesFromF
(getWatchList state) (getF state): 
unfolding InvariantWatchListsContainOnlyClausesFromF-def
by auto
hence InvariantWatchListsUniq (getWatchList ?state)
using :InvariantWatchListsUniq (getWatchList state);
using (nth ?clause' 0 ≠ nth ?clause' 1);
unfolding InvariantWatchListsUniq-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def uniqAppendIff)
moreover
from * 
have ¬ clauseFalse ?clause' (elements (getM state))
  using (?clause' ≠ [])
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
hence InvariantConflictFlagCharacterization (getConflictFlag ?state)
  (getF ?state') (getM ?state)
using :InvariantConflictFlagCharacterization (getConflictFlag state)
  (getF state) (getM state):
unfolding InvariantConflictFlagCharacterization-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def formulaFalseIffContainsFalse-Clause)
moreover
have ¬ (∃ l. isUnitClause ?clause' l (elements (getM state)))
proof–
  { 
  assume ¬ ?thesis
  then obtain l 
  where isUnitClause ?clause' l (elements (getM state))
  by auto
  hence l el ?clause'
  unfolding isUnitClause-def
  by simp
  have ∃ l'. l' el ?clause' ∧ l ≠ l'
  proof–
  from (length ?clause' > 1)
  obtain a1::Literal and a2::Literal
  where a1 el ?clause' a2 el ?clause' a1 ≠ a2
  using lengthGtOneTwoDistinctElements[of ?clause']
  by (auto simp add: uniqDistinct) (force)
  thus ?thesis
  proof (cases a1 = l)
  case True
  thus ?thesis
  using ⟨a1 ≠ a2⟩ ⟨a2 el ?clause'⟩
  by auto
  
  564
next
  case False
  thus ?thesis
  using \langle a_1 \text{ el } \text{ ?clause}' \rangle
  by auto
qed
qed
then obtain l':\text{Literal}
  where l \neq l' \text{ el } \text{ ?clause}'
  by auto
with *
  have \neg \text{ literalFalse} l' (\text{elements} (\text{getM} \text{ state}))
  by simp
  hence False
  using (\text{isUnitClause} \text{ ?clause}' l (\text{elements} (\text{getM} \text{ state})))
  using (l \neq l') (l' \text{ el } \text{ ?clause}')
  unfolding \text{isUnitClause-def}
  by auto
} thus ?thesis
by auto
qed
hence \text{InvariantQCharacterization} (\text{getConflictFlag} \text{ ?state}') (\text{getQ} \text{ ?state}') (\text{getF} \text{ ?state}') (\text{getM} \text{ ?state}')
using \text{assms}
unfolding \text{InvariantQCharacterization-def}
unfolding \text{setWatch2-def}
unfolding \text{setWatch1-def}
by (auto simp add: \text{Let-def})
moreover
  have \text{InvariantConflictClauseCharacterization} (\text{getConflictFlag} \text{ state}) (\text{getConflictClause} \text{ state}) (\text{getF} \text{ state} @ [\text{?clause}']) (\text{getM} \text{ state})
proof (cases \text{getConflictFlag} \text{ state})
  case False
  thus ?thesis
  unfolding \text{InvariantConflictClauseCharacterization-def}
  by simp
next
  case True
  hence \text{getConflictClause} \text{ state} < \text{length} (\text{getF} \text{ state})
  using (\text{InvariantConflictClauseCharacterization} (\text{getConflictFlag} \text{ state}) (\text{getConflictClause} \text{ state}) (\text{getF} \text{ state}) (\text{getM} \text{ state}))
  unfolding \text{InvariantConflictClauseCharacterization-def}
  by (auto simp add: \text{Let-def})
  hence nth ((\text{getF} \text{ state}) @ [\text{?clause}']) (\text{getConflictClause} \text{ state}) =
  nth (\text{getF} \text{ state}) (\text{getConflictClause} \text{ state})
  by (simp add: nth-append)
  thus ?thesis
  using \text{InvariantConflictClauseCharacterization} (\text{getConflictFlag}

565
state) (getConflictClause state) (getF state) (getM state))  
  unfolding InvariantConflictClauseCharacterization-def  
  by (auto simp add: Let-def clauseFalseAppendValuation)

qed

moreover

have InvariantGetReasonIsReason (getReason ?state′) (getF ?state′) (getM ?state′) (set (getQ ?state′))
  using ⟨currentLevel (getM state) = 0⟩
  using elementLevelLeqCurrentLevel[of - getM state]
  unfolding setWatch1-def
  unfolding setWatch2-def
  unfolding InvariantGetReasonIsReason-def
  by (simp add: Let-def)

moreover

have InvariantVarsF (getF ?state′) F0 Vbl
  using ⟨InvariantVarsF (getF state) F0 Vbl⟩
  using ⟨vars ?clause′ ⊆ vars F0⟩
  using varsAppendFormulae[of getF state [?clause′]]
  unfolding setWatch2-def
  unfolding setWatch1-def
  unfolding InvariantVarsF-def
  by (auto simp add: Let-def)

ultimately

show ?thesis
  using assms
  using ⟨length ?clause′ > 1⟩
  using ⟨?clause′ = []⟩
  using ⟨¬ clauseTrue ?clause′ (elements (getM state))⟩
  using ⟨length ?clause′ ≠ 1⟩
  using ⟨¬ clauseTautology ?clause′⟩
  unfolding addClause-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

qed

lemma InvariantEquivalentZLAfterAddClause:
fixes Phi :: Formula and clause :: Clause and state :: State and Vbl :: Variable set
assumes
  *:(getSATFlag state = UNDEF ∧ InvariantEquivalentZL (getF state) (getM state) Phi) ∨
    (getSATFlag state = FALSE ∧ ¬ satisfiable Phi)
  InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantUniqQ (getQ state) (getConflictFlag state) ∨ (getQ state) = []
currentLevel (getM state) = 0
InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
finite Vbl
vars clause ⊆ vars F0

shows
let state′ = addClause clause state in
let Phi′ = Phi @ [clause] in
let Phi'' = (if (clauseTautology clause) then Phi else Phi′) in
(getSATFlag state′ = UNDEF ∧ InvariantEquivalentZL (getF state′) (getM state′)) ∨
(getSATFlag state′ = FALSE ∧ ¬satisfiable Phi″)

proof–
let ?clause′ = remdups (removeFalseLiterals clause (elements (getM state)))
from (currentLevel (getM state) = 0)
have getM state = prefixToLevel 0 (getM state)
  by (rule currentLevelZeroTrailEqualsItsPrefixToLevelZero)

have **: ∀ l. l el ?clause′ → ¬literalFalse l (elements (getM state))
  unfolding removeFalseLiterals-def
  by auto

have vars ?clause′ ⊆ vars clause
  using varsSubsetValuation[of ?clause′ clause]
  unfolding removeFalseLiterals-def
  by auto

hence vars ?clause′ ⊆ vars F0
  using (vars clause ⊆ vars F0)
by simp

show ?thesis
proof (cases clauseTrue \ ?clause' \ (elements \ (getM \ state)))
  case True
  show ?thesis
  proof
    from True
    have clauseTrue clause \ (elements \ (getM \ state))
      using clauseTrueRemoveDuplicateLiterals
        \ [of \ removeFalseLiterals \ clause \ (elements \ (getM \ state)) \ elements
        \ (getM \ state)]
      using clauseTrueRemoveFalseLiterals
        \ [of \ elements \ (getM \ state) \ clause]
      using InvariantConsistent \ (getM \ state)
      unfolding InvariantConsistent-def
      by simp
    show ?thesis
    proof (cases getSATFlag \ state = UNDEF)
      case True
      thus ?thesis
        using *
        using \ (clauseTrue \ clause \ (elements \ (getM \ state)))
        using \ (getM \ state = \ prefixToLevel \ \ 0 \ \ (getM \ state))
        using satisfiedClauseCanBeRemoved
          \ [of \ getF \ state \ (elements \ (prefixToLevel \ \ 0 \ \ (getM \ state))) \ Phi
          \ clause]
        using \ (clauseTrue \ ?clause' \ (elements \ (getM \ state)))
        unfolding addClause-def
        unfolding InvariantEquivalentZL-def
        by auto
      next
      case False
      thus ?thesis
        using *
        using \ (clauseTrue \ ?clause' \ (elements \ (getM \ state)))
        using satisfiedAppend \ [of \ Phi \ [clause]]
        unfolding addClause-def
        by force
    qed
  qed
next
  case False
  show ?thesis
  proof (cases \ ?clause' = [])
    case True
    show ?thesis
    proof (cases getSATFlag \ state = UNDEF)
      case True
thus \( \text{?thesis} \)
using *
using falseAndDuplicateLiteralsCanBeRemoved
of getF state (elements (prefixToLevel 0 (getM state))) [] Phi clause
using \( \text{getM state} = \text{prefixToLevel 0 (getM state)} \)
using formulaWithEmptyClauseIsUnsatisfiable[of (getF state @ val2form (elements (getM state)) @ [[]])]
using satisfiableEquivalent
using \( \langle \text{?clause'} = [] \rangle \)
unfolding addClause-def
unfolding InvariantEquivalentZL-def
using satisfiableAppendTautology
by auto

next
case False
thus \( \text{?thesis} \)
using \( \langle \text{?clause'} = [] \rangle \)
using *
using satisfiableAppend[of Phi [clause]]
unfolding addClause-def
by force

qed

next
case False
thus \( \text{?thesis} \)
proof (cases length ?clause' = 1)
case True
from \( \langle \text{length ?clause'} = 1 \rangle \)
have \( \langle \text{hd ?clause'} = ?clause' \rangle \)
using lengthOneCharacterisation[of ?clause']
by simp

with \( \langle \text{length ?clause'} = 1 \rangle \)
have val2form (elements (getM state)) @ [?clause'] = val2form ((elements (getM state)) @ ?clause')
using val2formAppend[of elements (getM state) ?clause']
using val2formOfSingleLiteralValuation[of ?clause']
by auto

let \( \text{?state'} = \text{assertLiteral (hd ?clause')} \)
False state
have addClause clause state = exhaustiveUnitPropagate ?state'
using \( \langle \neg \text{clauseTrue ?clause' (elements (getM state))} \rangle \)
using \( \langle \neg ?clause' = [] \rangle \)
using \( \langle \text{length ?clause'} = 1 \rangle \)
unfolding addClause-def
by (simp add: Let-def)
moreover
from \( \langle ?clause' \neq [] \rangle \)
have \( \text{hd } ?\text{clause}' \in \text{set } ?\text{clause}' \)
using \( \text{hd-in-set[of } ?\text{clause}'] \)
by simp
with **
have \( \neg \text{literalFalse (hd } ?\text{clause}') \) (elements (getM state))
by simp
hence consistent (elements ((getM state) @ [(hd } ?\text{clause}', False)])
using assms
unfolding InvariantConsistent-def
using consistentAppendElement[of elements (getM state) hd ?\text{clause}']
by simp
hence consistent (elements (getM ?state'))
using assms
using assertLiteralEffect[of state hd ?\text{clause}' False]
by simp
moreover
from \( \neg \text{clauseTrue } ?\text{clause}' \) (elements (getM state))
have uniq (elements (getM ?state'))
using assms
using assertLiteralEffect[of state hd ?\text{clause}' False]
using \( \text{hd } ?\text{clause}' \in \text{set } ?\text{clause}' \)
unfolding InvariantUniq-def
by (simp add: uniqAppendIff clauseTrueIffContainsTrueLiteral)
moreover
have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state') (getF ?state')
and InvariantWatchListsUniq (getWatchList ?state')
and InvariantWatchListsCharacterization (getWatchList ?state')
(getWatch1 ?state') (getWatch2 ?state')
InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state')
and InvariantWatchesDiffer (getF ?state') (getWatch1 ?state') (getWatch2 ?state')
using assms
using WatchInvariantsAfterAssertLiteral[of state hd ?\text{clause}'] False]
by (auto simp add: Let-def)
moreover
have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getM ?state')
using assms
using InvariantWatchCharacterizationAfterAssertLiteral[of state hd ?\text{clause}' False]
using uniq (elements (getM ?state'))
using consistent (elements (getM ?state'))
unfolding InvariantConsistent-def
unfolding InvariantUniq-def

570
using `assertLiteralEffect[of state hd ?clause' False]
by (simp add: Let-def)
moreover
have `InvariantConflictFlagCharacterization (getConflictFlag ?state') (getF ?state') (getM ?state')
  using assms
  using `InvariantConflictFlagCharacterizationAfterAssertLiteral[of state hd ?clause' False]
  unfolding `InvariantConsistent-def
  using `assertLiteralEffect[of state hd ?clause' False]
  by (simp add: Let-def)
moreover
have `InvariantQCharacterization (getConflictFlag ?state') (getQ ?state') (getF ?state') (getM ?state')
proof (cases getConflictFlag state)
case True
hence getConflictFlag ?state'
  using assms
    using `assertLiteralConflictFlagEffect[of state hd ?clause' False]
  unfolding `InvariantConsistent-def
  unfolding `InvariantUniq-def
  using `assertLiteralEffect[of state hd ?clause' False]
  by (auto simp add: Let-def)
thus ?thesis
  using assms
  unfolding `InvariantQCharacterization-def
  by simp
next
case False
with `(getConflictFlag state) ∨ (getQ state) = []'
have getQ state = []
  by simp
thus ?thesis
using `InvariantQCharacterizationAfterAssertLiteralNotInQ[of state hd ?clause' False]
  using assms
  using `uniq (elements (getM ?state'))
  using `consistent (elements (getM ?state'))
  unfolding `InvariantConsistent-def
  unfolding `InvariantUniq-def
  using `assertLiteralEffect[of state hd ?clause' False]
  by (auto simp add: Let-def)
qed
moreover
have `InvariantUniqQ (getQ ?state')
using assms
using InvariantUniqQAAfterAssertLiteral[of state hd ?clause']

False]

by (simp add: Let-def)
moreover
have currentLevel (getM ?state') = 0
using assms
using (~ clauseTrue ?clause' (elements (getM state))
using (~ ?clause' = []);
using assertLiteralEffect[of state hd ?clause' False]
unfolding addClause-def
unfolding currentLevel-def
by (simp add: Let-def markedElementsAppend)
moreover
have var (hd ?clause') ∈ vars F0
using (?clause' ≠ [])
using hd-in-set[of ?clause']
using (vars ?clause' ⊆ vars F0.
using clauseContainsItsLiteralsVariable[of hd ?clause' ?clause']
by auto
hence InvariantVarsM (getM ?state') F0 Vbl
InvariantVarsQ (getQ ?state') F0 Vbl
InvariantVarsF (getF ?state') F0 Vbl
using (InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state))
(getWatch2 state):
using (InvariantWatchesEl (getF state) (getWatch1 state)
(getWatch2 state))
using (InvariantWatchListsUniq (getWatchList state))
using (InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state))
using (InvariantWatchesDiffer (getF state) (getWatch1 state)
(getWatch2 state))
using (InvariantWatchCharacterization (getF state) (getWatch1 state)
(getWatch2 state) (getM state))
using (InvariantVarsF (getF state) F0 Vbl)
using (InvariantVarsM (getM state) F0 Vbl)
using (InvariantVarsQ (getQ state) F0 Vbl)
using (consistent (elements (getM ?state')))
using (uniq (elements (getM ?state')));
using assertLiteralEffect[of state hd ?clause' False]
using varsAppendValuation[of elements (getM state) [hd
?clause']]
using InvariantVarsQAAfterAssertLiteral[of state hd ?clause'
False F0 Vbl]

unfolding InvariantVarsM-def
unfolding InvariantConsistent-def
unfolding InvariantUniq-def
by (auto simp add: Let-def)
moreover
have exhaustiveUnitPropagate-dom ?state'
using exhaustiveUnitPropagateTermination[of ?state' F0 Vbl]
using ⟨InvariantUniqQ (getQ ?state')⟩
using ⟨InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state') (getF ?state')⟩
using ⟨InvariantWatchListsUniq (getWatchList ?state')⟩
using ⟨InvariantWatchListsCharacterization (getWatchList ?state') (getWatch1 ?state') (getWatch2 ?state')⟩
using ⟨InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state')⟩
using ⟨InvariantWatchesDiffer (getF ?state') (getWatch1 ?state') (getWatch2 ?state')⟩
using ⟨InvariantQCharacterization (getConflictFlag ?state') (getQ ?state') (getF ?state') (getM ?state')⟩
using ⟨InvariantConflictFlagCharacterization (getConflictFlag ?state') (getF ?state') (getM ?state')⟩
using ⟨consistent (elements (getM ?state'))⟩
using ⟨uniq (elements (getM ?state'))⟩
using ⟨finite Vbl⟩
using ⟨InvariantVarsM (getM ?state') F0 Vbl⟩
using ⟨InvariantVarsQ (getQ ?state') F0 Vbl⟩
using ⟨InvariantVarsF (getF ?state') F0 Vbl⟩
unfolding InvariantConsistent-def
unfolding InvariantUniq-def
by simp
moreover
have ¬ clauseTautology clause
proof
{
  assume ¬ ?thesis
  then obtain l'
    where l' el clause opposite l' el clause
    by (auto simp add: clauseTautologyCharacterization)
  have False
  proof (cases l' el ?clause')
    case True
    have opposite l' el ?clause'
    proof
      { assume ¬ ?thesis
        hence literalFalse l' (elements (getM state))
        using l' el clause
        using ⟨opposite l' el clause⟩
        using ⟨¬ clauseTrue ?clause' (elements (getM state))⟩
        using clauseTrueIffContainsTrueLiteral[of ?clause'
          elements (getM state)]
        unfolding removeFalseLiterals-def}
by auto
dence False
using \( \ell' \) el ?clause',
 unfolding removeFalseLiterals-def
by auto
} thus \( ?\text{thesis} \)
by auto
qed

have \( \forall x. \) el ?\( \text{clause}' \) \( \rightarrow x = \ell' \)
using \( \ell' \) el ?\( \text{clause}' \)
using \( \text{length} \ ?\text{clause}' = 1 \)
using lengthOneImpliesOnlyElement[of \( ?\text{clause}' l' \)]
by simp
thus \( ?\text{thesis} \)
using \( \text{opposite} \ \ell' \) el ?\( \text{clause}' \)
by auto

next
case False
dence literalFalse \( \ell' \) (elements (getM state))
using \( \ell' \) el \( \text{clause}' \)
 unfolding removeFalseLiterals-def
by simp
dence \( \neg \) literalFalse \( \text{opposite} \ \ell' \) (elements (getM state))
using \( \text{InvariantConsistent} \ (\text{getM state}) \)
 unfolding InvariantConsistent-def
by (auto simp add: inconsistentCharacterization)
dence \( \text{opposite} \ \ell' \) el ?\( \text{clause}' \)
using \( \text{opposite} \ \ell' \) el ?\( \text{clause}' \)
 unsus unfolding removeFalseLiterals-def
by auto
dence \( ?\text{thesis} \)
using \( \neg \) literalFalse \( \ell' \) (elements (getM state))
using \( \neg \) clauseTrue \( ?\text{clause}' \) (elements (getM state))
by (simp add: clauseTrueIffContainsTrueLiteral)
qed
} thus \( ?\text{thesis} \)
by auto
qed

moreover

note clc = calculation

show \( ?\text{thesis} \)
proof (cases getSATFlag state = UNDEF)
case True
dence InvariantEquivalentZL (getF state) (getM state) \( \text{Phi} \)
using assms
by simp
dence InvariantEquivalentZL (getF ?state') (getM ?state')
(\( \text{Phi} \) @ [\( \text{clause} \)])
using *
using falseAndDuplicateLiteralsCanBeRemoved
[of getF state (elements (prefixToLevel 0 (getM state))) !!]

Phi clause

using (∃ hd ?clause' = ?clause')
using (getM state = prefixToLevel 0 (getM state))
using (currentLevel (getM state) = 0)
using prefixToLevelAppend[of 0 getM state ([hd ?clause', False)])

using (InvariantWatchesEl (getF state) (getWatch1 state)
(getWatch2 state))
using :InvariantWatchListsContainOnlyClausesFromF
(getWatchList state) (getF state):
using assertLiteralEffect[of state hd ?clause' False]
using :val2form (elements (getM state)) @ [?clause'] =
val2form ((elements (getM state)) @ ?clause')
using (∼ ?clause' = [])
using (∼ clauseTrue ?clause' (elements (getM state)))
using (length ?clause' = 1)
using (getSATFlag state = UNDEF)
unfolding addClause-def
unfolding InvariantEquivalentZL-def
by (simp add: Let-def)
hence let state'' = addClause clause state in
InvariantEquivalentZL (getF state') (getM state'') (Phi @
[clause]) ∧
getSATFlag state'' = getSATFlag state
using cle
using InvariantEquivalentZLAfterExhaustiveUnitPropagate[of
?state' Phi @ [clause]]
using exhaustiveUnitPropagatePreservedVariables[of ?state']
using assms
unfolding InvariantConsistent-def
unfolding InvariantUniq-def
using assertLiteralEffect[of state hd ?clause' False]
by (auto simp only: Let-def)
thus ?thesis
using True
using (∼ clauseTautology clause)
by (auto simp only: Let-def split: split-if)

next
case False
hence getSATFlag state = FALSE ∼ satisfiable Phi
using *
by auto
hence getSATFlag ?state' = FALSE
using assertLiteralEffect[of state hd ?clause' False]
using assms
by simp

575
hence getSATFlag (exhaustiveUnitPropagate ?state’) = FALSE

    using clc
    using exhaustiveUnitPropagatePreservedVariables[of ?state’]
    by (auto simp only: Let-def)

moreover
have ¬ satisfiable (Phi @ [clause])
    using satisfiableAppend[of Phi [clause]]
    using (¬ satisfiable Phi)
    by auto

ultimately
show ?thesis
    using clc
    using (¬ clauseTautology clause)
    by (simp only: Let-def) simp

qed

next
case False

thus ?thesis

proof (cases clauseTautology ?clause’)
case True

moreover
hence clauseTautology clause
    unfolding removeFalseLiterals-def
    by (auto simp add: clauseTautologyCharacterization)

ultimately
show ?thesis
    using *
    using (¬ ?clause’ = [])
    using (¬ clauseTrue ?clause’ (elements (getM state)))
    using (length ?clause’ ≠ 1)
    using satisfiableAppend[of Phi [clause]]
    unfolding addClause-def

    by (auto simp add: Let-def)

next
case False

have ¬ clauseTautology clause

proof (cases l’ el ?clause’)

{ assume ¬ ?thesis
then obtain l’
  where l’ el clause opposite l’ el clause
  by (auto simp add: clauseTautologyCharacterization)

have False
proof (cases l’ el ?clause’)
case True
hence ¬ opposite l’ el ?clause’
  using (¬ clauseTautology ?clause’)
by (auto simp add: clauseTautologyCharacterization)

hence literalFalse (opposite l') (elements (getM state))
  using (opposite l' el clause)
  unfolding removeFalseLiterals-def
  by auto
thus ?thesis
  using ¬ clauseTrue ?clause' (elements (getM state));
  using l' el ?clause'\;
  by (simp add: clauseTrueIffContainsTrueLiteral)

next
  case False
  hence literalFalse l' (elements (getM state))
    using (l' el clause)
    unfolding removeFalseLiterals-def
    by auto
hence ¬ literalFalse (opposite l') (elements (getM state))
  using InvariantConsistent (getM state);
  unfolding InvariantConsistent-def
  by (auto simp add: inconsistentCharacterization)

hence opposite l' el ?clause'
  using (opposite l' el clause)
  unfolding removeFalseLiterals-def
  by auto
thus ?thesis
  using ¬ clauseTrue ?clause' (elements (getM state));
  using literalFalse l' (elements (getM state));
  by (simp add: clauseTrueIffContainsTrueLiteral)

qed

} thus ?thesis
  by auto
qed

show ?thesis
proof (cases getSATFlag state = UNDEF)
  case True
  show ?thesis
    using *
    using falseAndDuplicateLiteralsCanBeRemoved
    [of getF state (elements (prefixToLevel 0 (getM state)))] []

\[ \text{Phi clause}\]

  using (getM state = prefixToLevel 0 (getM state));
  using (\range ?clause' = []);
  using ¬ clauseTrue ?clause' (elements (getM state));
  using (length ?clause' \neq 1);
  using ¬ clauseTautology ?clause'\;
  using ¬ clauseTautology clause\;
  using getSATFlag state = UNDEF;
  unfolding addClause-def
  unfolding InvariantEquivalentZL-def
  unfolding setWatch1-def

577
lemma InvariantsAfterInitializationStep:
fixes
  state :: State and Phi :: Formula and Vbl :: Variable set
assumes
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
  InvariantWatchListsUniq (getWatchList state) and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and

unfolding setWatch2-def
using clauseOrderIrrelevant[of getF state [?clause'] val2form
  (elements (getM state)) []]
using equivalentFormuleTransitivity[of
getF state @ remdups (removeFalseLiterals clause (elements
  (getM state))) ≠ val2form (elements (getM state))
getF state @ val2form (elements (getM state)) @ [remdups
  (removeFalseLiterals clause (elements (getM state)))]
  Phi @ [clause]]
by (auto simp add: Let-def)
next
case False
thus ?thesis
using *
using satisfiableAppend[of Phi [clause]]
using (¬ clauseTrue ?clause' (elements (getM state)));
using (length ?clause' ≠ 1);
using (¬ clauseTautology ?clause');
using (¬ clauseTautology clause);
unfolding addClause-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def)
qed
qed
qed
qed
qed


578
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
InvariantUniqQ (getQ state) (getConflictFlag state) ∨ (getQ state) = []
currentLevel (getM state) = 0
finite Vbl
InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
state' = initialize Phi state
set Phi ⊆ set F0

shows
InvariantConsistent (getM state') ∧
InvariantUniq (getM state') ∧
InvariantWatchListsContainOnlyClausesFromF (getWatchList state') (getF state') ∧
InvariantWatchListsUniq (getWatchList state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchesDiffer (getF state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchCharacterization (getF state') (getWatch1 state') (getWatch2 state') (getM state') ∧
InvariantConflictFlagCharacterization (getConflictFlag state') (getF state') (getM state') ∧
InvariantConflictClauseCharacterization (getConflictFlag state') (getConflictClause state') (getF state') (getM state') ∧
InvariantQCharacterization (getConflictFlag state') (getQ state') (getF state') (getM state') ∧
InvariantUniqQ (getQ state') (getConflictFlag state') ∨ (getQ state') = [] ∧
currentLevel (getM state') = 0 (is ?Inv state')

using assms

proof (induct Phi arbitrary: state)
case Nil
thus ?case
  by simp
next
  case (Cons clause Phi')
  let ?state' = addClause clause state
  have ?Inv ?state'
    using Cons
    using InvariantsAfterAddClause[of state F0 Vbl clause]
    using formulaContainsItsClausesVariables[of clause F0]
    by (simp add: Let-def)
  thus ?case
    using Cons(1)[of ?state'] finite Vbl Cons(18) Cons(19) Cons(20) Cons(21) Cons(22)
    by (simp add: Let-def)
qed

lemma InvariantEquivalentZLAfterInitializationStep:
fixes Phi :: Formula
assumes
  (getSATFlag state = UNDEF ∧ InvariantEquivalentZL (getF state)
   (getM state) (filter (λ c. ¬ clauseTautology c) Phi)) ∨
  (getSATFlag state = FALSE ∧ ¬ satisfiable (filter (λ c. ¬ clause- Tautology c) Phi))
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
  (getF state) and
  InvariantWatchListsUniq (getWatchList state) and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state)
  (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state)
  (getM state)
  InvariantConflictFlagCharacterization (getConflictFlag state) (getF state)
  (getM state)
  InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state)
  (getF state) (getM state)
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state)
  (getM state)
  InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel
  (getM state))
  InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel
  (getM state))
  InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
InvariantUniqQ (getQ state)
finite Vbl
InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
(getConflictFlag state) ∨ (getQ state) = []
currentLevel (getM state) = 0
F0 = Phi ⊕ Phi'

shows
let state' = initialize Phi' state in
  (getSATFlag state' = UNDEF ∧ InvariantEquivalentZL (getF state') (getM state') (filter (λ c. ¬ clauseTautology c) F0)) ∨
  (getSATFlag state' = FALSE ∧ ¬ satisfiable (filter (λ c. ¬ clauseTautology c) F0))
using assms

proof (induct Phi' arbitrary: state Phi)
  case Nil
  thus ?case
    unfolding prefixToLevel-def equivalentFormulae-def
    by simp
next
  case (Cons clause Phi'')
  let ?filt = λ F. (filter (λ c. ¬ clauseTautology c) F)
  let ?state' = addClause clause state
  let ?Phi' = ?filt Phi @ [clause]
  let ?Phi'' = if clauseTautology clause then ?filt Phi else ?Phi'
  from Cons
  have getSATFlag ?state' = UNDEF ∧ InvariantEquivalentZL (getF ?state') (getM ?state') (getF ?state') (getM ?state') (?filt ?Phi'') ∨
    getSATFlag ?state' = FALSE ∧ ¬ satisfiable (?filt ?Phi'')
    using formulaContainsItsClausesVariables[of clause F0]
    using InvariantEquivalentZLAfterAddClause[of state ?filt Phi F0 Vbl clause]
  by (simp add: Let-def)
  hence getSATFlag ?state' = UNDEF ∧ InvariantEquivalentZL (getF ?state') (getM ?state') (?filt (Phi @ [clause])) ∨
    getSATFlag ?state' = FALSE ∧ ¬ satisfiable (?filt (Phi @ [clause]))
  by auto
  moreover
  from Cons
  have InvariantConsistent (getM ?state') ∧
    InvariantUniq (getM ?state') ∧
    InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state') (getF ?state') ∧
    InvariantWatchListsUniq (getWatchList ?state') ∧
    InvariantWatchListsCharacterization (getWatchList ?state') (getWatch1 ?state') (getWatch2 ?state') ∧
    InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2
InvariantWatchesDiffer (getF ?state') (getWatch1 ?state') (getWatch2 ?state') ∧
InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getM ?state') ∧
InvariantConflictFlagCharacterization (getConflictFlag ?state') (getF ?state') (getM ?state') ∧
InvariantConflictClauseCharacterization (getConflictClause ?state') (getF ?state') (getM ?state') ∧
InvariantQCharacterization (getConflictFlag ?state') (getQ ?state') (getF ?state') (getM ?state') ∧
InvariantGetReasonIsReason (getReason ?state') (getF ?state') (getM ?state') (set (getQ ?state')) ∧
InvariantUniqQ (getQ ?state') ∧
InvariantVarsM (getM ?state') F0 Vbl ∧
InvariantVarsQ (getQ ?state') F0 Vbl ∧
InvariantVarsF (getF ?state') F0 Vbl ∧
((getConflictFlag ?state') ∨ (getQ ?state') = []) ∧
currentLevel (getM ?state') = 0
using formulaContainsItsClausesVariables[of clause F0]
using InvariantsAfterAddClause by (simp add: Let-def)
moreover
hence InvariantNoDecisionsWhenConflict (getF ?state') (getM ?state') (currentLevel (getM ?state'))
InvariantNoDecisionsWhenUnit (getF ?state') (getM ?state') (currentLevel (getM ?state'))
unfolding InvariantNoDecisionsWhenConflict-def
unfolding InvariantNoDecisionsWhenUnit-def
by auto
ultimately
show ?case
using Cons(1)[of ?state' Phi @ [clause]] :finite Vbl Cons(23)
Cons(24)
by (simp add: Let-def)
qed

lemma InvariantsAfterInitialization:
shows
let state' = (initialize F0 initialState) in
InvariantConsistent (getM state') ∧
InvariantUniq (getM state') ∧
InvariantWatchListsContainOnlyClausesFromF (getWatchList state') (getF state') ∧
InvariantWatchListsUniq (getWatchList state') ∧
InvariantWatchListsCharacterization (getWatchList state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchesDiffer (getF state') (getWatch1 state') (getWatch2 state') ∧
InvariantWatchCharacterization (getF state') (getWatch1 state') (getWatch2 state') (getM state') ∧
InvariantConflictFlagCharacterization (getConflictFlag state') (getWatch1 state') (getWatch2 state') (getM state') ∧
InvariantConflictClauseCharacterization (getConflictClause state') (getF state') (getM state') ∧
InvariantQCharacterization (getConflictFlag state') (getQ state') (getWatch1 state') (getWatch2 state') (getM state') ∧
InvariantConflictClauseCharacterization (getConflictClause state') (getF state') (getQ state') (getM state') ∧
InvariantGetReasonIsReason (getReason state') (getF state') (getQ state') (getM state') (set (getQ state')) ∧
InvariantUniqQ (getQ state') (getM state') \{\} ∧
InvariantVarsQ (getQ state') F0 \{\} ∧
InvariantVarsF (getF state') F0 \{\} ∧
((getConflictFlag state') ∨ (getQ state') = []) ∧
currentLevel (getM state') = 0

using assms
using InvariantsAfterInitializationStep[of initialState \{\} F0 initialize F0 initialState F0]
unfolding initialState-def
unfolding InvariantConsistent-def
unfolding InvariantUniq-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsCharacterization-def
unfolding InvariantWatchesEl-def
unfolding InvariantWatchesDiffer-def
unfolding InvariantWatchCharacterization-def
unfolding watchCharacterizationCondition-def
unfolding InvariantConflictFlagCharacterization-def
unfolding InvariantConflictClauseCharacterization-def
unfolding InvariantQCharacterization-def
unfolding InvariantUniqQ-def
unfolding InvariantNoDecisionsWhenConflict-def
unfolding InvariantNoDecisionsWhenUnit-def
unfolding InvariantGetReasonIsReason-def
unfolding InvariantVarsM-def
unfolding InvariantVarsQ-def
unfolding InvariantVarsF-def
unfolding currentLevel-def
by (simp) (force)

lemma InvariantEquivalentZLAfterInitialization:
fixes $F_0 :: \text{Formula}$

shows

let $\text{state}' = (\text{initialize } F_0 \text{ initialState})$ in
let $F_0' = (\text{filter} \ (\lambda c. \neg \text{clauseTautology c}) \ F_0)$ in
$(\text{getSATFlag state'}) = \text{UNDEF} \land \text{InvariantEquivalentZL} (\text{getF state'}) (\text{getM state'}) \neg F_0'$

using $\text{InvariantEquivalentZL}\text{AfterInitializationStep}[\text{of initialState}] [{} F_0 F_0]$

unfolding $\text{InitialState-def}$
unfolding $\text{InvariantEquivalentZL-def}$
unfolding $\text{InvariantConsistent-def}$
unfolding $\text{InvariantUniq-def}$
unfolding $\text{InvariantWatchesEl-def}$
unfolding $\text{InvariantWatchesDifer-def}$
unfolding $\text{InvariantWatchListsContainOnlyClausesFromF-def}$
unfolding $\text{InvariantWatchListsUniq-def}$
unfolding $\text{InvariantWatchListsCharacterization-def}$
unfolding $\text{InvariantConflictClapeCharacterization-def}$
unfolding $\text{InvariantQCharacterization-def}$
unfolding $\text{InvariantNoDecisionsWhenConflict-def}$
unfolding $\text{InvariantNoDecisionsWhenUnit-def}$
unfolding $\text{InvariantGetReasonIsReason-def}$
unfolding $\text{InvariantVarsM-def}$
unfolding $\text{InvariantVarsQ-def}$
unfolding $\text{InvariantVarsF-def}$
unfolding $\text{watchCharacterizationCondition-def}$
unfolding $\text{InvariantUniqQ-def}$
unfolding $\text{prefixToLevel-def}$
unfolding $\text{equivalentFormulae-def}$
unfolding $\text{currentLevel-def}$
by (auto simp add: Let-def)

end

theory ConflictAnalysis
imports AssertLiteral
begin

lemma clauseFalseInPrefixToLastAssertedLiteral:
assumes
isLastAssertedLiteral $l$ (oppositeLiteralList $c$) (elements $M$) and
clauseFalse $c$ (elements $M$) and
uniq (elements $M$)

584
shows clauseFalse c (elements (prefixToLevel (elementLevel l M) M))
proof –
{ 
fix l’::Literal
assume l’ el c
hence literalFalse l’ (elements M)
  using (clauseFalse c (elements M));
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
  hence literalTrue (opposite l’) (elements M)
    by simp

have opposite l’ el oppositeLiteralList c
  using (d’ el c)
    using literalElListIffOppositeLiteralElOppositeLiteralList[of l’ c]
    by simp

have elementLevel (opposite l’) M ≤ elementLevel l M
  using [lastAssertedLiteralHasHighestElementLevel[of l oppositeLiteralList c M]
    using (isLastAssertedLiteral l (oppositeLiteralList c) (elements M))
      using (uniq (elements M));
    using (opposite l’ el oppositeLiteralList c)
      using (literalTrue (opposite l’) (elements M))
        by auto
      hence opposite l’ el (elements (prefixToLevel (elementLevel l M) M))
        using elementLevelLtLevelImpliesMemberPrefixToLevel[of opposite l’ M elementLevel l M]
          using (literalTrue (opposite l’) (elements M))
            by simp
  } thus ?thesis
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
qed

lemma InvariantNoDecisionsWhenConflictEnsuresCurrentLevelCl:
assumes
InvariantNoDecisionsWhenConflict F M (currentLevel M)
clause el F
clauseFalse clause (elements M)
uniq (elements M)
currentLevel M > 0
shows
clause ≠ [] ∧
(let Cl = getLastAssertedLiteral (oppositeLiteralList clause) (elements M) in
 InvariantClCurrentLevel Cl M)
proof
  
  have clause ≠ []

proof

  { assume ¬ ?thesis
    hence clauseFalse clause (elements (prefixToLevel ((currentLevel M) − 1) M))
      by simp
    hence False
      using ⟨InvariantNoDecisionsWhenConflict F M (currentLevel M)⟩
    using ⟨currentLevel M > 0⟩
    using ⟨clause el F⟩
    unfolding InvariantNoDecisionsWhenConflict-def
    by (simp add: formulaFalseIffContainsFalseClause)
  } thus ?thesis
  by auto
qed

moreover
  let ?Cl = getLastAssertedLiteral (oppositeLiteralList clause) (elements M)

  have elementLevel ?Cl M = currentLevel M

proof

  have elementLevel ?Cl M ≤ currentLevel M
    using ⟨elementLevelLeqCurrentLevel[of ?Cl M]⟩
    by simp

  moreover
  have elementLevel ?Cl M ≥ currentLevel M

proof

  { assume elementLevel ?Cl M < currentLevel M
    have isLastAssertedLiteral ?Cl (oppositeLiteralList clause) (elements M)
      using ⟨getLastAssertedLiteralCharacterization[of clause elements M]⟩
      using ⟨uniq (elements M)⟩
      using ⟨clauseFalse clause (elements M)⟩
      using ⟨clause ≠ []⟩
      by simp
    hence clauseFalse clause (elements (prefixToLevel (elementLevel ?Cl M) M))
      using clauseFalseInPrefixToLastAssertedLiteral[of ?Cl clause M]
      using ⟨clauseFalse clause (elements M)⟩
      by simp
    hence False
      using ⟨clause el F⟩
      using ⟨InvariantNoDecisionsWhenConflict F M (currentLevel M)⟩
  }
using \( \langle \text{currentLevel } M > 0 \rangle \)
unfolding InvariantNoDecisionsWhenConflict-def
using \( \langle \text{elementLevel } \#Cl M < \text{currentLevel } M \rangle \)
by (simp add: formulaFalseIffContainsFalseClause)

} thus \(?\text{thesis}\)
by force
qed
ultimately
show \(?\text{thesis}\)
by simp
qed
ultimately
show \(?\text{thesis}\)
unfolding InvariantClCurrentLevel-def
by (simp add: Let-def)

lemma InvariantsClAfterApplyConflict:
assumes
getConflictFlag state
InvariantUniq (getM state)
InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))
InvariantEquivalentZL (getF state) (getM state) F0
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
currentLevel (getM state) > 0
shows
let state' = applyConflict state in
InvariantCFalse (getConflictFlag state') (getM state') (getC state')
\(\land\)
InvariantCEntailed (getConflictFlag state') F0 (getC state')
\(\land\)
InvariantClCharacterization (getCl state') (getC state') (getM state')
\(\land\)
InvariantClCurrentLevel (getCl state') (getM state') \(\land\)
InvariantCnCharacterization (getCn state') (getC state') (getM state')
\(\land\)
InvariantUniqC (getC state')

proof-
let \(?M0 = \text{elements } \text{prefixToLevel } 0 \text{ (getM state)}\)
let \(?oppM0 = \text{oppositeLiteralList } \?M0\)

let \(?\text{clause' } = \text{nth} \text{ (getF state)} \text{ (getConflictClause state)}\)
let \(?\text{clause'' } = \text{list-diff } \?\text{clause'} \?oppM0\)
let \(?\text{clause } = \text{remdup } \?\text{clause''}\)
let \(?\!l = \text{getLastAssertedLiteral } \text{oppositeLiteralList } \?\text{clause'} \text{ (elements}} \text{ (getM state)})\)
have clauseFalse ?clause' (elements (getM state)) ?clause' el (getF state)
  using 'getConflictFlag state
  using :InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
  unfolding InvariantConflictClauseCharacterization-def
  by (auto simp add: Let-def)

have ?clause' ≠ [] elementLevel ?l (getM state) = currentLevel (getM state)
  using InvariantNoDecisionsWhenConflictEnsuresCurrentLevelCI[of getF state getM state ?clause']
  using 'clause' el (getF state)
  using 'clauseFalse ?clause' (elements (getM state))
  using InvariantNoDecisionsWhenConflict (getF state) (getM state)
  (currentLevel (getM state))
  using (currentLevel (getM state) > 0)
  using (InvariantUniq (getM state))
  unfolding InvariantUniq-def
  unfolding InvariantClCurrentLevel-def
  by (auto simp add: Let-def)

have isLastAssertedLiteral ?l (oppositeLiteralList ?clause') (elements (getM state))
  using 'clause' ≠ []
  using 'clauseFalse ?clause' (elements (getM state))
  using InvariantUniq (getM state)
  unfolding InvariantUniq-def
  using getLastAssertedLiteralCharacterization[of ?clause' elements (getM state)]
  by simp
  hence ?l el (oppositeLiteralList ?clause')
  unfolding isLastAssertedLiteral-def
  by simp
  hence opposite ?l el ?clause'
    by auto

have ¬ ?l el ?M0
proof -
  { assume ¬ ?thesis
    hence elementLevel ?l (getM state) = 0
      using prefixToLevelElementsElementLevel[of ?l 0 getM state]
      by simp
    hence False
  }
using \( \langle \text{elementLevel} \ ?l \ (\text{getM state}) = \text{currentLevel} \ (\text{getM state}) \rangle \)
using \( \langle \text{currentLevel} \ (\text{getM state}) > 0 \rangle \)
by simp
\}
thus \( ?\text{thesis} \)
by auto
qed

hence \( \neg \text{opposite} \ ?l \ ?l \ ?\text{opM0} \)
using \( \text{literalElListIffOppositeLiteralElOppositeLiteralList} \ [\text{of} \ ?l \ \text{elements} \ (\text{prefixToLevel 0} \ (\text{getM state}))] \)
by simp

have \( \text{opposite} \ ?l \ ?l \ ?\text{clause''} \)
using \( \langle \text{opposite} \ ?l \ ?l \ ?\text{clause'} \rangle \)
using \( \langle \neg \text{opposite} \ ?l \ ?l \ ?\text{opM0} \rangle \)
using \( \text{listDiffIff} \ [\text{of opposite} \ ?l \ ?\text{clause'} \ ?\text{opM0}] \)
by simp
hence \( ?l \ ?l \ (\text{oppositeLiteralList} \ ?\text{clause''}) \)
using \( \text{literalElListIffOppositeLiteralElOppositeLiteralList} \ [\text{of opposite} \ ?l \ ?\text{clause'}] \)
by simp

have \( \text{set} \ (\text{oppositeLiteralList} \ ?\text{clause''}) \subseteq \text{set} \ (\text{oppositeLiteralList} \ ?\text{clause'}) \)
proof
fix \( x \)
assume \( x \in \text{set} \ (\text{oppositeLiteralList} \ ?\text{clause''}) \)
thus \( x \in \text{set} \ (\text{oppositeLiteralList} \ ?\text{clause'}) \)
using \( \text{literalElListIffOppositeLiteralElOppositeLiteralList} \ [\text{of opposite} \ x \ ?\text{clause'}] \)
using \( \text{literalElListIffOppositeLiteralElOppositeLiteralList} \ [\text{of opposite} \ x \ ?\text{clause''}] \)
using \( \text{listDiffIff} \ [\text{of opposite} \ x \ ?\text{clause'} \ \text{oppositeLiteralList} \ (\text{elements} \ (\text{prefixToLevel 0} \ (\text{getM state})))] \)
by auto
qed

have \( \text{isLastAssertedLiteral} \ ?l \ (\text{oppositeLiteralList} \ ?\text{clause'}) \ (\text{elements} \ (\text{getM state})) \)
using \( \langle ?l \ ?l \ (\text{oppositeLiteralList} \ ?\text{clause''}) \rangle \)
using \( \text{set} \ (\text{oppositeLiteralList} \ ?\text{clause''}) \subseteq \text{set} \ (\text{oppositeLiteralList} \ ?\text{clause'}) \)
using \( \text{isLastAssertedLiteral} \ ?l \ (\text{oppositeLiteralList} \ ?\text{clause'}) \ (\text{elements} \ (\text{getM state})) \)
using \( \text{isLastAssertedLiteral} \ ?l \ \text{oppositeLiteralList} \ ?\text{clause'} \ ?\text{elements} \ (\text{getM state}) \ \text{oppositeLiteralList} \ ?\text{clause''} \)
by auto
moreover
have set (oppositeLiteralList ?clause) = set (oppositeLiteralList ?clause’)
  unfolding oppositeLiteralList-def
  by simp
ultimately
have isLastAssertedLiteral ?l (oppositeLiteralList ?clause) (elements (getM state))
  unfolding isLastAssertedLiteral-def
  by auto

hence ?l el (oppositeLiteralList ?clause)
  unfolding isLastAssertedLiteral-def
  by simp
hence opposite ?l el ?clause
  by simp
hence ?clause ≠ []
  by auto

have clauseFalse ?clause’’ (elements (getM state))

proof –
{
  fix l::Literal
  assume l el ?clause’’
  hence l el ?clause’
    using listDiffIff[of l ?clause’ ?oppM0]
    by simp
  hence literalFalse l (elements (getM state))
    using (clauseFalse ?clause’ (elements (getM state)));
    by (simp add: clauseFalseIffAllLiteralsAreFalse)
}
thus ?thesis
  by (simp add: clauseFalseIffAllLiteralsAreFalse)
qed

hence clauseFalse ?clause (elements (getM state))
  by (simp add: clauseFalseIffAllLiteralsAreFalse)

let ?l’ = getLastAssertedLiteral (oppositeLiteralList ?clause) (elements (getM state))
have isLastAssertedLiteral ?l’ (oppositeLiteralList ?clause) (elements (getM state))
  using (?clause ≠ [])
  using (clauseFalse ?clause (elements (getM state)));
  using InvariantUniq (getM state);
  unfolding InvariantUniq-def
  using getLastAssertedLiteralCharacterization[of ?clause elements (getM state)]
  by simp
with (isLastAssertedLiteral ?l (oppositeLiteralList ?clause) (elements
(getM state))

have \( ?l = ?l' \)
using lastAssertedLiteralIsUniq
by simp

have formulaEntailsClause (getF state) ?clause' 
using \( \{?clause' \} \in (getF state) \)
by (simp add: formulaEntailsItsClauses)

let \( ?F0 = (getF state) \oplus \text{val2form} \ ?M0 \)

have formulaEntailsClause ?F0 ?clause'
using \( \text{formulaEntailsClause} \ (getF state) \ ?clause' \)
by (simp add: formulaEntailsClauseAppend)

hence formulaEntailsClause ?F0 ?clause''
using \( \text{formulaEntailsClause} \ (getF state) \ ?clause' \)
using formulaEntailsClauseRemoveEntailedLiteralOpposites[of ?F0 
?clause' \ ?M0]
using val2formIsEntailed[of getF state \ ?M0 []]
by simp
hence formulaEntailsClause ?F0 ?clause
unfolding formulaEntailsClause-def
by (simp add: clauseTrueIffContainsTrueLiteral)

hence formulaEntailsClause F0 ?clause
using \( \text{InvariantEquivalentZL} \ (getF state) \ (getM state) \ F0 \)
unfolding InvariantEquivalentZL-def
unfolding formulaEntailsClause-def
unfolding equivalentFormulae-def
by auto

show ?thesis
using \( \text{isLastAssertedLiteral} \ ?l' \ (\text{oppositeLiteralList} \ ?clause) \ \text{(elements} \\
\text{getM state)}) \)
using \( ?l = ?l' \)
using \( \text{elementLevel} \ ?l \ \text{getM state} = \text{currentLevel} \ \text{getM state} \)
using \( \text{clauseFalse} \ ?clause \ \text{(elements} \ \text{getM state}) \)
using \( \text{formulaEntailsClause} \ F0 \ ?clause \)
unfolding applyConflict-def
unfolding setConflictAnalysisClause-def
unfolding InvariantCICharacterization-def
unfolding InvariantCICurrentLevel-def
unfolding InvariantCFalse-def
unfolding InvariantCEntailed-def
unfolding InvariantCnCharacterization-def
unfolding InvariantUniqC-def
by (auto simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def uniqDistinct distinct-remdups-id)
lemma CnEqual1IffUIP:
assumes
InvClCharacterization (getCl state) (getC state) (getM state)
InvClCurrentLevel (getCl state) (getM state)
InvCnCharacterization (getCn state) (getC state) (getM state)
shows
(getCn state = 1) = isUIP (opposite (getCl state)) (getC state) (getM state)
proof—
let ?clls = filter (\ l. elementLevel (opposite l) (getM state) = currentLevel (getM state)) (remdups (getC state))
let ?Cl = getCl state

have isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state)) (elements (getM state))
  using \InvClCharacterization (getCl state) (getC state) (getM state):
    unfolding InvClCharacterization-def
  .
hence literalTrue ?Cl (elements (getM state)) ?Cl el (oppositeLiteralList (getC state))
  unfolding isLastAssertedLiteral-def
  by auto
hence opposite ?Cl el getC state
  using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite ?Cl getC state]
  by simp
hence opposite ?Cl el ?clls
  using (InvClCurrentLevel (getCl state) (getM state))
  unfolding InvClCurrentLevel-def
  by auto
hence ?clls ≠ []
  by force
hence length ?clls > 0
  by simp
have uniq ?clls
  by (simp add: uniqDistinct)
{
  assume getCn state ≠ 1
  hence length ?clls > 1

qed
using assms
using ⟨length ?clls > 0⟩
unfolding InvariantCnCharacterization-def
by (simp (no-asmp))
then obtain literal1::Literal and literal2::Literal
where literal1 el ?clls literal2 el ?clls literal1 ≠ literal2
using ⟨uniq ?clls⟩
using ⟨?clls ≠ []⟩
using lengthGtOneTwoDistinctElements[of ?clls]
by auto
then obtain literal::Literal
where literal el ?clls literal ≠ opposite ?Cl
using ⟨opposite ?Cl el ?clls⟩
by auto
hence ~ isUIP (opposite ?Cl) (getC state) (getM state)
using ⟨opposite ?Cl el ?clls⟩
unfolding isUIP-def
by auto
}
moreover
{
  assume getCn state = 1
  hence length ?clls = 1
  using ⟨InvariantCnCharacterization (getCn state) (getC state) (getM state)⟩
  unfolding InvariantCnCharacterization-def
  by auto
  
  fix literal::Literal
  assume literal el (getC state) literal ≠ opposite ?Cl
  have elementLevel (opposite literal) (getM state) < currentLevel
  (getM state)
  proof−
    have elementLevel (opposite literal) (getM state) ≤ currentLevel
    (getM state)
    using elementLevelLeqCurrentLevel[of opposite literal getM state]
    by simp
  moreover
  have elementLevel (opposite literal) (getM state) ≠ currentLevel
  (getM state)
  proof−
  
  assume ~ ?thesis
  with ⟨literal el (getC state)⟩
  have literal el ?clls
  by simp
  hence False
  using ⟨length ?clls = 1⟩
using (opposite ?Cl el ?clls)
using (literal ≠ opposite ?Cl)
using lengthOneImpliesOnlyElement[of ?clls opposite ?Cl]
  by auto
}
thus ?thesis
  by auto
qed
ultimately
show ?thesis
  by simp
qed

hence isUIP (opposite ?Cl) (getC state) (getM state)
using (isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state))
(elements (getM state)):
  using (opposite ?Cl el ?clls)
  unfolding isUIP-def
  by auto
}
ultimately
show ?thesis
  by auto
qed

lemma InvariantsClAfterApplyExplain:
assumes
  InvariantUniq (getM state)
  InvariantCFalse (getConflictFlag state) (getM state) (getC state)
  InvariantClCharacterization (getCl state) (getC state) (getM state)
  InvariantClCurrentLevel (getCl state) (getM state)
  InvariantCEntailed (getConflictFlag state) F0 (getC state)
  InvariantCnCharacterization (getCn state) (getC state) (getM state)
  InvariantEquivalentZL (getF state) (getM state) F0
  InvariantGetReasonIsReason (getReason state) (getF state) (getM state)
(set (getQ state))
  getOn state ≠ 1
  getConflictFlag state
  currentLevel (getM state) > 0
shows
  let state' = applyExplain (getCl state) state in
  InvariantCFalse (getConflictFlag state') (getM state') (getC state')
  ∧ InvariantCEntailed (getConflictFlag state') F0 (getC state')
  ∧ InvariantClCharacterization (getCl state') (getC state') (getM state')
  ∧ InvariantClCurrentLevel (getCl state') (getM state')
  ∧ InvariantCnCharacterization (getCn state') (getC state') (getM state')

state) ∧
InvariantUniqC (getC state')

proof
  let ?Cl = getCl state
  let ?oppM0 = oppositeLiteralList (elements (prefixToLevel 0 (getM state)))

  have isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state)) (elements (getM state))
    using InvariantClCharacterization (getCl state) (getC state) (getM state);
    unfolding InvariantClCharacterization-def
  .
  hence literalTrue ?Cl (elements (getM state)) ?Cl el (oppositeLiteralList (getC state))
    unfolding isLastAssertedLiteral-def
    by auto
  hence opposite ?Cl el getC state
    using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite ?Cl getC state]
    by simp

  have clauseFalse (getC state) (elements (getM state))
    using (getConflictFlag state);
    using InvariantCFalse (getConflictFlag state) (getM state) (getC state)
    unfolding InvariantCFalse-def
    by simp

  have ¬ isUIP (opposite ?Cl) (getM state)
    using CnEqualIffUIP[of state]
    using assms
    by simp

  have ¬ ?Cl el (decisions (getM state))
  proof
    { assume ¬ ?thesis
      hence isUIP (opposite ?Cl) (getM state)
        using InvariantUniq (getM state);
        using isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state)) (elements (getM state));
        using clauseFalse (getC state) (elements (getM state));
        using lastDecisionThenUIP[of getM state opposite ?Cl getC state]
        unfolding InvariantUniq-def
        by simp
  }
with (¬ isUIP (opposite ?Cl) (getC state) (getM state))
have False
by simp
} thus ?thesis
by auto
qed

have elementLevel ?Cl (getM state) = currentLevel (getM state)
using ⟨InvariantClCurrentLevel (getCl state) (getM state)⟩
unfolding InvariantClCurrentLevel-def
by simp
hence elementLevel ?Cl (getM state) > 0
using ⟨currentLevel (getM state) > 0⟩
by simp

obtain reason
where isReason (nth (getF state) reason) ?Cl (elements (getM state))
getReason state ?Cl = Some reason 0 ≤ reason ∧ reason < length (getF state)
using ⟨InvariantGetReasonIsReason (getReason state) (getF state)
(getM state) (set (getQ state))⟩
unfolding InvariantGetReasonIsReason-def
using ⟨literalTrue ?Cl (elements (getM state))⟩
using (¬ ?Cl el (decisions (getM state))):
using ⟨elementLevel ?Cl (getM state) > 0⟩
by auto

let ?res = resolve (getC state) (getF state ! reason) (opposite ?Cl)

obtain ol::Literal
where ol el (getC state)
  ol ≠ opposite ?Cl
elementLevel (opposite ol) (getM state) ≥ elementLevel ?Cl (getM state)
using ⟨isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state))
(elements (getM state))⟩
using (¬ isUIP (opposite ?Cl) (getC state) (getM state))
unfolding isUIP-def
by auto
hence ol el ?res
unfolding resolve-def
by simp
hence ?res ≠ []
by auto
have opposite ol el (oppositeLiteralList ?res)
using ⟨ol el ?res⟩
using literalElListJetOppositeLiteralElOppositeLiteralList[of ol ?res]
by simp
have opposite ol el (oppositeLiteralList (getC state))
using ⟨ol el (getC state)⟩
using literalElListIffOppositeLiteralElOppositeLiteralList[of ol getC state]
  by simp

have literalFalse ol (elements (getM state))
using ⟨clauseFalse (getC state) (elements (getM state))⟩
using ⟨ol el getC state⟩
by (simp add: clauseFalseIffAllLiteralsAreFalse)

have elementLevel (opposite ol) (getM state) = elementLevel ?Cl (getM state)
using ⟨elementLevel (opposite ol) (getM state) ≥ elementLevel ?Cl (getM state)⟩
using ⟨isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state)) (elements (getM state))⟩
using ⟨lastAssertedLiteralHasHighestElementLevel[of ?Cl oppositeLiteralList (getC state) getM state⟩
using ⟨InvariantUniq (getM state)⟩
unfolding InvariantUniq-def
using ⟨opposite ol el (oppositeLiteralList (getC state))⟩
using ⟨literalFalse ol (elements (getM state))⟩
by auto

hence elementLevel (opposite ol) (getM state) = currentLevel (getM state)
using ⟨elementLevel ?Cl (getM state) = currentLevel (getM state)⟩
by simp

have InvariantCFalse (getConflictFlag state) (getM state) ?res
using ⟨InvariantCFalse (getConflictFlag state) (getM state) (getC state)⟩
using ⟨InvariantCFalseAfterExplain[of getConflictFlag state getM state getC state ?Cl nth (getF state) reason ?res⟩
using ⟨isReason (nth (getF state) reason) ?Cl (elements (getM state))⟩
using ⟨opposite ?Cl el (getC state)⟩
by simp

hence clauseFalse ?res (elements (getM state))
using ⟨getConflictFlag state⟩
unfolding InvariantCFalse-def
by simp

let ?rc = nth (getF state) reason
let ?M0 = elements (prefixToLevel 0 (getM state))
let ?F0 = (getF state) @ (val2form ?M0)
let ?C' = list-diff ?res ?oppM0
let ?C = remdups ?C'
have \( \text{formulaEntailsClause} \) \((\text{getF state}) \) ?rc
  using \(\{0 \leq \text{reason} \land \text{reason} < \text{length} \) \((\text{getF state})\}\)
  using nth-mem\([\text{reason} \text{getF state}]\)
  by (simp add: \text{formulaEntailsItsClauses})

hence \( \text{formulaEntailsClause} \) \(?F0 \) ?rc
  by (simp add: \text{formulaEntailsClauseAppend})

hence \( \text{formulaEntailsClause} \) \(?F0 \) ?rc
  using \(\text{InvariantCEntailed} \) \((\text{getConflictFlag state}) \) \(?F0 \) \((\text{getC state})\)
  unfolding \text{InvariantCEntailed-def}
  unfolding \text{formulaEntailsClause-def}
  unfolding \text{equivalentFormulae-def}
  by simp

hence \( \text{formulaEntailsClause} \) \(?F0 \) ?res
  using \(\text{InvariantCEnrolled} \)
  unfolding \text{InvariantCEnrolled-def}
  by auto

hence \( \text{formulaEntailsClause} \) \(?F0 \) ?res
  using \(\text{formulaEntailsClauseRemoveEntailedLiteralOpposites} \) \(?F0 \) ?res \(?\text{M0}\)
  using \(\text{val2formIsEntailed} \) \(?\text{getF state} \) ?M0 \([]\)
  unfolding \text{formulaEntailsClause-def}
  by (auto simp add: \text{clauseTrueIffContainsTrueLiteral})

hence \( \text{formulaEntailsClause} \) \(?F0 \) ?C
  using \(\text{formulaEntailsClauseRemoveEntailedLiteralOpposites} \) \(?F0 \) \(?\text{M0}\)
  unfolding \text{formulaEntailsClause-def}
  by simp

let \(?ll = \text{getLastAssertedLiteral} \) \((\text{oppositeLiteralList} \) \(?\text{res}\) \) \((\text{elements} \) \((\text{getM state})\))

have \(\text{isLastAssertedLiteral} \) \(?ll \) \((\text{oppositeLiteralList} \) \(?\text{res}\) \) \((\text{elements} \) \((\text{getM state})\))
  using \(\{?\text{res} \neq \[]\}\)
  using \(\text{clauseFalse} \) \(?\text{res} \) \((\text{elements} \) \((\text{getM state})\)):
using ⟨InvariantUniq (getM state)⟩

unfolding InvariantUniq-def

using getLastAssertedLiteralCharacterization[of ?res elements (getM state)]

by simp

hence elementLevel (opposite ol) (getM state) ≤ elementLevel ?ll (getM state)

using ⟨opposite ol el (oppositeLiteralList (getC state))⟩

using lastAssertedLiteralHasHighestElementLevel[of ?ll oppositeLiteralList ?res getM state]

using ⟨InvariantUniq (getM state)⟩

using ⟨opposite ol el (oppositeLiteralList ?res)⟩

using ⟨literalFalse ol (elements (getM state))⟩

unfolding InvariantUniq-def

by simp

hence elementLevel ?ll (getM state) = currentLevel (getM state)

using (elementLevel (opposite ol) (getM state) = currentLevel (getM state))

using elementLevelLeqCurrentLevel[of ?ll getM state]

by simp

have ?ll el (oppositeLiteralList ?res)

using ⟨isLastAssertedLiteral ?ll (oppositeLiteralList ?res) (elements (getM state))⟩

unfolding isLastAssertedLiteral-def

by simp

hence opposite ?ll el ?res

using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite ?ll ?res]

by simp

have ¬ ?ll el (elements (prefixToLevel 0 (getM state)))

proof
{

  assume ¬ ?thesis

  hence elementLevel ?ll (getM state) = 0

  using prefixToLevelElementsElementLevel[of ?ll 0 getM state]

  by simp

  hence False

  using (elementLevel ?ll (getM state) = currentLevel (getM state))

  using (currentLevel (getM state) > 0)

  by simp

  }

  thus ?thesis

  by auto

qed

hence ¬ opposite ?ll el ?oppM0
using literalElListIffOppositeLiteralElOppositeLiteralList[of ?ll elements (prefixToLevel 0 (getM state))]
  by simp

have opposite ?ll el ?C'
  using ⟨opposite ?ll el ?res⟩
  using ⟨¬ opposite ?ll el ?oppM0⟩
  using listDiffIff[of opposite ?ll ?res ?oppM0]
  by simp
hence ?ll el (oppositeLiteralList ?C')
  using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite ?ll ?C']
  by simp

have set (oppositeLiteralList ?C') ⊆ set (oppositeLiteralList ?res)
proof
  fix x
  assume x ∈ set (oppositeLiteralList ?C')
  thus x ∈ set (oppositeLiteralList ?res)
    using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite x ?C']
    using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite x ?res]
    using listDiffIff[of opposite x ?res ?oppM0]
    by auto
qed

have isLastAssertedLiteral ?ll (oppositeLiteralList ?C') (elements (getM state))
  using ⟨?ll el (oppositeLiteralList ?C')⟩
  using ⟨set (oppositeLiteralList ?C') ⊆ set (oppositeLiteralList ?res)⟩
  using isLastAssertedLiteral ?ll (oppositeLiteralList ?res) (elements (getM state))
  using isLastAssertedLiteralSubset[of ?ll oppositeLiteralList ?res elements (getM state) oppositeLiteralList ?C']
  by auto
moreover
have set (oppositeLiteralList ?C) = set (oppositeLiteralList ?C')
  unfolding oppositeLiteralList-def
  by simp
ultimately
have isLastAssertedLiteral ?ll (oppositeLiteralList ?C) (elements (getM state))
  unfolding isLastAssertedLiteral-def
  by auto

hence ?ll el (oppositeLiteralList ?C)
  unfolding isLastAssertedLiteral-def
by simp
hence opposite ?ll el ?C
  by simp
hence ?C ≠ []
  by auto

have clauseFalse ?C' (elements (getM state))
proof –
  { fix l::Literal
    assume l el ?C'
    hence l el ?res
      using listDiffIff[of l ?res ?oppM0]
      by simp
    hence literalFalse l (elements (getM state))
      using clauseFalse ?res (elements (getM state));
      by (simp add: clauseFalseIffAllLiteralsAreFalse)
  }
thus ?thesis
  by (simp add: clauseFalseIffAllLiteralsAreFalse)
qed

hence clauseFalse ?C (elements (getM state))
  by (simp add: clauseFalseIffAllLiteralsAreFalse)

let ?l' = getLastAssertedLiteral (oppositeLiteralList ?C) (elements (getM state))
  have isLastAssertedLiteral ?l' (oppositeLiteralList ?C) (elements (getM state))
    using (?C ≠ []);
    using ⟨clauseFalse ?C (elements (getM state))⟩
    using ⟨InvariantUniq (getM state)⟩
    unfolding InvariantUniq-def
    using getLastAssertedLiteralCharacterization[of ?C elements (getM state)]
    by simp
  with ⟨isLastAssertedLiteral ?ll (oppositeLiteralList ?C) (elements (getM state))⟩
  have ?ll = ?l'
    using lastAssertedLiteralIsUniq
    by simp

  show ?thesis
    using ⟨isLastAssertedLiteral ?l' (oppositeLiteralList ?C) (elements (getM state))⟩
    using ⟨?ll = ?l'⟩
    using ⟨elementLevel ?ll (getM state) = currentLevel (getM state)⟩
    using ⟨getReason state ?Cl = Some reason⟩
using \langle \text{clauseFalse} \ ?C (\text{elements} (\text{getM} \ state)) \rangle
\text{using} \langle \text{formulaEntailsClause} \ F0 \ ?C \rangle
\text{unfolding applyExplain-def}
\text{unfolding InvariantCFalse-def}
\text{unfolding InvariantCEntailed-def}
\text{unfolding InvariantClCharacterization-def}
\text{unfolding InvariantClCurrentLevel-def}
\text{unfolding InvariantCnCharacterization-def}
\text{unfolding InvariantUniqC-def}
\text{unfolding setConflictAnalysisClause-def}
\text{by (simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def uniqDistinct distinct-remdups-id)}
\text{qed}

definition multLessState = \{(state1, state2). (\text{getM} \ state1 = \text{getM} \ state2) \land (\text{getC} \ state1, \text{getC} \ state2) \in \text{multLess} (\text{getM} \ state1)\}

lemma ApplyExplainUIPTermination:
\text{assumes}
InvariantUniq (\text{getM} \ state)
InvariantGetReasonIsReason (\text{getReason} \ state) (\text{getF} \ state) (\text{getM} \ state)
(set (\text{getQ} \ state))
InvariantCFalse (\text{getConflictFlag} \ state) (\text{getM} \ state) (\text{getC} \ state)
InvariantClCurrentLevel (\text{getCl} \ state) (\text{getM} \ state)
InvariantClCharacterization (\text{getCl} \ state) (\text{getC} \ state) (\text{getM} \ state)
InvariantCnCharacterization (\text{getCn} \ state) (\text{getC} \ state) (\text{getM} \ state)
InvariantCEntailed (\text{getConflictFlag} \ state) F0 (\text{getC} \ state)
InvariantEquivalentZL (\text{getF} \ state) (\text{getM} \ state) F0
\text{getConflictFlag} \ state
\text{currentLevel} (\text{getM} \ state) > 0
\text{shows}
applyExplainUIP-dom \ state
\text{using} \text{assms}
\text{proof (induct rule: wf-induct[of multLessState])}
\text{case 1}
\text{thus ?case}
\text{unfolding wf-eq-minimal}
\text{proof --}
\text{show } \forall Q (\text{state::State}). state \in Q \to (\exists \text{stateMin} \in Q. \forall \text{state}'. (state', \text{stateMin}) \in \text{multLessState} \to state' \notin Q)
\text{proof --}
\{ 
\text{fix Q :: State set and state :: State}
\text{assume state \in Q}
\}
let \(?M = (getM state)\)
let \(?Q1 = \{ C::Clause. \exists state. state \in Q \land (getM state) = \}
\(?M \land (getC state) = C\}\)
from \(\langle state \in Q \rangle\)
have \(getC state \in ?Q1\)
  by auto
with \(wfMultLess[of \ ?M]\)
obtain \(Cmin\) where \(\langle stateMin \in Q \mid \langle getM stateMin = \?M \rangle \rangle\)
proof
  unfolding \(wf-eq-minimal\)
  apply (erule-tac \(x=\?Q1\) in allE)
  apply (erule-tac \(x=\text{getC state}\) in allE)
  by auto
from \(\langle Cmin \in \?Q1 \rangle\) obtain \(stateMin\)
  where \(stateMin \in Q \langle getM stateMin = \?M \rangle\)
proof
  fix \(state'\)
  show \(\langle state', stateMin\rangle \in \text{multLessState} \rightarrow state' \notin Q\)
  proof
    assume \(\langle state', stateMin\rangle \in \text{multLessState}\)
    with \(\langle getM stateMin = \?M \rangle\)
    have \(\langle getM state' = getM stateMin \rangle\) (\(\langle getC state', getC stateMin\rangle \in \text{multLess ?M} \langle getC stateMin \rangle\))
      by auto
    show \(state' \notin Q\)
      by auto
  qed
  qed
  with \(\langle stateMin \in Q \rangle\)
  have \(\exists stateMin \in Q. (\forall state'. (state', stateMin) \in \text{multLessState} \rightarrow state' \notin Q)\)
    by auto
  \}
thus \(\langle stateMin \in Q \rangle\)
  by auto
qed
qed
next
case (2 state')
note ih = this
proof (cases getCn state' = 1)
  case True
  show \( \text{thesis} \)
  apply (rule applyExplainUIP-dom.intros)
  using True
  by simp
next
  case False
  let ?state'' = applyExplain (getCl state') state'
  have InvariantGetReasonIsReason (getReason ?state'') (getF ?state'')
      (getM ?state'') (set (getQ ?state''))
      InvariantUniq (getM ?state'')
      InvariantEquivalentZL (getF ?state'') (getM ?state'') F0
      getConflictFlag ?state''
      currentLevel (getM ?state'') > 0
      using ih
  unfolding applyExplain-def
  unfolding setConflictAnalysisClause-def
  by (auto simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def)
moreover
  have InvariantCFalse (getConflictFlag ?state'') (getM ?state'')
      (getC ?state'')
      InvariantClCharacterization (getCl ?state'') (getC ?state'') (getM ?state'')
      InvariantCnCharacterization (getCn ?state'') (getC ?state'') (getM ?state'')
      InvariantClCurrentLevel (getCl ?state'') (getM ?state'')
      InvariantCEnailed (getConflictFlag ?state'') F0 (getC ?state'')
      using InvariantsClAfterApplyExplain[of state' F0]
      using ih
      using False
      by (auto simp add:Let-def)
moreover
  have (?state'', state') \in\multLessState
proof
  have getM ?state'' = getM state'
  unfolding applyExplain-def
  unfolding setConflictAnalysisClause-def
  by (auto simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def)
  let ?Cl = getCl state'
  let ?oppM0 = oppositeLiteralList (elements (prefixToLevel 0 (getM state'))
have isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state')) (elements (getM state'))
  using ih
  unfolding InvariantClCharacterization-def
  by simp
hence literalTrue ?Cl (elements (getM state')) ?Cl el (oppositeLiteralList (getC state'))
  unfolding isLastAssertedLiteral-def
  by auto
hence opposite ?Cl el getC state'
  using literalElListIffOppositeLiteralElOppositeLiteralList[of opposite ?Cl getC state']
  by simp

have clauseFalse (getC state') (elements (getM state'))
  using ih
  unfolding InvariantCFalse-def
  by simp

have ¬ ?Cl el (decisions (getM state'))
proof{
  assume ¬ ?thesis
  hence isUIP (opposite ?Cl) (getC state') (getM state')
    using ih
    using (isLastAssertedLiteral ?Cl (oppositeLiteralList (getC state')) (elements (getM state')))
    unfolding clauseFalse (getC state') (elements (getM state'))
    using lastDecisionThenUIP[of getM state' opposite ?Cl getC state']
    unfolding InvariantUniq-def
    unfolding isUIP-def
    by simp
  with 'getCn state' ≠ 1'
  have False
    using CnEqual1IffUIP[of state']
    using ih
    by simp
} thus ?thesis
by auto
qed

have elementLevel ?Cl (getM state') = currentLevel (getM state')
  using ih
  unfolding InvariantClCurrentLevel-def
  by simp
hence elementLevel ?Cl (getM state') > 0
  using ih
  by simp
obtain reason

where isReason (nth (getF state') reason) ?Cl (elements (getM state'))

gGetReason state' ?Cl = Some reason 0 ≤ reason ∧ reason < length (getF state')

using ih

unfolding InvariantGetReasonIsReason-def
using ⟨literalTrue ?Cl (elements (getM state'))⟩;
using ⟨¬ ?Cl el (decisions (getM state'))⟩;
using ⟨elementLevel ?Cl (getM state') > 0⟩;
by auto

let ?res = resolve (getC state') (getF state' ! reason) (opposite ?Cl)

have getC state" = (remdups (list-diff ?res ?oppM0))

unfolding applyExplain-def
unfolding setConflictAnalysisClause-def
using ⟨getReason state' ?Cl = Some reason⟩;
by (simp add: Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def)

have (?res, getC state') ∈ multLess (getM state')

using multLessResolve[of ?Cl getC state' nth (getF state') reason getM state']

using ⟨opposite ?Cl el (getC state')⟩;
using ⟨isReason (nth (getF state') reason) ?Cl (elements (getM state'))⟩;
by simp
hence (list-diff ?res ?oppM0, getC state') ∈ multLess (getM state')

by (simp add: multLessListDiff)

have (remdups (list-diff ?res ?oppM0), getC state') ∈ multLess (getM state')

using ⟨(list-diff ?res ?oppM0, getC state') ∈ multLess (getM state')⟩;
by (simp add: multLessRemdups)
thus ?thesis

using ⟨getC ?state" = (remdups (list-diff ?res ?oppM0))⟩;
using ⟨getM ?state" = getM state'⟩;
unfolding multLessState-def
by simp
qed
ultimately
have applyExplainUIP-dom ?state";

using ih
by auto
thus ?thesis
using applyExplainUIP-dom.intros[of state]
using False
by simp
qed
qed

lemma ApplyExplainUIPPreservedVariables:
assumes
applyExplainUIP-dom state
shows
let state' = applyExplainUIP state in
\( (\text{getM} \text{ state}' = \text{getM} \text{ state}) \land \\
(\text{getF} \text{ state}' = \text{getF} \text{ state}) \land \\
(\text{getQ} \text{ state}' = \text{getQ} \text{ state}) \land \\
(\text{getWatch1} \text{ state}' = \text{getWatch1} \text{ state}) \land \\
(\text{getWatch2} \text{ state}' = \text{getWatch2} \text{ state}) \land \\
(\text{getWatchList} \text{ state}' = \text{getWatchList} \text{ state}) \land \\
(\text{getConflictFlag} \text{ state}' = \text{getConflictFlag} \text{ state}) \land \\
(\text{getConflictClause} \text{ state}' = \text{getConflictClause} \text{ state}) \land \\
(\text{getSATFlag} \text{ state}' = \text{getSATFlag} \text{ state}) \land \\
(\text{getReason} \text{ state}' = \text{getReason} \text{ state}) \)
(is let state' = applyExplainUIP state in ?p state state')
using assms
proof (induct state rule: applyExplainUIP-dom.induct)
case (step state')
  note ih = this
  show ?case
  proof (cases getCn state' = 1)
    case True
    with applyExplainUIP.simps[of state']
    have applyExplainUIP state' = state'
      by simp
    thus ?thesis
      by (auto simp only: Let-def)
  next
    case False
    let ?state' = applyExplainUIP (applyExplain (getCl state') state')
    from applyExplainUIP.simps[of state'] False
    have applyExplainUIP state' = ?state'
      by (simp add: Let-def)
    have ?p state' (applyExplain (getCl state') state')
      unfolding applyExplain-def
      unfolding setConflictAnalysisClause-def
      by (auto split: option.split simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def)
    thus ?thesis
      using ih
      using False
lemma isUIPApplyExplainUIP:
assumes applyExplainUIP-dom state
InvariantUniq (getM state)
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
InvariantCEntailed (getConflictFlag state) F0 (getC state)
InvariantCICharacterization (getCl state) (getC state) (getM state)
InvariantCnCharacterization (getCn state) (getC state) (getM state)
InvariantCICurrentLevel (getCl state) (getM state)
InvariantCGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
InvariantEquivalentZL (getF state) (getM state) F0
getConflictFlag state
currentLevel (getM state) > 0
shows let state' = (applyExplainUIP state) in
isUIP (opposite (getCl state')) (getC state') (getM state')
using assms
proof (induct state rule: applyExplainUIP-dom.induct)
  case (step state')
  note ih = this
  show ?case
  proof (cases getCn state' = 1)
    case True
    with applyExplainUIP.simps[of state']
    have applyExplainUIP state' = state'
      by simp
    thus ?thesis
      using ih
      using CnEqual1IfUIP[of state']
    using True
      by (simp add: Let-def)
  next
    case False
    let ?state'' = applyExplain (getCl state') state'
    let ?state' = applyExplainUIP ?state''
    from applyExplainUIP.simps[of state'] False
    have applyExplainUIP state' = ?state'
      by (simp add: Let-def)
  moreover
  have InvariantUniq (getM ?state'')
    InvariantCGetReasonIsReason (getReason ?state'') (getF ?state'') (getM ?state'') (set (getQ ?state''))
    InvariantEquivalentZL (getF ?state'') (getM ?state'') F0
    getConflictFlag ?state''
currentLevel (getM ?state'') > 0
using ih
unfolding applyExplain-def
unfolding setConflictAnalysisClause-def
by (auto split: option.split simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def)

moreover
have InvariantCFalse (getConflictFlag ?state") (getM ?state")
  (getC ?state")
InvariantCEntailed (getConflictFlag ?state") F0 (getC ?state")
InvariantCCharacterization (getCl ?state") (getC ?state") (getM ?state")
InvariantCnCharacterization (getCn ?state") (getC ?state") (getM ?state")
InvariantClCurrentLevel (getCl ?state") (getM ?state")
using False
using ih
using InvariantsClAfterApplyExplain[of state' F0]
by (auto simp add: Let-def)

ultimately
show ?thesis
using ih(2)
using False
by (simp add: Let-def)

qed

qed

lemma InvariantsClAfterExplainUIP:
assumes
applyExplainUIP-dom state
InvariantUniq (getM state)
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
InvariantCEntailed (getConflictFlag state) F0 (getC state)
InvariantCCharacterization (getCl state) (getC state) (getM state)
InvariantCnCharacterization (getCn state) (getC state) (getM state)
InvariantClCurrentLevel (getCl state) (getM state)
InvariantUniqC (getC state)
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
InvariantEquivalentZL (getF state) (getM state) F0
getConflictFlag state
currentState (getM state) > 0
shows
let state' = applyExplainUIP state in
  InvariantCFalse (getConflictFlag state') (getM state') (getC state')
∧
  InvariantCEntailed (getConflictFlag state') F0 (getC state') ∧
  InvariantCCharacterization (getCl state') (getC state') (getM state') ∧
  InvariantCnCharacterization (getCn state') (getC state') (getM state') ∧

609
InvariantCnCharacterization (getCn state') (getC state') (getM state')
  \land
InvariantClCurrentLevel (getCl state') (getM state')
\land
InvariantUniqC (getC state')
using assms

proof (induct state rule: applyExplainUIP-dom.induct)
  case (step state')
  note ih = this
  show ?case
  proof (cases getCn state' = 1)
  case True
  with applyExplainUIP.simps[of state']
  have applyExplainUIP state' = state'
    by simp
  thus ?thesis
  using assms
  using ih
  by (auto simp only: Let-def)
  next
  case False
  let ?state'' = applyExplain (getCl state') state'
  let ?state' = applyExplainUIP ?state''
  from applyExplainUIP.simps[of state'] False
  have applyExplainUIP state' = ?state'
    by (simp add: Let-def)
  moreover
  have InvariantUniq (getM ?state'')
    InvariantGetReasonIsReason (getReason ?state'') (getF ?state'') (getM ?state'')
    (set (getQ ?state''))
    InvariantEquivalentZL (getF ?state'') (getM ?state'') F0
    getConflictFlag ?state''
    currentLevel (getM ?state'') > 0
    using ih
    unfolding applyExplain-def
    unfolding setConflictAnalysisClause-def
    by (auto split: option.split simp add: findLastAssertedLiteral-def countCurrentLevelLiterals-def Let-def)
  moreover
  have InvariantCFalse (getConflictFlag ?state'') (getM ?state'')
    (getC ?state'')
    InvariantCEntailed (getConflictFlag ?state'') F0 (getC ?state'')
    InvariantClCharacterization (getCl ?state'') (getM ?state'')
    (getM ?state'')
    InvariantCnCharacterization (getCn ?state'') (getC ?state'') (getM ?state'')
    InvariantClCurrentLevel (getCl ?state'') (getM ?state'')
    InvariantUniqC (getC ?state'')
  using False
  using ih

610
using Invariants CIAfterApplyExplain [of state' F0]
by (auto simp add: Let-def)
ultimately
show ?thesis
  using False
  using ih(2)
  by simp
qed
qed

lemma oneElementSetCharacterization:
shows (set l = {a}) = ((remdups l) = [a])
proof (induct l)
case Nil
  thus ?case
  by simp
next
case (Cons a' l')
  show ?case
  proof (cases l' = [])
    case True
    thus ?thesis
    by simp
  next
  case False
  then obtain b
  where b ∈ set l'
  by force
  show ?thesis
  proof
    assume set (a' # l') = {a}
    hence a' = a set l' ⊆ {a}
    by auto
    hence b = a
    using (b ∈ set l')
    by auto
    hence {a} ⊆ set l'
    using (b ∈ set l')
    by auto
    hence set l' = {a}
    using (set l' ⊆ {a})
    by auto
    thus remdups (a' # l') = [a]
    using (a' = a)
using Cons
by simp
next
assume remdups (a’ # l’) = [a]
thus set (a’ # l’) = {a}
using set-remdups[of a’ # l’]
by auto
qed
qed
qed

lemma uniqOneElementCharacterization:
assumes
uniq l
shows
(l = [a]) = (set l = {a})
using assms
using uniqDistinct[of l]
using oneElementSetCharacterization[of l a]
using distinct-remdups-id[of l]
by auto

lemma isMinimalBackjumpLevelGetBackjumpLevel:
assumes
InvariantUniq (getM state)
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
InvariantClCharacterization (getCl state) (getC state) (getM state)
InvariantCluCharacterization (getCl state) (getC state) (getM state)
(getM state)
InvariantClCurrentLevel (getCl state) (getM state)
InvariantUniqC (getC state)
getConflictFlag state
isUIP (opposite (getCl state)) (getC state) (getM state)
currentLevel (getM state) > 0
shows
isMinimalBackjumpLevel (getBackjumpLevel state) (opposite (getCl state)) (getC state) (getM state)
proof –
let ?oppC = oppositeLiteralList (getC state)
let ?Cl = getCl state
have isLastAssertedLiteral ?Cl ?oppC (elements (getM state))
using ⟨InvariantClCharacterization (getCl state) (getC state) (getM state)⟩
unfolding InvariantClCharacterization-def
by simp
have elementLevel ?Cl (getM state) > 0
using \( \text{InvariantClCurrentLevel} (\text{getCl state}) (\text{getM state}) \)
using \( \text{currentLevel} (\text{getM state}) > 0 \)
unfolding \( \text{InvariantClCurrentLevel-def} \)
by simp

have clauseFalse \( (\text{getC state}) (\text{elements} (\text{getM state})) \)
using \( \text{getConflictFlag state} \)
using \( \text{InvariantCFalse} (\text{getConflictFlag state}) (\text{getM state}) \) \( (\text{getC state}) \)
unfolding \( \text{InvariantCFalse-def} \)
by simp

show \(?\text{thesis}\)
proof (cases \(\text{getC state} = [\text{opposite } ?\text{Cl}]\))
case True
thus \(?\text{thesis}\)
using \( \text{backjumpLevelZero} [\text{of opposite } ?\text{Cl} \text{ oppositeLiteralList } ?\text{oppC getM state}] \)
using \( \text{isLastAssertedLiteral } ?\text{Cl} \text{ oppC} (\text{elements} (\text{getM state})) \)
using \( \text{True} \)
using \( \text{(elementLevel } ?\text{Cl} \text{ (getM state}) > 0 \)\)
unfolding \( \text{getBackjumpLevel-def} \)
unfolding \( \text{isMinimalBackjumpLevel-def} \)
by (simp add: \text{Let-def})
next
let \( ?\text{Cll} = \text{getCll state} \)
base False
with \( \text{InvariantCllCharacterization} \) \( \text{(getCl state}) \text{ (getCll state)}(\text{getC state}) \text{ (getM state)} \)
\( \text{(InvariantUniqC} \text{ (getC state)} \)
have \( \text{isLastAssertedLiteral } ?\text{Cl} \text{ (removeAll } ?\text{Cl} \text{ oppC} \) \( \text{(elements} \text{(getM state)}) \)
unfolding \( \text{InvariantCllCharacterization-def} \)
unfolding \( \text{InvariantUniqC-def} \)
using \( \text{uniqOneElementCharacterization} [\text{of getC state opposite} \text{Cl}] \)
by simp
hence \( ?\text{Cll el oppC} \text{Cl} \neq ?\text{Cl} \)
unfolding \( \text{isLastAssertedLiteral-def} \)
by auto
hence \( \text{opposite } ?\text{Cll el (getC state)} \)
using \( \text{literalElListIffOppositeLiteralElOppositeLiteralList} [\text{of } ?\text{Cl} \text{ oppC}] \)
by auto

show \(?\text{thesis}\)
using \( \text{backjumpLevelLastLast} [\text{of opposite } ?\text{Cl} \text{ getM state opposite } ?\text{Cll}] \)
using \( \text{(isUIP } \text{(opposite } \text{(getCl state)}) \text{ (getC state}) \text{ (getM state)}) \)
using ⟨clauseFalse (getC state) (elements (getM state))⟩
using ⟨isLastAssertedLiteral ?Cll (removeAll ?Cl ?oppC) (elements (getM state))⟩
using ⟨InvariantUniq (getM state)⟩
using ⟨InvariantUniqC (getC state)⟩
using uniqOneElementCharacterization[of getC state opposite ?Cl]
unfoldingInvariantUniqC-def
unfoldingInvariantUniq-def
using False
using ⟨opposite ?Cll el (getC state)⟩
unfolding getBackjumpLevel-def
unfolding isMinimalBackjumpLevel-def
by (auto simp add: Let-def)
qed
qed

lemma applyLearnPreservedVariables:
let state' = applyLearn state in
getM state' = getM state ∧
getQ state' = getQ state ∧
getC state' = getC state ∧
getCl state' = getCl state ∧
getConflictFlag state' = getConflictFlag state ∧
getConflictClause state' = getConflictClause state ∧
getF state' = (if getC state = [opposite (getCl state)] then
  getF state
else
  (getF state @ [getC state]))
proof (cases getC state = [opposite (getCl state)])
case True
  thus ?thesis
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (simp add: Let-def)
next
case False
  thus ?thesis
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (simp add: Let-def)
variant WatchInvariantsAfterApplyLearn:

assumes

InvariantUniq (getM state) and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and

InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state) and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
InvariantClCharacterization (getCl state) (getC state) (getM state)

and

getConflictFlag state
InvariantCllFalse (getConflictFlag state) (getM state) (getC state)
InvariantUniqC (getC state)

shows

let state’ = (applyLearn state) in

InvariantWatchesEl (getF state’) (getWatch1 state’) (getWatch2 state’) \land
InvariantWatchesDiffer (getF state’) (getWatch1 state’) (getWatch2 state’) \land
InvariantWatchCharacterization (getF state’) (getWatch1 state’) (getWatch2 state’) (getM state’) \land
InvariantWatchListsContainOnlyClausesFromF (getWatchList state’) (getF state’) \land
InvariantWatchListsUniq (getWatchList state’) \land
InvariantWatchListsCharacterization (getWatchList state’) (getWatch1 state’) (getWatch2 state’)

proof (cases getC state \neq [opposite (getCl state)])

next

let oppC = oppositeLiteralList (getC state)

proof

next

let ?l = getCl state
let ?l’ = getLastAssertedLiteral (removeAll ?l oppC) (elements (getM state))
have clauseFalse (getC state) (elements (getM state))
using (getConflictFlag state)
using (InvariantCFalse (getConflictFlag state) (getM state) (getC state))
unfolding InvariantCFalse-def
by simp

from True
have set (getC state) ≠ {opposite ?l}
using (InvariantUniqC (getC state))
using uniqOneElementCharacterization[of getC state opposite ?l]
unfolding InvariantUniqC-def
by (simp add: Let-def)

have isLastAssertedLiteral ?l ?oppC (elements (getM state))
using (InvariantClCharacterization (getCl state) (getC state) (getM state))
unfolding InvariantClCharacterization-def
by simp

have opposite ?l el (getC state)
using (isLastAssertedLiteral ?l ?oppC (elements (getM state)));
unfolding isLastAssertedLiteral-def
by simp

have removeAll ?l ?oppC ≠ []
proof –
{
assume ¬thesis
hence set ?oppC ⊆ {?l}
using set-removeAll[of ?l ?oppC]
by auto
have set (getC state) ⊆ {opposite ?l}
proof
fix x
assume x ∈ set (getC state)

hence opposite x ∈ set ?oppC
using literalElListIffOppositeLiteralElOppositeLiteralList[of x (getC state)]
by simp
hence opposite x ∈ {?l}
using (set ?oppC ⊆ {?l})
by auto
thus x ∈ {opposite ?l}
using oppositeSymmetry[of x {?l}]
by force
qed
hence False
  using (set (getC state) ≠ {opposite ?l})
  using (opposite ?l el getC state)
  by (auto simp add: Let-def)
} thus ?thesis
  by auto
qed

have clauseFalse (oppositeLiteralList (removeAll ?l ?oppC)) (elements (getM state))
  using ⟨clauseFalse (getC state) (elements (getM state)): ⟩
  using oppositeLiteralListRemove[of ?l ?oppC]
  by (simp add: clauseFalseIffAllLiteralsAreFalse)
moreover
have oppositeLiteralList (removeAll ?l ?oppC) ≠ []
  using (removeAll ?l ?oppC ≠ [])
  using oppositeLiteralListNonempty
  by simp
ultimately
have isLastAssertedLiteral ?ll (removeAll ?l ?oppC) (elements (getM state))
  using ⟨InvariantUniq (getM state): ⟩
  unfolding InvariantUniq-def
  using getLastAssertedLiteralCharacterization[of oppositeLiteralList (removeAll ?l ?oppC) elements (getM state)]
  by auto
hence ?ll el (removeAll ?l ?oppC)
  unfolding isLastAssertedLiteral-def
  by auto
hence ?l el ?oppC ?ll ≠ ?l
  by auto
hence opposite ?ll el (getC state)
  by auto

let ?state' = applyLearn state

have InvariantWatchesEl (getF ?state') (getWatch1 ?state') (getWatch2 ?state')
proof –
  { fix clause::nat
    assume 0 ≤ clause ∧ clause < length (getF ?state')
    have ∃w1 w2. getWatch1 ?state' clause = Some w1 ∧
      getWatch2 ?state' clause = Some w2 ∧
      w1 el (getF ?state' ! clause) ∧ w2 el (getF ?state' ! clause)
  }
proof (cases clause < length (getF state))
case True
  thus ?thesis
    using 'InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state):
      unfolding InvariantWatchesEl-def
      using (set (getC state) ≠ {opposite ?l})
      unfolding applyLearn-def
      unfolding setWatch1-def
      unfolding setWatch2-def
      by (auto simp add: Let-def nth-append)
next
case False
  with (0 ≤ clause ∧ clause < length (getF ?state'))
  have clause = length (getF state)
    using (getC state ≠ [opposite ?l])
    unfolding applyLearn-def
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def)
moreover
  have getWatch1 ?state' clause = Some (opposite ?l) getWatch2
    ?state' clause = Some (opposite ?ll)
    using (clause = length (getF state))
    using (set (getC state) ≠ {opposite ?l})
    unfolding applyLearn-def
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def)
moreover
  have getF ?state' ! clause = (getC state)
    using (clause = length (getF state))
    using (set (getC state) ≠ {opposite ?l})
    unfolding applyLearn-def
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def)
ultimately
  show ?thesis
    using (opposite ?l el (getC state)) (opposite ?ll el (getC state))
    by force
  qed
} thus ?thesis
  unfolding InvariantWatchesEl-def
  by auto
qed
moreover
have InvariantWatchesDiffer (getF ?state') (getWatch1 ?state') (getWatch2 ?state')

618
proof
{
  fix clause :: nat
assume 0 ≤ clause ∧ clause < length (getF ?state')
have getWatch1 ?state' clause ≠ getWatch2 ?state' clause
proof (cases clause < length (getF ?state))
case True
  thus ?thesis
  using ⟨InvariantWatchesDiffer (getF state) (getWatch1 state)⟩
  unfolding InvariantWatchesDiffer-def
  using ⟨set (getC state) ≠ {opposite ?l}⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def nth-append)
next
case False
  with ⟨0 ≤ clause ∧ clause < length (getF ?state')⟩
have clause = length (getF state)
  using ⟨getC state ≠ {opposite ?l}⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
moreover
  have getWatch1 ?state' clause = Some (opposite ?l) getWatch2 ?state' clause = Some (opposite ?ll)
  using ⟨clause = length (getF state)⟩
  using ⟨set (getC state) ≠ {opposite ?l}⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
moreover
  have getF ?state' ! clause = (getC state)
  using ⟨clause = length (getF state)⟩
  using ⟨set (getC state) ≠ {opposite ?l}⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
ultimately
  show ?thesis
  using ⟨?ll ≠ ?l⟩
  by force
qed
} thus ?thesis
unfolding InvariantWatchesDiffer-def
by auto
qed
moreover
have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getM ?state')
proof
\{ 
  fix clause::nat and w1::Literal and w2::Literal
  assume *: 0 ≤ clause ∧ clause < length (getF ?state')
  assume **: Some w1 = getWatch1 ?state' clause Some w2 =
  getWatch2 ?state' clause
  have watchCharacterizationCondition w1 w2 (getM ?state') (getF
  ?state' ! clause) ∧
  watchCharacterizationCondition w2 w1 (getM ?state') (getF
  ?state' ! clause)
  proof (cases clause < length (getF state))
    case True
    thus ?thesis
    using ⟨InvariantWatchCharacterization (getF state) (getWatch1
    state) (getWatch2 state) (getM state)⟩
    unfolding InvariantWatchCharacterization-def
    using (set (getC state) ≠ {opposite ?l})
    using **
    unfolding applyLearn-def
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def nth-append)
  next
    case False
    with ⟨0 ≤ clause ∧ clause < length (getF ?state')⟩
    have clause = length (getF state)
    using ⟨getC state ≠ [opposite ?l]⟩
    unfolding applyLearn-def
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def)
  moreover
  have getWatch1 ?state' clause = Some (opposite ?l) getWatch2
  ?state' clause = Some (opposite ?ll)
  using ⟨clause = length (getF state)⟩
  using ⟨set (getC state) ≠ {opposite ?l}⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
  moreover
  have ∀ l. l el (getC state) ∧ l ≠ opposite ?l ∧ l ≠ opposite ?ll
  →
  elementLevel (opposite l) (getM state) ≤ elementLevel
  620
\(?l \ (\text{getM} \ \text{state}) \land \ \text{elementLevel} \ (\text{opposite} \ l) \ (\text{getM} \ \text{state}) \leq \ \text{elementLevel} \ \text{getM} \ \text{state})\)

\ ?ll \ (\text{getM} \ \text{state})

\text{proof}\ {-}
\{
\text{fix} \ l
\text{assume} \ l \ el \ (\text{getC} \ \text{state}) \ l \neq \text{opposite} \ ?l \ ?l \neq \text{opposite} \ ?ll
\text{hence} \ \text{opposite} \ l \ el \ ?oppC
\text{using} \ \text{literalElListIfOppositeLiteralElOppositeLiteralList}[of \ l \ get\ C \ \text{state}]
\text{by} \ \text{simp}
\}

\text{moreover}
\text{from} \ ?l \neq \text{opposite} \ ?ll
\text{have} \ \text{opposite} \ l \neq \ ?ll
\text{using} \ \text{oppositeSymmetry}[of \ l \ ?ll]
\text{by} \ \text{blast}

\text{ultimately}
\text{have} \ \text{opposite} \ l \ el \ (\text{removeAll} \ ?l \ ?oppC)
\text{by} \ \text{simp}

\text{from} \ (\text{clauseFalse} \ (\text{getC} \ \text{state}) \ (\text{elements} \ (\text{getM} \ \text{state})))
\text{have} \ \text{literalFalse} \ l \ (\text{elements} \ (\text{getM} \ \text{state}))
\text{using} \ (l \ el \ (\text{getC} \ \text{state}))
\text{by} \ (\text{simp add} \colon \text{clauseFalseIffAllLiteralsAreFalse})

\text{hence} \ \text{elementLevel} \ (\text{opposite} \ l) \ (\text{getM} \ \text{state}) \leq \ \text{elementLevel} \ \text{getM} \ \text{state})
\land \ \text{elementLevel} \ (\text{opposite} \ l) \ (\text{getM} \ \text{state}) \leq \ \text{elementLevel} \ ?ll
\text{getM} \ \text{state})
\text{using} \ (\text{InvariantUniq} \ (\text{getM} \ \text{state}))
\text{unfolding} \ \text{InvariantUniq-def}
\text{using} \ (\text{isLastAssertedLiteral} \ ?l \ ?oppC \ (\text{elements} \ (\text{getM} \ \text{state})))
\text{using} \ \text{lastAssertedLiteralHasHighestElementLevel}[of \ ?l \ ?oppC \ \text{getM} \ \text{state}]
\text{using} \ (\text{isLastAssertedLiteral} \ ?ll \ (\text{removeAll} \ ?l \ ?oppC) \ (\text{elements} \ (\text{getM} \ \text{state})))
\text{using} \ \text{lastAssertedLiteralHasHighestElementLevel}[of \ ?ll \ ?oppC \ \text{getM} \ \text{state}]
\text{using} \ (\text{opposite} \ l \ el \ ?oppC) \ (\text{opposite} \ l \ el \ (\text{removeAll} \ ?l \ ?oppC))
\text{by} \ \text{simp}
\}

\text{thus} \ ?\text{thesis}
\text{by} \ \text{simp}
\text{qed}

\text{moreover}
\text{have} \ \text{getF} \ ?\text{state}' ! \ \text{clause} = \ (\text{getC} \ \text{state})
\text{using} \ (\text{clause} = \ \text{length} \ (\text{getF} \ \text{state}))
\text{using} \ (\text{set} \ (\text{getC} \ \text{state}) \neq \ \{\text{opposite} \ ?l\})
unfolding applyLearn-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def)
moreover
have getM ?state' = getM state
  using (set (getC state) ≠ {opposite ?l});
unfolding applyLearn-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def)
ultimately
show ?thesis
  using ⟨clauseFalse (getC state) (elements (getM state))⟩;
  using **
  unfolding watchCharacterizationCondition-def
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
qed
} thus ?thesis
unfolding InvariantWatchCharacterization-def
by auto
qed
moreover
have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state') (getF ?state')
proof –
{ fix clause::nat and literal::Literal
  assume clause ∈ set (getWatchList ?state' literal)
  have clause < length (getF ?state')
  proof (cases clause ∈ set (getWatchList state literal))
    case True
    thus ?thesis
    using ⟨InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)⟩;
    unfolding InvariantWatchListsContainOnlyClausesFromF-def
    using (set (getC state) ≠ {opposite ?l});
    unfolding applyLearn-def
    unfolding setWatch1-def
    unfolding setWatch2-def
    by (auto simp add: Let-def nth-append) (force)+
next
   case False
  with ⟨clause ∈ set (getWatchList ?state' literal)⟩
  have clause = length (getF state)
  using (set (getC state) ≠ {opposite ?l});
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def

622
by (auto simp add: Let-def nth-append split: split-if-asm)
thus ?thesis
using (set (getC state) \neq \{ opposite ?l\})
unfolding applyLearn-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def nth-append)

qed

thus ?thesis
unfolding InvariantWatchListsContainOnlyClausesFromF-def
by simp

qed

moreover
have InvariantWatchListsUniq (getWatchList ?state')
unfolding InvariantWatchListsUniq-def
proof
fix l::Literal
show uniq (getWatchList ?state' l)
proof(cases l = opposite ?l \lor l = opposite ?ll)
case True
hence getWatchList ?state' l = (length (getF state)) \# getWatchList state l
using (set (getC state) \neq \{ opposite ?l\})
unfolding applyWatch1-def
unfolding setWatch1-def
unfolding setWatch2-def
using (?ll \neq ?l)
by (auto simp add: Let-def nth-append)
moreover
have length (getF state) \notin set (getWatchList state l)
using :InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
unfolding InvariantWatchListsContainOnlyClausesFromF-def
by auto
ultimately
show ?thesis
using (InvariantWatchListsUniq (getWatchList state))
unfolding InvariantWatchListsUniq-def
by (simp add: uniqAppendIff)

next
case False
hence getWatchList ?state' l = getWatchList state l
using (set (getC state) \neq \{ opposite ?l\})
unfolding applyWatch1-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add: Let-def nth-append)
thus ?thesis
using (InvariantWatchListsUniq (getWatchList state))
unfolding InvariantWatchListsUniq-def
  by simp
qed
qed
moreover
  have InvariantWatchListsCharacterization (getWatchList ?state)
       (getWatch1 ?state') (getWatch2 ?state')
  proof
    { 
      fix c::nat and l::Literal
      have (c ∈ set (getWatchList ?state' l)) = (Some l = getWatch1
           ?state' c ∨ Some l = getWatch2 ?state' c)
      proof (cases c = length (getF state))
        case False
        thus ?thesis
          using ⟨InvariantWatchListsCharacterization (getWatchList
           state) (getWatch1 state) (getWatch2 state)⟩
        unfolding InvariantWatchListsCharacterization-def
        using ⟨set (getC state) ≠ {opposite ?l}⟩
        unfolding applyLearn-def
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add:Let-def nth-append)
      next
        case True
        have length (getF state) ∈ set (getWatchList state l)
          using ⟨InvariantWatchListsContainOnlyClausesFromF (getWatchList
           state) (getF state)⟩
        unfolding InvariantWatchListsContainOnlyClausesFromF-def
        by auto
        thus ?thesis
          using ⟨c = length (getF state)⟩
        unfolding InvariantWatchListsCharacterization (getWatchList
           state) (getWatch1 state) (getWatch2 state)
        unfolding InvariantWatchListsCharacterization-def
        using ⟨set (getC state) ≠ {opposite ?l}⟩
        unfolding applyLearn-def
        unfolding setWatch1-def
        unfolding setWatch2-def
        by (auto simp add:Let-def nth-append)
      qed
    } thus ?thesis
  unfolding InvariantWatchListsCharacterization-def
  by simp
qed
moreover
  have InvariantClCharacterization (getCl ?state) (getC ?state') (getM
       ?state')
  using ⟨InvariantClCharacterization (getCl state) (getC state) (getM
       state)⟩

624
state)
  using (set (getC state) ≠ {opposite ?l})
unfolding applyLearn-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add:Let-def)
moreover
have InvariantCllCharacterization (getC state) (getCll state)
  (getCl state) (getM state)
  unfolding InvariantCllCharacterization-def
  using (isLastAssertedLiteral ?ll (removeAll ?l ?oppC) (elements (getM state))
  using (set (getC state) ≠ {opposite ?l})
unfolding applyLearn-def
unfolding setWatch1-def
unfolding setWatch2-def
by (auto simp add:Let-def)
ultimately
show ?thesis
  by simp
qed

lemma InvariantCllCharacterizationAfterApplyLearn:
assumes
  InvariantUniq (getM state)
  InvariantCllCharacterization (getCl state) (getC state) (getM state)
  InvariantCllFalse (getConflictFlag state) (getM state) (getC state)
  InvariantUniqC (getC state)
  getConflictFlag state
shows
  let state' = applyLearn state in
    InvariantCllCharacterization (getCl state') (getCll state') (getC state') (getM state')
proof (cases getC state ≠ [opposite (getC state)])
case False
  thus ?thesis
  using assms
  unfolding applyLearn-def
unfolding InvariantCllCharacterization-def
by (simp add: Let-def)
next
case True

  let ?oppC = oppositeLiteralList (getC state)
  let ?l = getCl state
  let ?ll = getLastAssertedLiteral (removeAll ?l ?oppC) (elements (getM state))

  have clauseFalse (getC state) (elements (getM state))
using ⟨getConflictFlag state⟩
using ⟨InvariantCFalse (getConflictFlag state) (getM state) (getC state)⟩

unfolding InvariantCFalse-def
by simp

from True
have set (getC state) ≠ {opposite ?l}
using ⟨InvariantUniqC (getC state)⟩
using uniqOneElementCharacterization[of getC state opposite ?l]
unfolding InvariantUniqC-def
by (simp add: Let-def)

have isLastAssertedLiteral ?l ?oppC (elements (getM state))
using ⟨InvariantClCharacterization (getCl state) (getC state) (getM state)⟩
unfolding InvariantClCharacterization-def
by simp

have opposite ?l el (getC state)
using ⟨isLastAssertedLiteral ?l ?oppC (elements (getM state))⟩
unfolding isLastAssertedLiteral-def
by simp

have removeAll ?l ?oppC ≠ []
proof –
{
  assume ¬ thesis
  hence set ?oppC ⊆ {?l}
    using set-removeAll[of ?l ?oppC]
  by auto
  have set (getC state) ⊆ {opposite ?l}
  proof
    fix x
    assume x ∈ set (getC state)
    hence opposite x ∈ set ?oppC
      using literalElListIffOppositeLiteralElOppositeLiteralList[of x getC state]
      by simp
    hence opposite x ∈ {?l}
      using ⟨set ?oppC ⊆ {?l}⟩
      by auto
    thus x ∈ {opposite ?l}
      using oppositeSymmetry[of x ?l]
      by force
  qed
  hence False
}

626
using \( \{ \text{set } (\text{getC state}) \neq \{ \text{opposite } ?l \} \} \)

using \( \{ \text{opposite } ?l \text{ el getC state} \} \)

by (auto simp add: Let-def)

\} thus \( \text{thesis} \)

by auto

qed

have clauseFalse \( \{ \text{oppositeLiteralList } (\text{removeAll } ?l \ ?oppC) \} \) \( \{ \text{elements } (\text{getM state}) \} \)

using \( \{ \text{clauseFalse } (\text{getC state}) \} \) \( \{ \text{elements } (\text{getM state}) \} \)

using oppositeLiteralListRemove \[ of ?l \ ?oppC \]

by (simp add: clauseFalseIffAllLiteralsAreFalse)

moreover

have oppositeLiteralList \( \{ \text{removeAll } ?l \ ?oppC \} \neq [] \)

using \( \{ \text{removeAll } ?l \ ?oppC \neq [] \} \)

using oppositeLiteralListNonempty

by simp

ultimately

have isLastAssertedLiteral \( ?ll \{ \text{removeAll } ?l \ ?oppC \} \) \( \{ \text{elements } (\text{getM state}) \} \)

using \( \{ \text{getLastAssertedLiteralCharacterization } \{ \text{removeAll } ?l \ ?oppC \} \} \) \( \{ \text{elements } (\text{getM state}) \} \)

using \( \{ \text{InvariantUniq } (\text{getM state}) \} \)

unfolding InvariantUniq-def

by auto

thus \( \text{thesis} \)

using \( \{ \text{set } (\text{getC state}) \neq \{ \text{opposite } ?l \} \} \)

unfolding applyLearn-def

unfolding setWatch1-def

unfolding setWatch2-def

unfolding InvariantCllCharacterization-def

by (auto simp add: Let-def)

qed

lemma InvariantConflictClauseCharacterizationAfterApplyLearn:

assumes

getConflictFlag state
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)

shows

let state' = applyLearn state in

InvariantConflictClauseCharacterization (getConflictFlag state')
(getConflictClause state') (getF state') (getM state')

proof-

have getConflictClause state < length (getF state)

using assms

unfolding InvariantConflictClauseCharacterization-def

by (auto simp add: Let-def)
hence \( n\mathit{th}((\mathit{getF\ state} )\ @ \ [\mathit{getC\ state}])\ (\mathit{getConflictClause\ state}) = \ n\mathit{th}((\mathit{getF\ state} )\ (\mathit{getConflictClause\ state})) \)
by \((\mathit{simp\ add: \ nth-append})\)
thus \( ?\mathit{thesis} \)
using \(\mathit{InvariantConflictClauseCharacterization\ characterization\ (\mathit{getConflictFlag\ state})\ (\mathit{getConflictClause\ state})\ (\mathit{getF\ state} )\ (\mathit{getM\ state}))\)
unfolding \(\mathit{InvariantConflictClauseCharacterization-def}\)
unfolding \(\mathit{applyLearn-def}\)
unfolding \(\mathit{setWatch1-def}\)
unfolding \(\mathit{setWatch2-def}\)
by \((\mathit{auto\ simp\ add: \ Let-def\ clauseFalseAppendValuation})\)
qed

lemma \(\mathit{InvariantGetReasonIsReasonAfterApplyLearn}\):
assumes
\(\mathit{InvariantGetReasonIsReason\ (\mathit{getReason\ state})\ (\mathit{getF\ state} )\ (\mathit{getM\ state})\ (\mathit{set\ (\mathit{getQ\ state}))}\)
shows
\(\mathit{let\ state'} = \mathit{applyLearn\ state\ in}\n\mathit{InvariantGetReasonIsReason\ (\mathit{getReason\ state}')\ (\mathit{getF\ state}')\ (\mathit{getM\ state}')\ (\mathit{set\ (\mathit{getQ\ state}'))}\)
proof \((\mathit{cases\ getC\ state = [\mathit{opposite\ (\mathit{getCl\ state})}]})\)
case \(\mathit{True}\)
thus \( ?\mathit{thesis} \)
unfolding \(\mathit{applyLearn-def}\)
using \(\mathit{assms}\)
by \((\mathit{simp\ add: \ Let-def})\)
next
case \(\mathit{False}\)
have \(\mathit{InvariantGetReasonIsReason\ (\mathit{getReason\ state})\ ((\mathit{getF\ state} )\ @ \ [\mathit{getC\ state}])\ (\mathit{getM\ state})\ (\mathit{set\ (\mathit{getQ\ state}))}\)
using \(\mathit{assms}\)
using \(\mathit{nth-append[of\ getF\ state\ [getC\ state]]}\)
unfolding \(\mathit{InvariantGetReasonIsReason-def}\)
by \(\mathit{auto}\)
thus \( ?\mathit{thesis} \)
using \(\mathit{False}\)
unfolding \(\mathit{applyLearn-def}\)
unfolding \(\mathit{setWatch1-def}\)
unfolding \(\mathit{setWatch2-def}\)
by \((\mathit{simp\ add: \ Let-def})\)
qed

lemma \(\mathit{InvariantQCharacterizationAfterApplyLearn}\):
assumes
\(\mathit{getConflictFlag\ state}\)
\(\mathit{InvariantQCharacterization\ (\mathit{getConflictFlag\ state})\ (\mathit{getQ\ state})\ (\mathit{getF\ state})\ (\mathit{getM\ state})}\)
shows
    let state' = applyLearn state in
    InvariantQCharacterization (getConflictFlag state') (getQ state')
    (getF state') (getM state')
  using assms
  unfolding InvariantQCharacterization-def
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (simp add: Let-def)

lemma InvariantUniqQAfterApplyLearn:
  assumes 
    InvariantUniqQ (getQ state)
  shows
    let state' = applyLearn state in
    InvariantUniqQ (getQ state')
  using assms
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (simp add: Let-def)

lemma InvariantConflictFlagCharacterizationAfterApplyLearn:
  assumes
    getConflictFlag state
    InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
  shows
    let state' = applyLearn state in
    InvariantConflictFlagCharacterization (getConflictFlag state')
    (getF state') (getM state')
  using assms
  unfolding InvariantConflictFlagCharacterization-def
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def formulaFalseIffContainsFalseClause)

lemma InvariantNoDecisionsWhenConflictNorUnitAfterApplyLearn:
  assumes
    InvariantUniq (getM state)
    InvariantConsistent (getM state)
    InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))
    InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))
    InvariantCFalse (getConflictFlag state) (getM state) (getC state)
  and
InvariantClCharacterization (getCl state) (getC state) (getM state)
InvariantClCurrentLevel (getCl state) (getM state)
InvariantUniqC (getC state)

getConflictFlag state
isUIP (opposite (getCl state)) (getC state) (getM state)
currentLevel (getM state) > 0

shows
let state' = applyLearn state in
InvariantNoDecisionsWhenConflict (getF state) (getM state')
(currentLevel (getM state')) ∧
InvariantNoDecisionsWhenUnit (getF state) (getM state') (currentLevel (getM state')) ∧
InvariantNoDecisionsWhenConflict [getC state] (getM state') (getBackjumpLevel state') ∧
InvariantNoDecisionsWhenUnit [getC state] (getM state') (getBackjumpLevel state')

proof–
let ?state' = applyLearn state
let ?l = getCl state

have clauseFalse (getC state) (elements (getM state))
  using (getConflictFlag state)
  using (InvariantCFalse (getConflictFlag state) (getM state) (getC state))
  unfolding InvariantCFalse-def
  by simp

have getM ?state' = getM state getCl ?state' = getCl state getConflictFlag ?state' = getConflictFlag state
  unfolding applyLearn-def
  unfolding setWatch2-def
  unfolding setWatch1-def
  by (auto simp add: Let-def)

hence InvariantNoDecisionsWhenConflict (getF state) (getM ?state')
  (currentLevel (getM ?state')) ∧
  InvariantNoDecisionsWhenUnit (getF state) (getM ?state')
  (currentLevel (getM ?state'))∧
  using (InvariantNoDecisionsWhenConflict (getF state) (getM state)
  (currentLevel (getM state))))
  using (InvariantNoDecisionsWhenUnit (getF state) (getM state)
  (currentLevel (getM state))))
  by simp

moreover
have InvariantClCharacterization (getCl ?state') (getCl ?state')
  (getC ?state') (getM ?state')
  using assms
using InvariantCLIClCharacterizationAfterApplyLearn[of \ state]

by (simp add: Let-def)

hence isMinimalBackjumpLevel (getBackjumpLevel \ ?state\') (opposite \ ?l) (getC \ ?state\') (getM \ ?state')

using assms

using \ getM \ ?state\' = getM \ state \ getC \ ?state\' = getC \ state \ getCl \ ?state\' = getCl \ state \ getConflictFlag \ ?state\' = getConflictFlag \ state

using isMinimalBackjumpLevelGetBackjumpLevel[of \ ?state']

unfolding isUIP-def

unfolding SatSolverVerification.isUIP-def

by (simp add: Let-def)

hence getBackjumpLevel \ ?state\' < elementLevel \ ?l (getM \ ?state')

unfolding isMinimalBackjumpLevel-def

unfolding isBackjumpLevel-def

by simp

hence getBackjumpLevel \ ?state\' < currentLevel (getM \ ?state')

using elementLevelLeqCurrentLevel[of \ ?l getM \ ?state']

by simp

have InvariantNoDecisionsWhenConflict \ [getC \ state] (getM \ ?state') (getBackjumpLevel \ ?state') ∧

InvariantNoDecisionsWhenUnit \ [getC \ state] (getM \ ?state') (getBackjumpLevel \ ?state')

proof –

{ fix clause::Clause

assume clause el [getC \ state]

hence clause = getC \ state

by simp

have \ (∀ \ level'. \ level' < (getBackjumpLevel \ ?state') \ →

\ ¬ \ clauseFalse \ clause \ (elements \ (prefixToLevel \ level' \ (getM \ ?state')))) \ ∧

\ (∀ \ level'. \ level' < (getBackjumpLevel \ ?state') \ →

\ ¬ \ (∃ \ l. \ isUnitClause \ clause \ l \ (elements \ (prefixToLevel \ level' \ (getM \ ?state'))))) \ (is \ ?false \ ∧ \ ?unit)

proof (cases \ getC \ state = \ [opposite \ ?l])

case True

thus \ ?thesis

using \ (getM \ ?state' = getM \ state) \ (getC \ ?state' = getC \ state)

\ (getCl \ ?state' = getCl \ state)

unfolding getBackjumpLevel-def

by (simp add: Let-def)

next

case False

hence getF \ ?state' = getF \ state \ @ [getC \ state]

unfolding applyLearn-def

unfolding setWatch2-def

631
unfolding setWatch1-def
by (auto simp add: Let-def)

show ?thesis
proof
  have ?unit
    using ⟨clause = getC state⟩
    using ⟨InvariantUniq (getM state)⟩
    using ⟨InvariantConsistent (getM state)⟩
    using ⟨getM ?state' = getM state; getC ?state' = getC state⟩
    using ⟨clauseFalse (getC state) (elements (getM state))⟩
    using ⟨isMinimalBackjumpLevel (getBackjumpLevel ?state') (opposite ?!) (getC ?state') (getM ?state')⟩
    unfolding InvariantUniq-def
    unfolding InvariantConsistent-def
    by simp
  moreover
    have isUnitClause (getC state) (opposite ?!) (elements (prefixToLevel (getBackjumpLevel ?state') (getM state)))
    using ⟨InvariantUniq (getM state)⟩
    using ⟨InvariantConsistent (getM state)⟩
    using ⟨isMinimalBackjumpLevel (getBackjumpLevel ?state') (opposite ?!) (getC ?state') (getM ?state')⟩
    unfolding InvariantUniq-def
    unfolding InvariantConsistent-def
    by simp
  hence ¬ clauseFalse (getC state) (elements (prefixToLevel (getBackjumpLevel ?state') (getM state)))
  unfolding isUnitClause-def
  by (auto simp add: clauseFalseIffAllLiteralsAreFalse)
  have ?false
proof
  fix level'
  show level' < getBackjumpLevel ?state' ⟷ ¬ clauseFalse clause (elements (prefixToLevel level' (getM ?state')))
  proof
    assume level' < getBackjumpLevel ?state'
    show ¬ clauseFalse clause (elements (prefixToLevel level' (getM ?state')))
    proof
      have isPrefix (prefixToLevel level' (getM state)) (prefixToLevel (getBackjumpLevel ?state') (getM state))
using \langle \text{level}' < \text{getBackjumpLevel} \ ?\text{state}' \rangle

using \text{isPrefixPrefixToLevelLowerLevel}[^{\text{of level}' \text{getBackjumpLevel} \ ?\text{state}' \text{getM} \ ?\text{state}}]

by simp

then obtain \ s

where \text{prefixToLevel} level' (\text{getM} \ ?\text{state}') \ @ \ s = \text{prefixToLevel} (\text{getBackjumpLevel} \ ?\text{state}') (\text{getM} \ ?\text{state})

unfolding \text{isPrefix-def}

by auto

hence \text{prefixToLevel} (\text{getBackjumpLevel} \ ?\text{state}') (\text{getM} \ ?\text{state}) = \text{prefixToLevel} level' (\text{getM} \ ?\text{state}) \ @ \ s

by (rule sym)

thus \ ?\text{thesis}

using (\text{getM} \ ?\text{state}' = \text{getM} \ ?\text{state})

using (\text{clause} = \text{getC} \ ?\text{state})

using (\neg \text{clauseFalse} (\text{getC} \ ?\text{state}) (\text{elements} (\text{prefixToLevel} (\text{getBackjumpLevel} \ ?\text{state}') (\text{getM} \ ?\text{state}))))

unfolding \text{isPrefix-def}

by (auto simp add: \text{clauseFalseIffAllLiteralsAreFalse})

qed

qed

ultimately

show \ ?\text{thesis}

by simp

qed

qed

} thus \ ?\text{thesis}

unfolding \text{InvariantNoDecisionsWhenConflict-def}

unfolding \text{InvariantNoDecisionsWhenUnit-def}

by (auto simp add: \text{formulaFalseIffContainsFalseClause})

qed

ultimately

show \ ?\text{thesis}

by (simp add: \text{Let-def})

qed

lemma \text{InvariantEquivalentZLAfterApplyLearn}:

assumes

\text{InvariantEquivalentZL} (\text{getF} \ ?\text{state}) (\text{getM} \ ?\text{state}) F0 \ \text{and}

\text{InvariantCEntailed} (\text{getConflictFlag} \ ?\text{state}) F0 (\text{getC} \ ?\text{state}) \ \text{and}

\text{getConflictFlag} \ ?\text{state}

shows

let \ ?\text{state}' = \text{applyLearn} \ ?\text{state} in

\text{InvariantEquivalentZL} (\text{getF} \ ?\text{state}') (\text{getM} \ ?\text{state}') F0

proof

let \ ?\text{M0} = \text{val2form} (\text{elements} (\text{prefixToLevel} 0 (\text{getM} \ ?\text{state})))

have \text{equivalentFormulae} F0 (\text{getF} \ ?\text{state} \ @ \ ?\text{M0})

using (\text{InvariantEquivalentZL} (\text{getF} \ ?\text{state}) (\text{getM} \ ?\text{state}) F0)
using equivalentFormulaeSymmetry[of F0 getF state @ ?M0]
unfolding InvariantEquivalentZL-def
by simp

moreover
have formulaEntailsClause (getF state @ ?M0) (getC state)
  using assms
  unfolding InvariantEquivalentZL-def
  unfolding InvariantCEntailed-def
  unfolding equivalentFormulae-def
  unfolding formulaEntailsClause-def
  by auto
ultimately
have equivalentFormulae F0 ((getF state @ ?M0) @ [getC state])
  using extendEquivalentFormulaWithEntailedClause[of F0 getF state @ ?M0 getC state]
  by simp
hence equivalentFormulae ((getF state @ ?M0) @ [getC state]) F0
  by (simp add: equivalentFormulaeSymmetry)
have equivalentFormulae ((getF state) @ [getC state] @ ?M0) F0
proof –
  { fix valuation::Valuation
      have formulaTrue ((getF state @ ?M0) @ [getC state]) valuation
    = formulaTrue ((getF state) @ [getC state] @ ?M0) valuation
    by (simp add: formulaTrueIffAllClausesAreTrue)
  }
thus ?thesis
  using (equivalentFormulae ((getF state @ ?M0) @ [getC state]) F0)
  unfolding equivalentFormulae-def
  by auto
qed
thus ?thesis
  using assms
  unfolding InvariantEquivalentZL-def
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
qed

lemma InvariantVarsFAfterApplyLearn:
assumes
  InvariantCFalse (getConflictFlag state) (getM state) (getC state)
  getConflictFlag state
  InvariantVarsF (getF state) F0 Vbl
  InvariantVarsM (getM state) F0 Vbl
shows
let state' = applyLearn state in
InvariantVarsF (getF state') F0 Vbl

proof –
from assms
have clauseFalse (getC state) (elements (getM state))
  unfolding InvariantCFalse-def
  by simp
hence vars (getC state) ⊆ vars (elements (getM state))
  using valuationContainsItsFalseClausesVariables[of getC state elements (getM state)]
  by simp
thus ?thesis
  using applyLearnPreservedVariables[of state]
  using assms
  using varsAppendFormulae[of getF state [getC state]]
  unfolding InvariantVarsF-def
  unfolding InvariantVarsM-def
  by (auto simp add: Let-def)
qed

lemma applyBackjumpEffect:
assumes
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
  getConflictFlag state
  InvariantCFalse (getConflictFlag state) (getM state) (getC state)
and
  InvariantCEntailed (getConflictFlag state) F0 (getC state) and
  InvariantClCharacterization (getCl state) (getC state) (getM state)
and
  InvariantClCharacterization (getCl state) (getC state) (getM state)
  (getM state) and
  InvariantClCurrentLevel (getCl state) (getM state)
  InvariantUniqC (getC state)
  isUIP (opposite (getCl state)) (getC state) (getM state)
  currentLevel (getM state) > 0
shows
let l = (getCl state) in
let bClause = (getC state) in
let bLiteral = opposite l in
let level = getBackjumpLevel state in
let prefix = prefixToLevel level (getM state) in
let state'' = applyBackjump state in
(formulaEntailsClause F0 bClause \and
  isUnitClause bClause bLiteral (elements prefix) \and
  (getM state'') = prefix @ [(bLiteral, False)] \and
  getF state'' = getF state)

proof
  let ?l = getCl state
  let ?level = getBackjumpLevel state
  let ?prefix = prefixToLevel ?level (getM state)
  let ?state' = state(! getConflictFlag := False, getQ := [], getM := ?prefix )
  let ?state'' = applyBackjump state

  have clauseFalse (getC state) (elements (getM state))
    using (getConflictFlag state)
    using (InvariantCFalse (getConflictFlag state) (getM state) (getC state))
    unfolding InvariantCFalse-def
    by simp

  have formulaEntailsClause F0 (getC state)
    using (getConflictFlag state)
    using (InvariantCEntailed (getConflictFlag state) F0 (getC state))
    unfolding InvariantCEntailed-def
    by simp

  have isBackjumpLevel ?level (opposite ?l) (getC state) (getM state)
    using assms
    using isMinimalBackjumpLevelGetBackjumpLevel[of state]
    unfolding isMinimalBackjumpLevel-def
    by (simp add: Let-def)
  then have isUnitClause (getC state) (opposite ?l) (elements ?prefix)
    using assms
    using (clauseFalse (getC state) (elements (getM state))): using isBackjumpLevelEnsuresIsUnitInPrefix[of getM state getC state ?level opposite ?l]
    unfolding InvariantConsistent-def
    unfolding InvariantUniq-def
    by simp
  moreover
  have getM ?state'' = ?prefix @ [(opposite ?l, False)] getF ?state'' = getF state
    unfolding applyBackjump-def
    using assms

  636
using assertLiteralEffect
unfolding setReason-def
by (auto simp add: Let-def)
ultimately
show ?thesis
using ⟨formulaEntailsClause F0 (getC state)⟩
by (simp add: Let-def)
qed

lemma applyBackjumpPreservedVariables:
assumes
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
InvariantWatchesEI (getF state) (getWatch1 state) (getWatch2 state)
shows
let state' = applyBackjump state in
getSATFlag state' = getSATFlag state
using assms
unfolding applyBackjump-def
unfolding setReason-def
by (auto simp add: Let-def assertLiteralEffect)

lemma InvariantWatchCharacterizationInBackjumpPrefix:
assumes
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
shows
let l = getCl state in
let level = getBackjumpLevel state in
let prefix = prefixToLevel level (getM state) in
let state' = state [] getConflictFlag := False, getQ := [], getM := prefix []
InvariantWatchCharacterization (getF state') (getWatch1 state') (getWatch2 state') (getM state')
proof -
let ?l = getCl state
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level (getM state)
let ?state' = state [] getConflictFlag := False, getQ := [], getM := ?prefix []
{
  fix c w1 w2
  assume c < length (getF state) Some w1 = getWatch1 state c
  Some w2 = getWatch2 state c
  with ⟨InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)⟩
have watchCharacterizationCondition w1 w2 (getM state) (nth (getF state) c)
watchCharacterizationCondition w2 w1 (getM state) (nth (getF state) c)

unfolding InvariantWatchCharacterization-def
by auto

let ?clause = nth (getF state) c
let ?a state w1 w2 = \exists l. l el ?clause \land literalTrue l (elements (getM state)) \land
  elementLevel l (getM state) \leq elementLevel (opposite w1) (getM state)
let ?b state w1 w2 = \forall l. l el ?clause \land l \neq w1 \land l \neq w2 \rightarrow
  literalFalse l (elements (getM state)) \land
  elementLevel (opposite l) (getM state) \leq elementLevel (opposite w1) (getM state)

have watchCharacterizationCondition w1 w2 (getM ?state') ?clause \land
  watchCharacterizationCondition w2 w1 (getM ?state') ?clause

proof[
  { assume literalFalse w1 (elements (getM ?state'))
  hence literalFalse w1 (elements (getM state))
  using isPrefixPrefixToLevel[of ?level getM state]
  using isPrefixElements[of prefixToLevel ?level (getM state)]
  getM state]
  using prefixIsSubset[of elements (prefixToLevel ?level (getM state)) elements (getM state)]
  by auto

  from (literalFalse w1 (elements (getM ?state')))
  have elementLevel (opposite w1) (getM state) \leq ?level
  using prefixToLevelElementsElementLevel[of opposite w1 ?level getM state]
  by simp

  from (literalFalse w1 (elements (getM ?state')))
  have elementLevel (opposite w1) (getM ?state') = elementLevel (opposite w1) (getM state)
  using elementLevelPrefixElement
  by simp

have ?a ?state' w1 w2 \lor ?b ?state' w1 w2
proof (cases ?a state w1 w2)
case True
then obtain l
  where l el ?clause literalTrue l (elements (getM state))
\[\text{elementLevel } l \text{ (getM state)} \leq \text{elementLevel (opposite } w1\text{)} \]

\[(\text{getM state})\]

by auto

have \(\text{literalTrue } l \text{ (elements (getM } ?\text{state}')})\)

using \(\text{elementLevel (opposite } w1\text{)} \text{ (getM state)} \leq ?\text{level})\)

using \(\text{elementLevelLtLevelImpliesMemberPrefixToLevel[of } l \text{ getM state } ?\text{level}]\)

using \(\text{elementLevel } l \text{ (getM state)} \leq \text{elementLevel (opposite } w1\text{)} \text{ (getM state)})\)

using \((\text{literalTrue } l \text{ (elements (getM state))})\)

by simp

moreover

from \(\text{literalTrue } l \text{ (elements (getM } ?\text{state}')})\)

have \(\text{elementLevel } l \text{ (getM } ?\text{state}') = \text{elementLevel } l \text{ (getM state)}\)

using \(\text{elementLevelPrefixElement}\)

by simp

ultimately

show \(\text{thesis}\)

using \(\text{elementLevel (opposite } w1\text{)} \text{ (getM } ?\text{state}') = \text{elementLevel (opposite } w1\text{)} \text{ (getM state)})\)

using \(\text{elementLevel } l \text{ (getM state)} \leq \text{elementLevel (opposite } w1\text{)} \text{ (getM state)})\)

using \(l \text{ el } ?\text{clause}\)

by auto

next

case False
{
fix \(l\)
assume \(l \text{ el } ?\text{clause } l \neq w1 l \neq w2\)

hence \(\text{literalFalse } l \text{ (elements (getM state))}\)

\(\text{elementLevel (opposite } l\text{) (getM state)} \leq \text{elementLevel (opposite } w1\text{) (getM state)})\)

using \(\text{literalFalse } w1 \text{ (elements (getM state))}\)

using \(\text{False}\)

using \(\text{watchCharacterizationCondition w1 w2 (getM state) } ?\text{clause})\)

unfolding \text{watchCharacterizationCondition-def}\n
by auto

have \(\text{literalFalse } l \text{ (elements (getM } ?\text{state}')}) \land\)

\(\text{elementLevel (opposite } l\text{) (getM } ?\text{state}') \leq \text{elementLevel (opposite } w1\text{) (getM } ?\text{state}')}\)

proof

have \(\text{literalFalse } l \text{ (elements (getM } ?\text{state}')})\)

using \(\text{elementLevel (opposite } w1\text{) (getM state)} \leq ?\text{level})\)

using \(\text{elementLevelLtLevelImpliesMemberPrefixToLevel[of opposite } l \text{ getM state } ?\text{level}]\)
using \( \text{elementLevel} (\text{opposite } l) \ (\text{getM} \ \text{state}) \leq \text{elementLevel} (\text{opposite } w1) \ (\text{getM} \ \text{state}) \)

by simp

moreover

from \( \text{literalFalse} \ l \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)

have \( \text{elementLevel} \ (\text{opposite } l) \ (\text{getM} \ ?\text{state}') = \text{elementLevel} \ (\text{opposite } l) \ (\text{getM} \ \text{state}) \)

using \( \text{elementLevelPrefixElement} \)

by simp

ultimately

show ?thesis

using \( \text{elementLevel} \ (\text{opposite } w1) \ (\text{getM} \ ?\text{state}') = \text{elementLevel} \ (\text{opposite } w1) \ (\text{getM} \ \text{state}) \)

using \( \text{elementLevel} \ (\text{opposite } l) \ (\text{getM} \ \text{state}) \leq \text{elementLevel} \ (\text{opposite } w1) \ (\text{getM} \ \text{state}) \)

using \( \text{elementLevelPrefixElement} \)

by simp

qed

thus ?thesis

by auto

qed

moreover

{ 
  assume \( \text{literalFalse} \ w2 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)

  hence \( \text{literalFalse} \ w2 \ (\text{elements} \ (\text{getM} \ \text{state})) \)

  using \( \text{isPrefixPrefixToLevel}[\text{of} \ ?\text{level} \ (\text{getM} \ \text{state})] \)

  using \( \text{isPrefixElements}[\text{of} \ (\text{prefixToLevel} \ ?\text{level} \ (\text{getM} \ \text{state})) \ (\text{getM} \ \text{state})] \)

  by auto

  from \( \text{literalFalse} \ w2 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)

  have \( \text{elementLevel} \ (\text{opposite } w2) \ (\text{getM} \ \text{state}) \leq \ ?\text{level} \)

  using \( \text{prefixToLevelElementsElementLevel}[\text{of} \ (\text{prefixToLevel} \ ?\text{level} \ (\text{getM} \ \text{state})) \ (\text{getM} \ \text{state})] \)

  by simp

  from \( \text{literalFalse} \ w2 \ (\text{elements} \ (\text{getM} \ ?\text{state}')) \)

  have \( \text{elementLevel} \ (\text{opposite } w2) \ (\text{getM} \ ?\text{state}') = \text{elementLevel} \ (\text{opposite } w2) \ (\text{getM} \ \text{state}) \)

  using \( \text{elementLevelPrefixElement} \)

  by simp

  have \( ?a \ ?\text{state}' \ w2 \ w1 \lor \ ?b \ ?\text{state}' \ w2 \ w1 \)

proof (cases \( ?a \ ?\text{state} \ w2 \ w1 \))

640
case True
then obtain l
where l el ?clause literalTrue l (elements (getM state))
  elementLevel l (getM state) ≤ elementLevel (opposite w2) (getM state)
  by auto

  have literalTrue l (elements (getM ?state'))
  using elementLevel (opposite w2) (getM state) ≤ ?level;
  using elementLevelLtLevelImpliesMemberPrefixToLevel[of l getM state ?level]
  using elementLevel l (getM state) ≤ elementLevel (opposite w2) (getM state)
  using literalTrue l (elements (getM state));
  by simp
moreover
from (literalTrue l (elements (getM ?state'))):
  have elementLevel l (getM ?state') = elementLevel l (getM state)
  using elementLevelPrefixElement
  by simp
ultimately
show ?thesis
  using (elementLevel (opposite w2) (getM ?state') =
  elementLevel (opposite w2) (getM state));
  using elementLevel l (getM state) ≤ elementLevel (opposite w2) (getM state);
  using l el ?clause
  by auto
next
case False
{
  fix l
  assume l el ?clause l ≠ w1 l ≠ w2
  hence literalFalse l (elements (getM state))
    elementLevel (opposite l) (getM state) ≤ elementLevel (opposite w2) (getM state)
    using (literalFalse w2 (elements (getM state)));
    using False
    using (watchCharacterizationCondition w2 w1 (getM state) ?clause)
      unfolding watchCharacterizationCondition-def
      by auto

  have literalFalse l (elements (getM ?state')) ∧
    elementLevel (opposite l) (getM ?state') ≤ elementLevel (opposite w2) (getM ?state')
    proof
      have literalFalse l (elements (getM ?state'))
using \langle\text{elementLevel} (\text{opposite} \ w2) (\text{getM} \ \text{state}) \leq \?\text{level}\rangle

using \langle\text{elementLevelLtLevelImpliesMemberPrefixToLevel}[\text{of} \ \text{opposite} \ l \ \text{getM} \ \text{state} \ ?\text{level}\rangle

using \langle\text{elementLevel} (\text{opposite} \ l) (\text{getM} \ \text{state})\rangle

by simp

moreover

from \langle\text{literalFalse} \ l \ (\text{elements} (\text{getM} \ ?\text{state}'))\rangle

have \text{elementLevel} (\text{opposite} \ l) (\text{getM} \ ?\text{state}') = \text{elementLevel} (\text{opposite} \ l) (\text{getM} \ \text{state})

by simp

ultimately

show \ ?\text{thesis}

using \langle\text{elementLevel} (\text{opposite} \ w2) (\text{getM} \ ?\text{state}') = \text{elementLevel} (\text{opposite} \ w2) (\text{getM} \ \text{state})\rangle

using \langle\text{elementLevel} (\text{opposite} \ l) (\text{getM} \ \text{state})\rangle

by \text{auto}

qed

}\)

thus \ ?\text{thesis}

by \text{auto}

qed

ultimately

show \ ?\text{thesis}

unfolding \text{watchCharacterizationCondition-def}

by \text{auto}

qed

}\)

thus \ ?\text{thesis}

unfolding \text{InvariantWatchCharacterization-def}

by \text{auto}

qed

\textbf{lemma} \textbf{InvariantConsistentAfterApplyBackjump}:  

\textbf{assumes}

\textbf{InvariantConsistent} (\text{getM} \ \text{state})

\textbf{InvariantUniq} (\text{getM} \ \text{state})

\textbf{InvariantWatchesEl} (\text{getF} \ \text{state}) (\text{getWatch1} \ \text{state}) (\text{getWatch2} \ \text{state})

\textbf{and}

\textbf{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList} \ \text{state}) (\text{getF} \ \text{state}) \ \textbf{and}

\text{getConflictFlag} \ \text{state}

\textbf{InvariantCFalse} (\text{getConflictFlag} \ \text{state}) (\text{getM} \ \text{state}) (\text{getC} \ \text{state})
and
InvariantUniqC (getC state)
InvariantCEntailed (getConflictFlag state) F0 (getC state) and
InvariantCICharacterization (getCl state) (getC state) (getM state)
and
InvariantCllCharacterization (getCl state) (getCl state) (getC state) (getM state) and
InvariantCICharacterization (getCl state) (getCl state) (getC state) (getM state)
and
InvariantCllCharacterization (getCl state) (getCl state) (getC state) (getM state)
currentLevel (getM state) > 0
isUIP (opposite (getCl state)) (getC state) (getM state)
shows
let state = applyBackjump state in
InvariantConsistent (getM state')

proof–
let ?l = getCl state
let ?bClause = getC state
let ?bLiteral = opposite ?l
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level (getM state)
let ?state'' = applyBackjump state

have formulaEntailsClause F0 ?bClause and
  isUnitClause ?bClause ?bLiteral (elements ?prefix) and
  getM ?state'' = ?prefix @ [(?bLiteral, False)]
using assms
using applyBackjumpEffect[of state]
by (auto simp add: Let-def)

thus ?thesis
using InvariantConsistent (getM state)
using InvariantConsistentAfterBackjump[of getM state ?prefix ?bClause ?bLiteral getM ?state'']
using isPrefixPrefixToLevel
by (auto simp add: Let-def)
qed

lemma InvariantUniqAfterApplyBackjump:
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
getConflictFlag state
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
and
InvariantUniqC (getC state)
InvariantCEntailed (getConflictFlag state) F0 (getC state) and
InvariantClCharacterization (getCl state) (getC state) (getM state)
and
InvariantClCharacterization (getCl state) (getClI state) (getC state) (getM state) and
InvariantClCurrentLevel (getCl state) (getM state) (currentLevel (getM state)) > 0
isUIP (opposite (getCl state)) (getC state) (getM state)
says
let state' = applyBackjump state in
InvariantUniq (getM state')

proof –
let ?l = getCl state
let ?bClause = getC state
let ?bLiteral = opposite ?l
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level (getM state)
let ?state'' = applyBackjump state

have clauseFalse (getC state) (elements (getM state))
    using (getConflictFlag state)
    using (InvariantCFalse (getConflictFlag state) (getM state) (getC state))
    unfolding InvariantCFalse-def
    by simp

have isUnitClause ?bClause ?bLiteral (elements ?prefix) and
    getM ?state'' = ?prefix @ [(?bLiteral, False)]
    using assms
    using applyBackjumpEffect[of state]
    by (auto simp add: Let-def)
thus ?thesis
    using (InvariantUniq (getM state))
    using isPrefixPrefixToLevel
    by (auto simp add: Let-def)
qed

lemma WatchInvariantsAfterApplyBackjump:
assumes
    InvariantConsistent (getM state)
    InvariantUniq (getM state)
    InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
    InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization \( (gF \text{ state}) \) \( (gWatch1 \text{ state}) \) \( (gWatch2 \text{ state}) \) \( (gM \text{ state}) \) \text{ and } \nabla
InvariantWatchListsContainOnlyClausesFromF \( (gWatchList \text{ state}) \) \( (gF \text{ state}) \) \text{ and } \nabla
InvariantWatchListsUniq \( (gWatchList \text{ state}) \) \text{ and } \nabla
InvariantWatchListsCharacterization \( (gWatchList \text{ state}) \) \( (gWatch1 \text{ state}) \) \( (gWatch2 \text{ state}) \) \( (gM \text{ state}) \)

\( \text{getConflictFlag} \text{ state} \)
\n\( \text{InvariantUniqC} \) \( (gC \text{ state}) \) \text{ and } \nabla
\( \text{InvariantCFalse} \) \( (gConflictFlag \text{ state}) \) \( (gM \text{ state}) \) \( (gC \text{ state}) \) \text{ and } \nabla
\( \text{InvariantCEntailed} \) \( (gConflictFlag \text{ state}) \) \( (gM \text{ state}) \) \text{ and } \nabla
\( \text{InvariantCllCharacterization} \) \( (gCl \text{ state}) \) \( (gC \text{ state}) \) \( (gM \text{ state}) \) \text{ and } \nabla
\( \text{InvariantCllCurrentLevel} \) \( (gCl \text{ state}) \) \( (gM \text{ state}) \)

\( \text{isUIP} \) \( (\text{opposite} \ (gCl \text{ state})) \) \( (gC \text{ state}) \) \( (gM \text{ state}) \)

\( \text{currentLevel} \) \( (gM \text{ state}) > 0 \)

\textbf{shows}

\( \text{let state}' = (\text{applyBackjump \ state}) \text{ in} \)
\( \text{InvariantWatchesEl} \) \( (gF \text{ state}') \) \( (gWatch1 \text{ state}') \) \( (gWatch2 \text{ state}') \) \text{ and } \nabla
\( \text{InvariantWatchesDiffer} \) \( (gF \text{ state}') \) \( (gWatch1 \text{ state}') \) \( (gWatch2 \text{ state}') \) \text{ and } \nabla
\( \text{InvariantWatchCharacterization} \) \( (gF \text{ state}') \) \( (gWatch1 \text{ state}') \) \( (gWatch2 \text{ state}') \) \( (gM \text{ state}') \) \text{ and } \nabla
\( \text{InvariantWatchListsContainOnlyClausesFromF} \) \( (gWatchList \text{ state}') \) \( (gF \text{ state}') \) \text{ and } \nabla
\( \text{InvariantWatchListsUniq} \) \( (gWatchList \text{ state}') \) \text{ and } \nabla
\( \text{InvariantWatchListsCharacterization} \) \( (gWatchList \text{ state}') \) \( (gWatch1 \text{ state}') \) \( (gWatch2 \text{ state}') \) \( (gM \text{ state}') \)

\( \text{is let state}' = (\text{applyBackjump \ state}) \text{ in } ?\text{inv state}' \)

\textbf{proof—}

\( \text{let } ?l = gCl \text{ state} \)
\( \text{let } ?\text{level} = \text{getBackjumpLevel \ state} \)
\( \text{let } \text{?prefix} = \text{prefixToLevel} \ ?\text{level} \ (gM \text{ state}) \)
\( \text{let } ?\text{state}' = \text{state}() \text{ getConflictFlag} := \text{False}, \text{ getQ} := [], \text{ getM} := \text{?prefix} \)
\( \text{let } ?\text{state}'' = \text{setReason} \ (\text{opposite} \ (gCl \text{ state})) \ (\text{length} \ (gF \text{ state}) \ - \ 1) \ ?\text{state}' \)
\( \text{let } ?\text{state}0 = \text{assertLiteral} \ (\text{opposite} \ (gCl \text{ state})) \text{ False } ?\text{state}'' \)

\textbf{have}

\( gF \ ?\text{state}' = gF \text{ state} \text{ getWatchList } ?\text{state}' = gWatchList \text{ state} \)
\( gWatch1 \ ?\text{state}' = gWatch1 \text{ state} \text{ getWatch2 } ?\text{state}' = \text{ getWatch2 state} \)
unfolding setReason-def
by (auto simp add: Let-def)
moreover
have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getM ?state')
using assms
using InvariantWatchCharacterizationInBackjumpPrefix[of state]
unfolding setReason-def
by (simp add: Let-def)
moreover
have InvariantConsistent (?prefix @ [(opposite ?l, False)])
using assms
using InvariantConsistentAfterApplyBackjump[of state F0]
using assertLiteralEffect
unfolding applyBackjump-def
unfolding setReason-def
by (auto simp add: Let-def split: split-if-asm)
moreover
have InvariantUniq (?prefix @ [(opposite ?l, False)])
using assms
using InvariantUniqAfterApplyBackjump[of state F0]
using assertLiteralEffect
unfolding applyBackjump-def
unfolding setReason-def
by (auto simp add: Let-def split: split-if-asm)
ultimately
show ?thesis
using assms
using WatchInvariantsAfterAssertLiteral[of ?state'' opposite ?l False]
using WatchInvariantsAfterAssertLiteral[of ?state' opposite ?l False]
using InvariantWatchCharacterizationAfterAssertLiteral[of ?state'' opposite ?l False]
using InvariantWatchCharacterizationAfterAssertLiteral[of ?state' opposite ?l False]
unfolding applyBackjump-def
unfolding setReason-def
by (auto simp add: Let-def)
qed

lemma InvariantUniqQAAfterApplyBackjump:
assumes
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
shows
let state' = applyBackjump state in
InvariantUniqQ (getQ state')
proof
let ?l = getCl state
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level (getM state)
let ?state' = state( getConflictFlag := False, getQ := [], getM := ?prefix )
let ?state'' = setReason (opposite (getCl state)) (length (getF state) – 1) ?state'

show ?thesis
using assms
unfolding applyBackjump-def
using InvariantUniqQAAfterAssertLiteral[of ?state’ opposite ?? False]
using InvariantUniqQAAfterAssertLiteral[of ?state’’ opposite ?? False]
unfolding InvariantUniqQ-def
unfolding setReason-def
by (auto simp add: Let-def)

qed

lemma invariantQCharacterizationAfterApplyBackjump-1:
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state) and
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state) and
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state) and

InvariantUniqC (getC state)
getC state = [opposite (getCl state)]
InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))
InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))

getConflictFlag state
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
InvariantCEntailed \((\text{getConflictFlag state})\) \(F_{0}\) \((\text{getC state})\) and

InvariantClCharacterization \((\text{getCl state})\) \((\text{getC state})\) \((\text{getM state})\) and

InvariantClCurrentLevel \((\text{getCl state})\) \((\text{getM state})\) currentLevel \((\text{getM state})\) > 0

isUIP \((\text{opposite (getCl state)})\) \((\text{getC state})\) \((\text{getM state})\)

shows

let state"" = (applyBackjump state) in

InvariantQCharacterization \((\text{getConflictFlag state''})\) \((\text{getQ state''})\) \((\text{getF state'')}\) \((\text{getM state''})\)

proof –

let ?l = getCl state
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level \((\text{getM state})\)
let ?state' = state[] getConflictFlag := False, getQ := [], getM := ?prefix []
let ?state'' = setReason \((\text{opposite (getCl state)})\) \((\text{length (getF state)} - 1)\) ?state'

let ?state'1 = assertLiteral \((\text{opposite ?l})\) False ?state'
let ?state''1 = assertLiteral \((\text{opposite ?l})\) False ?state''

have ?level < elementLevel ?l \((\text{getM state})\)
using assms
using isMinimalBackjumpLevelGetBackjumpLevel[of state]
unfolding isMinimalBackjumpLevel-def
unfolding isBackjumpLevel-def
by (simp add: Let-def)
hence ?level < currentLevel \((\text{getM state})\)
using elementLevelLeqCurrentLevel[of ?l getM state]
by simp
hence InvariantQCharacterization \((\text{getConflictFlag ?state'})\) \((\text{getQ ?state'})\) \((\text{getF ?state'})\) \((\text{getM ?state'})\)
InvariantConflictFlagCharacterization \((\text{getConflictFlag ?state'})\) \((\text{getF ?state'})\) \((\text{getM ?state'})\)
unfolding InvariantQCharacterization-def
unfolding InvariantConflictFlagCharacterization-def
using \((\text{InvariantNoDecisionsWhenConflict (getF state) (getM state)})\)
\((\text{currentLevel (getM state)})\)
using \((\text{InvariantNoDecisionsWhenUnit (getF state) (getM state)})\)
\((\text{currentLevel (getM state)})\)
unfolding InvariantNoDecisionsWhenConflict-def
unfolding InvariantNoDecisionsWhenUnit-def
unfolding applyBackjump-def
by (auto simp add: Let-def set-conv-nth)
moreover
have InvariantConsistent (?prefix @ [(opposite ?l, False)])
  using assms
  using InvariantConsistentAfterApplyBackjump[of state F0]
  using assertLiteralEffect
  unfolding applyBackjump-def
  unfolding setReason-def
  by (auto simp add: Let-def split: split-if-asm)
moreover
have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state')
  (getWatch2 ?state') (getM ?state')
  using InvariantWatchCharacterizationInBackjumpPrefix[of state]
  using assms
  by (simp add: Let-def)
moreover
have ¬ opposite ?l el (getQ ?state'1) ¬ opposite ?l el (getQ ?state''1)
  using assertedLiteralIsNotUnit[of ?state' opposite ?l False]
  using assertedLiteralIsNotUnit[of ?state'' opposite ?l False]
  using InvariantQCharacterization (getConflictFlag ?state') (getQ ?state')
  (getM ?state')
  using InvariantConsistent (?prefix @ [(opposite ?l, False)]);
  using InvariantWatchCharacterization (getF ?state') (getWatch1 ?state')
  (getWatch2 ?state') (getM ?state')
  unfolding applyBackjump-def
  unfolding setReason-def
  by (auto simp add: Let-def split: split-if-asm)
hence removeAll (opposite ?l) (getQ ?state'1) = getQ ?state'1
  removeAll (opposite ?l) (getQ ?state''1) = getQ ?state''1
  unfolding removeAll-id[of opposite ?l getQ ?state'1]
  unfolding removeAll-id[of opposite ?l getQ ?state''1]
  by auto
ultimately
show ?thesis
  using assms
  using InvariantWatchCharacterizationInBackjumpPrefix[of state]
  using InvariantQCharacterizationAfterAssertLiteral[of ?state' opposite ?l False]
  using InvariantQCharacterizationAfterAssertLiteral[of ?state'' opposite ?l False]
  unfolding applyBackjump-def
  unfolding setReason-def
  by (auto simp add: Let-def)
qed

lemma invariantQCharacterizationAfterApplyBackjump-2:
  fixes state::State
  assumes
\textbf{InvariantConsistent (getM state)}

\textbf{InvariantUniq (getM state)}

\textbf{InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and}

\textbf{InvariantWatchListsUniq (getWatchList state) and}

\textbf{InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and}

\textbf{InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and}

\textbf{InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and}

\textbf{InvariantWatchCharacterization (getWatch1 state) (getWatch2 state) (getM state) and}

\textbf{InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state) and}

\textbf{InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state) and}

\textbf{InvariantUniqC (getC state)}

\textbf{getC state} \not= \text{opposite (getCl state)}

\textbf{InvariantNoDecisionsWhenUnit (butlast (getF state)) (getM state) (currentLevel (getM state))}

\textbf{InvariantNoDecisionsWhenConflict (butlast (getF state)) (getM state) (currentLevel (getM state))}

\textbf{getF state} = []

\textbf{last (getF state) = getC state}

\textbf{getConflictFlag state}

\textbf{InvariantCFalse (getConflictFlag state) (getM state) (getC state) and}

\textbf{InvariantCEntailed (getConflictFlag state) F0 (getC state) and}

\textbf{InvariantClCharacterization (getCl state) (getC state) (getM state) and}

\textbf{InvariantClCurrentLevel (getCl state) (getM state) and}

\textbf{currentLevel (getM state) > 0}

\textbf{isUIP (opposite (getCl state)) (getC state) (getM state)}

\textbf{shows}

\textbf{let state" = (applyBackjump state) in}

\textbf{InvariantQCharacterization (getConflictFlag state") (getQ state") (getF state") (getM state")}

\textbf{proof=}

\textbf{let } ?l = getCl state

\textbf{let } ?level = getBackjumpLevel state

\textbf{let } ?prefix = prefixToLevel ?level (getM state)

\textbf{let } ?state" = state[] getConflictFlag := False, getQ := [], getM :=

650
let \(\text{state}' = \text{setReason}(\text{opposite}(\text{getCl \ state})) \ (\text{length}(\text{getF \ state}) - 1) \ \text{?state}'\)

have \(?\text{level} < \text{elementLevel} \ ?l \ (\text{getM \ state})\)
using \(\text{assms}\)
using \(\text{isMinimalBackjumpLevelGetBackjumpLevel[of \ state]}\)
unfolding \(\text{isMinimalBackjumpLevel-def}\)
unfolding \(\text{isBackjumpLevel-def}\)
by \((\text{simp add: Let-def})\)

hence \(?\text{level} < \text{currentLevel} \ (\text{getM \ state})\)
using \(\text{elementLevelLeqCurrentLevel[of \ ?l \ getM \ state]}\)
by \(\text{simp}\)

have \(\text{isUnitClause} \ (\text{last}(\text{getF \ state}))(\text{opposite} \ ?l)(\text{elements} \ ?\text{prefix})\)
using \(\text{last}(\text{getF \ state}) = \text{getC \ state}\)
using \(\text{isMinimalBackjumpLevelGetBackjumpLevel[of \ state]}\)
using \(\text{InvariantUniq} \ (\text{getM \ state})\)
using \(\text{InvariantConsistent} \ (\text{getM \ state})\)
using \(\text{getConflictFlag \ state}\)
using \(\text{InvariantUniqC} \ (\text{getC \ state})\)
using \(\text{InvariantCFalse} \ (\text{getConflictFlag \ state}) \ (\text{getM \ state}) \ (\text{getC \ state})\)
using \(\text{isBackjumpLevelEnsuresIsUnitInPrefix[of \ getM \ state \ getC \ state \ getBackjumpLevel \ state \ opposite \ ?l]}\)
using \(\text{InvariantClCharacterization} \ (\text{getCl \ state}) \ (\text{getC \ state}) \ (\text{getM \ state})\)
using \(\text{InvariantCllCharacterization} \ (\text{getCl \ state}) \ (\text{getCll \ state}) \ (\text{getC \ state}) \ (\text{getM \ state})\)
using \(\text{InvariantClCurrentLevel} \ (\text{getCl \ state}) \ (\text{getM \ state})\)
using \(\text{currentLevel} \ (\text{getM \ state}) > 0\)
unfolding \(\text{isMinimalBackjumpLevel-def}\)
unfolding \(\text{InvariantUniq-def}\)
unfolding \(\text{InvariantConsistent-def}\)
unfolding \(\text{InvariantCFalse-def}\)
by \((\text{simp add: Let-def})\)

hence \(\neg \text{clauseFalse} \ (\text{last}(\text{getF \ state}))(\text{elements} \ ?\text{prefix})\)
unfolding \(\text{isUnitClause-def}\)
by \((\text{auto simp add: clauseFalseIffAllLiteralsAreFalse})\)

have \(\text{InvariantConsistent} \ (?\text{prefix} @ ([\text{opposite} \ ?l, \text{False}]))\)
using \(\text{assms}\)
using \(\text{InvariantConsistentAfterApplyBackjump[of \ state \ F0]}\)
using \(\text{assertLiteralEffect}\)
unfolding \(\text{applyBackjump-def}\)
unfolding \(\text{setReason-def}\)
by \((\text{auto simp add: Let-def split: split-if-asm})\)
have InvariantUniq (?prefix @ [(opposite ?l, False)])
  using assms
  using InvariantUniqAfterApplyBackjump[of state F0]
  unfolding assertLiteralEffect
  unfolding applyBackjump-def
  unfolding setReason-def
  by (auto simp add: Let-def split: split-if-asm)

let ?state'1 = ?state' @ [getQ := getQ ?state' @ [opposite ?l]]
let ?state'2 = assertLiteral (opposite ?l) False ?state'1

let ?state''1 = ?state'' @ [getQ := getQ ?state'' @ [opposite ?l]]
let ?state''2 = assertLiteral (opposite ?l) False ?state''1

have InvariantQCharacterization (getConflictFlag ?state') ((getQ ?state' @ [opposite ?l]) (getF ?state') (getM ?state'))
  proof
    have ∀ l c el (butlast (getF state)) −→ ¬isUnitClause c l (elements (getM state'))
      using :InvariantNoDecisionsWhenUnit (butlast (getF state)) (getM state) (currentLevel (getM state))
      unfolding :InvariantNoDecisionsWhenUnit-def
      by simp
    have ∀ l. (∃ c. c el (getF state) ∧ isUnitClause c l (elements (getM state'))) = (l = opposite ?l)
      proof
        fix l
        show (∃ c. c el (getF state) ∧ isUnitClause c l (elements (getM state'))) = (l = opposite ?l) (is ?lhs = ?rhs)
        proof
          assume ?lhs
          then obtain c::Clause
            where c el (getF state) and isUnitClause c l (elements ?prefix)
            by auto
          show ?rhs
            proof (cases c el (butlast (getF state)))
              case True
              thus ?thesis
                using ∀ l c el (butlast (getF state)) −→ ¬isUnitClause c l (elements (getM state'))
                unfolding :isUnitClause c l (elements ?prefix)
                by auto
            next
              case False
              from (getF state ≠ [])
              have butlast (getF state) @ [last (getF state)] = getF state
using append-butlast-last-id[of getF state]
by simp
hence getF state = butlast (getF state) @ [last (getF state)]
by (rule sym)
with \(c \in \text{getF state}\)
have \(c \in \text{butlast (getF state)} \lor c \in \text{[last (getF state)]}\)
using set-append[of butlast (getF state) [last (getF state)]]
by auto
hence \(c = \text{last (getF state)}\)
using \((\sim c \in \text{[butlast (getF state)]})\)
by simp
thus \(?\)thesis
using (isUnitClause (last (getF state)) (opposite ?l) (elements ?prefix))
using (isUnitClause c l (elements ?prefix))
unfolding isUnitClause-def
by auto

done

next
from \(\text{getF state} \neq []\)
have last (getF state) el (getF state)
by auto
assume ?rhs
thus ?rhs
using (isUnitClause (last (getF state)) (opposite ?l) (elements ?prefix))
using (last (getF state) el (getF state))
by auto

done

hence InvariantQCharacterization (getConflictFlag ?state'1) (getQ ?state'1) (getF ?state'1) (getM ?state'1)
by simp
hence InvariantQCharacterization (getConflictFlag ?state''1) (getQ ?state''1) (getF ?state''1) (getM ?state''1)
unfolding setReason-def
by simp

have InvariantWatchCharacterization (getF ?state'1) (getWatch1 ?state'1) (getWatch2 ?state'1) (getM ?state'1)
using InvariantWatchCharacterizationInBackjumpPrefix[of state]
using assms
by (simp add: Let-def)


qed
unfolding setReason-def
by simp

have InvariantWatchCharacterization (getF ?state') (getWatch1 ?state') (getWatch2 ?state') (getM ?state')
  using InvariantWatchCharacterizationInBackjumpPrefix[of state]
  using assms
by (simp add: Let-def)
hence InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getM ?state'')
unfolding setReason-def
by simp

have InvariantConflictFlagCharacterization (getConflictFlag ?state'1) (getF ?state'1) (getM ?state'1)
proof -
  |
  |
fix c::Clause
assume c el (getF state)
have ¬ clauseFalse c (elements ?prefix)
proof (cases c el (butlast (getF state)))
case True
  thus ?thesis
    using InvariantNoDecisionsWhenConflict (butlast (getF state)) (getM state) (currentLevel (getM state))
    using (?level < currentLevel (getM state))
    unfolding InvariantNoDecisionsWhenConflict-def
    by (simp add: formulaFalseIffContainsFalseClause)
next
case False
from getF state ≠ []
have butlast (getF state) @ [last (getF state)] = getF state
  using append-butlast-last-id[of getF state]
by simp
hence getF state = butlast (getF state) @ [last (getF state)]
  by (rule sym)
with ⟨c el getF state⟩
have c el butlast (getF state) ∨ c el [last (getF state)]
  using set-append[of butlast (getF state) [last (getF state)]]
by auto
hence c = last (getF state)
  using ¬ c el (butlast (getF state))
by simp
thus ?thesis
  using ¬ clauseFalse (last (getF state)) (elements ?prefix)
  by simp
qed
} thus ?thesis
unfolding InvariantConflictFlagCharacterization-def
by (simp add: formulaFalseIffContainsFalseClause)
qed

hence InvariantConflictFlagCharacterization (getConflictFlag \?state'"1) (getF \?state'"1) (getM \?state'"1)

unfolding setReason-def

by simp

have InvariantQCharacterization (getConflictFlag \?state'2) (removeAll (opposite \?l) (getQ \?state'2)) (getF \?state'2) (getM \?state'2)

using assms

using (InvariantConsistent (?prefix @ [(opposite \?l, False)]))

using (InvariantUniq (?prefix @ [(opposite \?l, False)]))

using (InvariantConflictFlagCharacterization (getConflictFlag \?state'1) (getF \?state'1) (getM \?state'1))

using (InvariantWatchCharacterization (getF \?state'1) (getWatch1 \?state'1) (getWatch2 \?state'1) (getM \?state'1))

using (InvariantQCharacterization (getConflictFlag \?state'1) (getQ \?state'1) (getM \?state'1))

using (InvariantQCharacterizationAfterAssertLiteral[of \?state'1 opposite \?l False])

by (simp add: Let-def)

have InvariantQCharacterization (getConflictFlag \?state'"2) (removeAll (opposite \?l) (getQ \?state'"2)) (getF \?state'"2) (getM \?state'"2)

using assms

using (InvariantConsistent (?prefix @ [(opposite \?l, False)]))

using (InvariantUniq (?prefix @ [(opposite \?l, False)]))

using (InvariantConflictFlagCharacterization (getConflictFlag \?state'"1) (getF \?state'"1) (getM \?state'"1))

using (InvariantWatchCharacterization (getF \?state'"1) (getWatch1 \?state'"1) (getWatch2 \?state'"1) (getM \?state'"1))

using (InvariantQCharacterization (getConflictFlag \?state'"1) (getQ \?state'"1) (getM \?state'"1))

using (InvariantQCharacterizationAfterAssertLiteral[of \?state'"1 opposite \?l False])

unfolding setReason-def

by (simp add: Let-def)

let \?stateB = applyBackjump state

show \?thesis

proof (cases getBackjumpLevel state > 0)

case False

let \?state01 = state( getConflictFlag := False, getM := \?prefix )

have InvariantWatchesEl (getF \?state01) (getWatch1 \?state01) (getWatch2 \?state01)

using (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state))

unfolding InvariantWatchesEl-def
by auto

have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state01) (getF ?state01)
  using InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
by auto

have assertLiteral (opposite ?l) False (state (getConflictFlag := False, getQ := [], getM := ?prefix )) =
  assertLiteral (opposite ?l) False (state (getConflictFlag := False, getQ := [], getM := ?prefix ),
  getQ := [])
  state (getConflictFlag := False, getM := ?prefix, getQ := [])
λ x. assertLiteral (opposite ?l) False x
by simp

hence getConflictFlag ?stateB = getConflictFlag ?state'2
  getF ?stateB = getF ?state'2
  getM ?stateB = getM ?state'2
unfolding applyBackjump-def
  using InvariantWatchesEl (getF ?state01) (getWatch1 ?state01)
  (getWatch2 ?state01);
  using InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state01) (getF ?state01);
  using (¬ getBackjumpLevel state > 0)
by (auto simp add: Let-def)

have set (getQ ?stateB) = set (removeAll (opposite ?l) (getQ ?state'2))
proof−
  have set (getQ ?stateB) = set (getQ ?state'2) - {opposite ?l}
proof−
  let ?ulSet = { ul. (∃ uc. el (getF ?state'1) ∧ ?l el uc ∧
  isUnitClause uc ul ((elements (getM ?state'1)) @ [opposite ?l])) }
  have set (getQ ?state'2) = {opposite ?l} ∪ ?ulSet
  using assertLiteralQEffect[of ?state'1 opposite ?l False]
  using assms
  using InvariantConsistent (?prefix @ [(opposite ?l, False)]);
  using InvariantUniq (?prefix @ [(opposite ?l, False)]);
  using InvariantWatchCharacterization (getF ?state'1) (getWatch1 ?state'1) (getWatch2 ?state'1) (getM ?state'1);
by (simp add: Let-def)
moreover
have \( \text{set} \ (\text{getQ} \ ?\text{stateB}) = \ ?\text{ulSet} \)
  using \text{assertLiteralQEffect}[\text{of} \ ?\text{state} \ \text{opposite} \ ?l \ False]
  using \text{assms}
  using \langle \text{InvariantConsistent} \ (\text{prefix} @ [(\text{opposite} \ ?l, \ False)]) \rangle
  using \langle \text{InvariantUniq} \ (\text{prefix} @ [(\text{opposite} \ ?l, \ False)]) \rangle
using \langle \text{InvariantWatchCharacterization} \ (\text{getF} \ ?\text{state}') \ (\text{getWatch1} \ ?\text{state}') \ (\text{getWatch2} \ ?\text{state}') \ (\text{getM} \ ?\text{state}') \rangle
  by (\text{simp add: Let-def})
moreover
have \( \neg \ (\text{opposite} \ ?l) \in \ ?\text{ulSet} \)
  using \text{assertedLiteralIsNotUnl}[\text{of} \ ?\text{state}']
  using \text{assms}
  using \langle \text{InvariantConsistent} \ (\text{prefix} @ [(\text{opposite} \ ?l, \ False)]) \rangle
  using \langle \text{InvariantUniq} \ (\text{prefix} @ [(\text{opposite} \ ?l, \ False)]) \rangle
using \langle \text{InvariantWatchCharacterization} \ (\text{getF} \ ?\text{state}') \ (\text{getWatch1} \ ?\text{state}') \ (\text{getWatch2} \ ?\text{state}') \ (\text{getM} \ ?\text{state}') \rangle
  by (\text{simp add: Let-def})
ultimately
show ?thesis
  by simp
qed
thus ?thesis
  by simp
qed

show ?thesis
  using \langle \text{InvariantQCharacterization} \ (\text{getConflictFlag} \ ?\text{state}') \ (\text{removeAll} \ (\text{opposite} \ ?l) \ (\text{getQ} \ ?\text{state}')) \ (\text{getF} \ ?\text{state}') \ (\text{getM} \ ?\text{state}')) \rangle
  by (\text{simp add: Let-def})
next
\text{case} True
let ?\text{state02} = \text{setReason} \ (\text{opposite} \ (\text{getCl} \ \text{state})) \ (\text{length} \ (\text{getF} \ \text{state}) - 1)
  \text{state}[(\text{getConflictFlag} := \ False, \ \text{getM} := \ ?\text{prefix})]
have \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state02}) \ (\text{getWatch1} \ ?\text{state02}) \ (\text{getWatch2} \ ?\text{state02})
  using \text{InvariantWatchesEl} \ (\text{getF} \ ?\text{state}) \ (\text{getWatch1} \ ?\text{state}) \ (\text{getWatch2} \ ?\text{state})

657
unfolding InvariantWatchesEl-def
unfolding setReason-def
by auto

have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state02) (getF ?state02)
using InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state):
unfolding InvariantWatchListsContainOnlyClausesFromF-def
unfolding setReason-def
by auto

let ?stateTmp′ = assertLiteral (opposite (getCl state)) False
(setReason (opposite (getCl state))) (length (getF state) − 1)
state ⟨(getConflictFlag := False,
getM := prefixToLevel (getBackjumpLevel state) (getM state),
geQ := []⟩
)

let ?stateTmp′′ = assertLiteral (opposite (getCl state)) False
(setReason (opposite (getCl state))) (length (getF state) − 1)
state ⟨(getConflictFlag := False,
getM := prefixToLevel (getBackjumpLevel state) (getM state),
geQ := [opposite (getCl state)]⟩
)

have getM ?stateTmp′ = getM ?stateTmp′′
getF ?stateTmp′ = getF ?stateTmp′′
getSATFlag ?stateTmp′ = getSATFlag ?stateTmp′′
getConflictFlag ?stateTmp′ = getConflictFlag ?stateTmp′′
using AssertLiteralStartQIrelevant[of ?state02 opposite ?l False [] [opposite ?l]]
using InvariantWatchesEl (getF ?state02) (getWatch1 ?state02)
(getWatch2 ?state02)
using InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state02) (getF ?state02)
by (auto simp add: Let-def)
moreover
have ?stateB = ?stateTmp′
using (getBackjumpLevel state > 0)
using arg-cong[of state ⟨
    getConflictFlag := False,
geQ := [],
geM := ?prefix,
geReason := getReason state (opposite ?l →
length (getF state) − 1)⟩
\[
\begin{align*}
\text{state} & \quad \text{state} \\
\text{getReason} & := \text{getReason state (opposite \ ?l \mapsto length (getF state) - 1)}, \\
\text{getConflictFlag} & := \text{False}, \\
\text{getM} & := \text{prefixToLevel (getBackjumpLevel state) (getM state)}, \\
\text{getQ} & := [] \\
\text{λ} x. \text{assertLiteral (opposite \ ?l) \ False} \\
\text{unfolding \ applyBackjump-def} \\
\text{unfolding \ setReason-def} \\
\text{by (auto simp add: Let-def)} \\
\text{moreover} \\
\text{have \ ?state\tmp" = ?state\''2} \\
\text{unfolding \ setReason-def} \\
\text{using \ arg-cong[of state \ \getReason \ state} \text{getReason state (opposite \ ?l \mapsto length (getF state) - 1)}, \\
\text{getConflictFlag} := \text{False}, \\
\text{getM} := ?\prefix, \text{getQ} := [\text{opposite \ ?l}] \\
\text{getQ} := [\text{opposite \ ?l}] \\
\text{by simp} \\
\text{ultimately} \\
\text{have \ getConflictFlag \ ?state\B \ = \ getConflictFlag \ ?state\''2} \\
\text{getF \ ?state\B \ = \ getF \ ?state\''2} \\
\text{getM \ ?state\B \ = \ getM \ ?state\''2} \\
\text{by \ auto} \\
\text{have \ set \ (getQ \ ?state\B) \ = \ set \ (removeAll \ (opposite \ ?l) \ (getQ \ ?state\''2))} \\
\text{proof} \\
\text{have \ set \ (getQ \ ?state\B) \ = \ set (getQ \ ?state\''2) - \ {opposite \ ?l}} \\
\text{proof} \\
\text{let \ ?ulSet = \ { \ ul \ \exists \ uc. \ uc \ el \ (getF \ ?state\''1) \ \land \\
\text{\ ?l \ el \ uc} \ \land \\
\text{\ isUnitClause uc \ ul \ ((elements (getM \ ?state\''1)) \ @ \ [opposite \ ?l])}} \\
\text{have \ set \ (getQ \ ?state\''2) \ = \ {opposite \ ?l} \cup \ ?ulSet} \\
\text{using \ assertLiteralQEffect[of \ ?state\''1 \ opposite \ ?l \ False]} \\
\text{using \ assms} \\
\text{using \ \langle InvariantConsistent \ (?prefix \ @ \ [(opposite \ ?l, \ False)]) \rangle} \\
\text{using \ \langle InvariantUniq \ (?prefix \ @ \ [(opposite \ ?l, \ False)]) \rangle} \\
\text{using \ \langle InvariantWatchCharacterization \ (getF \ ?state\''1) \rangle} \\
\end{align*}
\]
(getWatch1 ?state"1") (getWatch2 ?state"1") (getM ?state"1")
  unfolding setReason-def
  by (simp add: Let-def)
moreover
  have set (getQ ?stateB) = ?ulSet
    using assertLiteralQEffect[of ?state" opposite ?l False]
    using assms
  using (InvariantConsistent (?prefix @ [(opposite ?l, False)]))
  using (InvariantUniq (?prefix @ [(opposite ?l, False)]))
  using (InvariantWatchCharacterization (getF ?state") (getWatch1 ?state") (getWatch2 ?state") (getM ?state")
    using (getBackjumpLevel state > 0)
    unfolding applyBackjump-def
    unfolding setReason-def
    by (simp add: Let-def)
moreover
  have ¬ (opposite ?l) ∈ ?ulSet
    using assertedLiteralIsNotUnit[of ?state" opposite ?l False]
    using assms
  using (InvariantConsistent (?prefix @ [(opposite ?l, False)]))
  using (InvariantUniq (?prefix @ [(opposite ?l, False)]))
  using (InvariantWatchCharacterization (getF ?state") (getWatch1 ?state") (getWatch2 ?state") (getM ?state")
    using (set (getQ ?stateB) = ?ulSet)
    using (getBackjumpLevel state > 0)
    unfolding applyBackjump-def
    unfolding setReason-def
    by (simp add: Let-def)
ultimately
  show ?thesis
  by simp
qed
thus ?thesis
by simp
qed

show ?thesis
  using (InvariantQCharacterization (getConflictFlag ?state"2")
  (removeAll (opposite ?l) (getQ ?state"2")) (getF ?state"2") (getM ?state"2")
    using (set (getQ ?stateB) = set (removeAll (opposite ?l) (getQ
 ?state"2")))
    using (getConflictFlag ?stateB = getConflictFlag ?state"2")
    using (getF ?stateB = getF ?state"2")
    using (getM ?stateB = getM ?state"2")
    unfolding InvariantQCharacterization-def
    by (simp add: Let-def)
qed
qed
lemma InvariantConflictFlagCharacterizationAfterApplyBackjump-1:
assumes
*InvariantConsistent* (*getM* state)
*InvariantUniq* (*getM* state)
*InvariantWatchListsContainOnlyClausesFromF* (*getWatchList* state)
(*getF* state) and
*InvariantWatchListsUniq* (*getWatchList* state) and
*InvariantWatchListsCharacterization* (*getWatchList* state) (*getWatch1* state) (*getWatch2* state)
*InvariantWatchesEl* (*getF* state) (*getWatch1* state) (*getWatch2* state) and
*InvariantWatchesDiffer* (*getF* state) (*getWatch1* state) (*getWatch2* state) (*getM* state) and
*InvariantWatchCharacterization* (*getF* state) (*getWatch1* state) (*getWatch2* state) (*getM* state)
*InvariantUniqC* (*getC* state)
*getC* state = [opposite (*getCl* state)]
*InvariantNoDecisionsWhenConflict* (*getF* state) (*getM* state) (*currentLevel* (*getM* state))

*getConflictFlag* state
*InvariantCFalse* (*getConflictFlag* state) (*getM* state) (*getC* state) and
*InvariantCEntailed* (*getConflictFlag* state) *F0* (*getC* state) and
*InvariantClCharacterization* (*getCl* state) (*getM* state) (*getC* state) (*getM* state)
*InvariantClCurrentLevel* (*getCl* state) (*getM* state)
*currentLevel* (*getM* state) > 0
*isUIP* (opposite (*getCl* state)) (*getC* state) (*getM* state)

shows
let state’ = (applyBackjump state) in
*InvariantConflictFlagCharacterization* (*getConflictFlag* state’) (*getF* state’) (*getM* state’)

proof–
let ?l = *getCl* state
let ?level = *getBackjumpLevel* state
let ?prefix = prefixToLevel ?level (*getM* state)
let ?state’’ = setReason (opposite ?l) (length (*getF* state) - 1) ?state’
let ?stateB = applyBackjump state

have ?level < *elementLevel* ?l (*getM* state)
using assms
using isMinimalBackjumpLevel-getBackjumpLevel[of state]
unfolding isMinimalBackjumpLevel-def
unfolding isBackjumpLevel-def
by (simp add: Let-def)

hence ?level < currentLevel (getM state)
using elementLevelLeqCurrentLevel[of ?l getM state]
by simp

hence InvariantConflictFlagCharacterization (getConflictFlag ?state) (getF ?state) (getM ?state)
using (InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state)))
unfolding InvariantNoDecisionsWhenConflict-def
unfolding InvariantConflictFlagCharacterization-def
by simp

moreover
have InvariantConsistent (?prefix @ [(opposite ?l, False)])
using assms
using InvariantConsistentAfterApplyBackjump[of state F0]
using assertLiteralEffect
unfolding applyBackjump-def
unfolding setReason-def
by (auto simp add: Let-def split: split-if-asm)

ultimately
show ?thesis
using InvariantConflictFlagCharacterizationAfterAssertLiteral[of ?state]
using InvariantConflictFlagCharacterizationAfterAssertLiteral[of ?state]
using InvariantWatchCharacterizationInBackjumpPrefix[of state]
using assms
unfolding applyBackjump-def
unfolding setReason-def
using assertLiteralEffect
by (auto simp add: Let-def)

qed

lemma InvariantConflictFlagCharacterizationAfterApplyBackjump-2:
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
\begin{itemize}
  \item \textbf{InvariantWatchCharacterization} \( (\text{getF state}) \) \( (\text{getWatch1 state}) \) \( (\text{getWatch2 state}) \) \( (\text{getM state}) \) and
  \begin{align*}
    &\text{InvariantUniqC} \ (\text{getC state}) \\
    &\text{getC state} \neq \{\text{opposite (getC state)}\} \\
    &\text{InvariantNoDecisionsWhenConflict} \ (\text{butlast (getF state)}) \ (\text{getM state}) \\
    &\text{currentLevel} \ (\text{getM state}) \\
    &\text{getF state} \neq [] \text{ last (getF state)} = \text{getC state}
  \end{align*}

  \item \text{getConflictFlag state}
  \begin{align*}
    &\text{InvariantCFalse} \ (\text{getConflictFlag state}) \ (\text{getM state}) \ (\text{getC state}) \\
    \text{and}
    &\text{InvariantCEntailed} \ (\text{getConflictFlag state}) \ F0 \ (\text{getC state}) \ \text{and}
    &\text{InvariantClCharacterization} \ (\text{getC state}) \ (\text{getC state}) \ (\text{getM state}) \\
    \text{and}
    &\text{InvariantCllCharacterization} \ (\text{getC state}) \ (\text{getCl state}) \ (\text{getC state}) \ (\text{getM state})
  \end{align*}

  \item currentLevel \ (\text{getM state}) > 0
  \begin{align*}
    \text{isUIP} \ (\text{opposite (getC state)}) \ (\text{getC state}) \ (\text{getM state})
  \end{align*}

  \item shows
  \begin{align*}
    \text{let state'} = (\text{applyBackjump state}) \text{ in} \\
    \text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag state'}) \ (\text{getF state'}) \ (\text{getM state'})
  \end{align*}

  \item proof—
  \begin{align*}
    \text{let } ?l &= \text{getC state} \\
    \text{let } ?level &= \text{getBackjumpLevel state} \\
    \text{let } ?prefix &= \text{prefixToLevel} \ ?level \ (\text{getM state}) \\
    \text{let } ?state' &= \text{state[} \ (\text{getConflictFlag} := \text{False}, \ \text{getQ} := [], \ \text{getM} := \text{getF state}) \\
    \text{let } ?state'' &= \text{setReason (opposite ?l) (length (getF state) - 1)} \\
    \text{let } ?state' \\
    \text{let } ?stateB &= \text{applyBackjump state}
  \end{align*}

  \item have \( ?level < \text{elementLevel} \ ?l \ (\text{getM state}) \)
  \begin{align*}
    \text{using } \text{assms} \\
    \text{using } \text{isMinimalBackjumpLevelGetBackjumpLevel[of state]} \\
    \text{unfolding } \text{isMinimalBackjumpLevel-def} \\
    \text{unfolding } \text{isBackjumpLevel-def} \\
    \text{by } (\text{simp add: Let-def}) \\
    \text{hence } ?level < \text{currentLevel} \ (\text{getM state}) \\
    \text{using } \text{elementLevelLeqCurrentLevel[of ?l getM state]} \\
    \text{by } \text{simp}
  \end{align*}

  \item hence \text{InvariantConflictFlagCharacterization} \ (\text{getConflictFlag state'}) \\
  \begin{align*}
    \text{(butlast (getF state'))} \ (\text{getM state'}) \\
    \text{using } \text{InvariantNoDecisionsWhenConflict (butlast (getF state))}
  \end{align*}
\end{itemize}
(getM state) (currentLevel (getM state))

  unfolding InvariantNoDecisionsWhenConflict-def
  unfolding InvariantConflictFlagCharacterization-def
  by simp

moreover
  have isBackjumpLevel (getBackjumpLevel state) (opposite (getCl state)) (getC state) (getM state)
    using assms
    using isMinimalBackjumpLevelGetBackjumpLevel[of state]
    unfolding isMinimalBackjumpLevel-def
    by (simp add: Let-def)
  hence isUnitClause (last (getF state)) (opposite ?l) (elements ?prefix)
    using isBackjumpLevelEnsuresIsUnitInPrefix[of getM state getC state getBackjumpLevel state opposite ?l]
    using (InvariantUniq (getM state))
    using (InvariantConsistent (getM state))
    using (getConflictFlag state)
    using (InvariantCFalse (getConflictFlag state) (getM state) (getC state))
    using (last (getF state) = getC state)
    unfolding InvariantUniq-def
    unfolding InvariantConsistent-def
    unfolding InvariantCFalse-def
    by (simp add: Let-def)
  hence ¬ clauseFalse (last (getF state)) (elements ?prefix)
    unfolding isUnitClause-def
    by (auto simp add: clauseFalseIffAllLiteralsAreFalse)

moreover
  from (getF state ≠ [])
  have butlast (getF state) @ [last (getF state)] = getF state
    using append-butlast-last-id[of getF state]
    by simp
  hence getF state = butlast (getF state) @ [last (getF state)]
    by (rule sym)

ultimately
  have InvariantConflictFlagCharacterization (getConflictFlag ?state') (getF ?state') (getM ?state')
    using set-append[of butlast (getF state) [last (getF state)]]
    unfolding InvariantConflictFlagCharacterization-def
    by (auto simp add: formulaFalseIffContainsFalseClause)

moreover
  have InvariantConsistent (?prefix @ [(opposite ?l, False)])
    using assms
    using InvariantConsistentAfterApplyBackjump[of state F0]
    using assertLiteralEffect
    unfolding applyBackjump-def
    unfolding setReason-def
    by (auto simp add: Let-def split: split-if-asm)
ultimately show thesis
  using InvariantConflictFlagCharacterizationAfterAssertLiteral[of state]
  using InvariantConflictFlagCharacterizationAfterAssertLiteral[of state’]
  using InvariantWatchCharacterizationInBackjumpPrefix[of state]
  using assms
  using assertLiteralEffect
  unfolding applyBackjump-def
  unfolding setReason-def
  by (auto simp add: Let-def)
qed

lemma InvariantConflictClauseCharacterizationAfterApplyBackjump:
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
  (getF state) and
  InvariantWatchListsUniq (getWatchList state) and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
shows
  let state’ = applyBackjump state in
  InvariantConflictClauseCharacterization (getConflictFlag state’)
  (getConflictClause state’)
  (getF state’)
  (getM state’)
proof
  let ?l = getC1 state
  let ?level = getBackjumpLevel state
  let ?prefix = prefixToLevel ?level (getM state)
  let ?state’ = state[| getConflictFlag := False, getQ := [], getM := ?prefix |
  let ?state’’ = if 0 < ?level then setReason (opposite ?l) (length (getF state) – 1) ?state’ else ?state’

  have – getConflictFlag ?state’
    by simp
  hence InvariantConflictClauseCharacterization (getConflictFlag ?state’’)
    (getConflictClause ?state’’)
    (getF ?state’’)
    (getM ?state’’)
  unfolding InvariantConflictClauseCharacterization-def
  unfolding setReason-def
  by auto
moreover
  have getF ?state’’ = getF state
  getWatchList ?state’’ = getWatchList state
  getWatch1 ?state’’ = getWatch1 state
  getWatch2 ?state’’ = getWatch2 state
  unfolding setReason-def
  by auto

665
ultimately show thesis using assms
  using InvariantConflictClauseCharacterizationAfterAssertLiteral[of state’']
  unfolding applyBackjump-def
  by (simp only: Let-def)
qed

lemma InvariantGetReasonIsReasonAfterApplyBackjump:
  assumes
    InvariantConsistent (getM state)
    InvariantUniq (getM state)
    InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
    InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
    InvariantWatchListsUniq (getWatchList state) and
    InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
    getConflictFlag state
    InvariantUniqC (getC state)
    InvariantCFalse (getConflictFlag state) (getM state) (getC state)
    InvariantCEntailed (getConflictFlag state) F0 (getC state)
    InvariantClCharacterization (getCl state) (getC state) (getM state)
    InvariantClCharacterization (getCl state) (getC state) (getC state)
    (getM state)
    InvariantClCurrentLevel (getCl state) (getM state)
    isUIP (opposite (getCl state)) (getC state) (getM state)
    0 < currentLevel (getM state)
    InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
    getBackjumpLevel state > 0 \rightarrow getF state \neq [] \land \text{last} (getF state)
= getC state
shows
  let state’ = applyBackjump state in
  InvariantGetReasonIsReason (getReason state’) (getF state’) (getM state’)
  (set (getQ state’))

proof
  let ?l = getCl state
  let ?level = getBackjumpLevel state
  let ?prefix = prefixToLevel ?level (getM state)
  let ?state’ = state[ (getConflictFlag := False, getQ := [], getM := ?prefix )]
  let ?state'' = if 0 < ?level then setReason (opposite ?l) (length (getF state) - 1) ?state’ else ?state’
  let ?stateB = applyBackjump state
  have InvariantGetReasonIsReason (getReason ?state’) (getF ?state’) (getM ?state’)
  (set (getQ ?state’))
proof
{|fix l::Literal
assume *: l el (elements ?prefix) ∧ ¬ l el (decisions ?prefix) ∧
elementLevel l ?prefix > 0

  hence l el (elements (getM state)) ∧ ¬ l el (decisions (getM state)) ∧ elementLevel l (getM state) > 0
using (InvariantUniq (getM state))
unfolding InvariantUniq-def
using isPrefixPrefixToLevel[of ?level (getM state)]
using isPrefixElements[of ?prefix getM state]
using prefixIsSubset[of elements ?prefix elements (getM state)]
using markedElementsTrailMemPrefixAreMarkedElementsPrefix[of getM state ?prefix l]
using elementLevelPrefixElement[of l getBackjumpLevel state getM state]

by auto

with assms
obtain reason
  where reason < length (getF state) isReason (nth (getF state) reason) l (elements (getM state))
  getReason state l = Some reason
unfolding InvariantGetReasonIsReason-def
by auto

  hence ∃ reason. getReason state l = Some reason ∧
  reason < length (getF state) ∧
  isReason (nth (getF state) reason) l (elements ?prefix)
using isReasonHoldsInPrefix[of l elements ?prefix elements (getM state) nth (getF state) reason]
using isPrefixPrefixToLevel[of ?level (getM state)]
using isPrefixElements[of ?prefix getM state]
using *
by auto

|}

thus ?thesis
  unfolding InvariantGetReasonIsReason-def
  by auto
qed

let ?stateM = ?state'′ @ (getM := getM ?state'' @ [(opposite ?l, False)])

have **: getM ?stateM = ?prefix @ [(opposite ?l, False)]
getF ?stateM = getF state
getQ ?stateM = []
getWatchList ?stateM = getWatchList state

667
getWatch1 ?stateM = getWatch1 state
getWatch2 ?stateM = getWatch2 state

unfolding setReason-def
by auto

have InvariantGetReasonIsReason (getReason ?stateM) (getF ?stateM)
(getM ?stateM) (set (getQ ?stateM))
proof –
  { 
    fix l::Literal
    assume *: l el (elements (getM ?stateM)) \land \neg l el (decisions
    (getM ?stateM)) \land elementLevel l (getM ?stateM) > 0

    have isPrefix ?prefix (getM ?stateM)
      unfolding setReason-def
      unfolding isPrefix-def
      by auto

    have \exists reason. getReason ?stateM l = Some reason \land
      reason < length (getF ?stateM) \land
      isReason (nth (getF ?stateM) reason) l (elements
      (getM ?stateM))
      proof (cases l = opposite ?l)
        case False
        hence l el (elements ?prefix)
          using *
          using **
          by auto
        moreover
        hence \neg l el (decisions ?prefix)
          using elementLevelAppend[of l ?prefix [(opposite ?l, False)]]
          using (isPrefix ?prefix (getM ?stateM))
          using markedElementsPrefixAreMarkedElementsTrail[of ?prefix
          getM ?stateM l]
          using *
          using **
          by auto
        moreover
        have elementLevel l ?prefix = elementLevel l (getM ?stateM)
          using \{ l el (elements ?prefix)\}
          using *
          using **
          using elementLevelAppend[of l ?prefix [(opposite ?l, False)]]
          by auto
        hence elementLevel l ?prefix > 0
          using *
          by simp
        ultimately
        obtain reason

668
where reason < length (getF state)
   isReason nth (getF state) reason l (elements ?prefix)
getReason state l = Some reason
using :InvariantGetReasonIsReason (getReason ?state') (getF
?state') (getM ?state') (set (getQ ?state'))
   unfolding InvariantGetReasonIsReason-def
   by auto
moreover
have getReason ?stateM l = getReason ?state' l
   using False
   unfolding setReason-def
   by auto
ultimately
show ?thesis
   using isReasonAppend[of nth (getF state) reason l elements
?prefix [opposite ?l]]
   using **
   by auto
next
case True
   show ?thesis
   proof (cases ?level = 0)
     case True
     hence currentLevel (getM ?stateM) = 0
     using currentLevelPrefixToLevel[of 0 getM state]
     using *
     unfolding currentLevel-def
     by (simp add: markedElementsAppend)
     hence elementLevel l (getM ?stateM) = 0
     using (?level = 0)
     using elementLevelLeqCurrentLevel[of l getM ?stateM]
     by simp
     with *
     have False
     by simp
     thus ?thesis
     by simp
   next
case False
let reason = length (getF state) − 1
have getReason ?stateM l = Some reason
   using (?level ≠ 0)
   using l = opposite ?l
   unfolding setReason-def
   by auto
moreover
have (nth (getF state) reason) = (getC state)
   using (?level ≠ 0)
using \langle \text{getBackjumpLevel } state > 0 \rightarrow \text{getF } state \neq \text{[]} \wedge \\
\text{last } (\text{getF } state) = \text{getC } state \\
\text{using} \ \text{last-conv-nth[of getF state]} \\
\text{by simp} \\
\text{hence} \ \text{isUnitClause} \ (\text{nth } (\text{getF } state) \ ?\text{reason}) \ l \ (\text{elements } \ ?\text{prefix}) \\
\text{using} \ \text{assms} \\
\text{using} \ \text{applyBackjumpEffect[of state } F0] \\
\text{using} \ (l = \text{opposite } ?l) \\
\text{by } (\text{simp add: Let-def}) \\
\text{hence} \ \text{isReason} \ (\text{nth } (\text{getF } state) \ ?\text{reason}) \ l \ (\text{elements } (\text{getM } \ ?\text{stateM})) \\
\text{using} ** \\
\text{using} \ \text{isUnitClauseIsReason[of nth } (\text{getF } state) \ ?\text{reason } l \\
\text{elements } ?\text{prefix} \ [\text{opposite } ?l]] \\
\text{using} \ (l = \text{opposite } ?l) \\
\text{by simp} \\
\text{moreover} \\
\text{have } ?\text{reason } < \text{length } (\text{getF } state) \\
\text{using } (?\text{level } \neq 0) \\
\text{using} \ (\text{getBackjumpLevel } state > 0 \rightarrow \text{getF } state \neq \text{[]} \wedge \\
\text{last } (\text{getF } state) = \text{getC } state \\
\text{by simp} \\
\text{ultimately} \\
\text{show } ?\text{thesis} \\
\text{using } (?\text{level } \neq 0) \\
\text{using} \ (l = \text{opposite } ?l) \\
\text{using} ** \\
\text{by auto} \\
\text{qed} \\
\text{qed} \\
\text{thus } ?\text{thesis} \\
\text{unfolding} \ \text{InvariantGetReasonIsReason-def} \\
\text{unfolding} \ \text{setReason-def} \\
\text{by auto} \\
\text{qed} \\
\text{thus } ?\text{thesis} \\
\text{using} \ \text{InvariantGetReasonIsReasonAfterNotifyWatches[of } ?\text{stateM getWatchList } ?\text{stateM } ?l \ ?l \ ?\text{prefix False } \text{[]} \text{[]} \\
\text{unfolding} \ \text{applyBackjump-def} \\
\text{unfolding} \ \text{Let-def} \\
\text{unfolding} \ \text{assertLiteral-def} \\
\text{unfolding} \ \text{Let-def} \\
\text{unfolding} \ \text{notifyWatches-def} \\
\text{using} ** \\
\text{using} \ \text{assms} \\
\text{unfolding} \ \text{InvariantWatchListsCharacterization-def} \\
\text{670}
unfolding InvariantWatchListsUniq-def
unfolding InvariantWatchListsContainOnlyClausesFromF-def
by auto
qed

lemma InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-1: assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
and
InvariantUniqC (getC state)
ggetC state = [opposite (getCl state)]
InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))
InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
and
InvariantCEntailed (getConflictFlag state) F0 (getC state) and
InvariantClCharacterization (getCl state) (getC state) (getM state)
and
InvariantClCharacterization (getCl state) (getC state) (getM state)
and
InvariantClCurrentLevel (getCl state) (getM state)
ggetConflictFlag state
isUIP (opposite (getCl state)) (getC state) (getM state)
currentLevel (getM state) > 0
shows
let state’ = applyBackjump state in
InvariantNoDecisionsWhenConflict (getF state’) (getM state’)
(currentLevel (getM state’)) ∧
InvariantNoDecisionsWhenUnit (getF state’) (getM state’)
(currentLevel (getM state’))
proof−
let ?l = getCl state
let ?bClause = getC state
let ?bLiteral = opposite ?l
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level (getM state)
let ?state’ = applyBackjump state
have getM ?state’ = ?prefix @ [(?bLiteral, False)] getF ?state’ = getF state
using assms
using applyBackjumpEffect[of state]
by (auto simp add: Let-def)

show ?thesis

proof –

have ?level < elementLevel ?l (getM state)
  using assms
  using isMinimalBackjumpLevelGetBackjumpLevel[of state]
  unfolding isMinimalBackjumpLevel-def
  unfolding isBackjumpLevel-def
  by (simp add: Let-def)

hence ?level < currentLevel (getM state)
  using elementLevelLeqCurrentLevel[of ?l getM state]
  by simp

have currentLevel (getM ?state') = currentLevel ?prefix
  using (getM ?state') = ?prefix @ [(?bLiteral, False)]
  using markedElementsAppend[of ?prefix [(?bLiteral, False)]]
  unfolding currentLevel-def
  by simp

hence currentLevel (getM ?state') ≤ ?level
  using currentLevelPrefixToLevel[of ?level getM state]
  by simp

show ?thesis

proof –

{ fix level
  assume level < currentLevel (getM ?state')
  hence level < currentLevel ?prefix
    using (currentLevel (getM ?state') = currentLevel ?prefix)
    by simp
  hence prefixToLevel level (getM (applyBackjump state)) =
    prefixToLevel level ?prefix
    using (getM ?state' = ?prefix @ [(?bLiteral, False)]);
    using prefixToLevelAppend[of level ?prefix [(?bLiteral, False)]]
    by simp
  have level < ?level
    using (level < currentLevel ?prefix)
    using (currentLevel (getM ?state') ≤ ?level)
    using (currentLevel (getM ?state') = currentLevel ?prefix)
    by simp
  have prefixToLevel level (getM ?state') = prefixToLevel level ?prefix
    using (getM ?state' = ?prefix @ [(?bLiteral, False)]);
    using prefixToLevelAppend[of level ?prefix [(?bLiteral, False)]]
    using (level < currentLevel ?prefix)
by simp  

hence \( \neg \text{formulaFalse} \ (\text{getF} \ ?\text{state}') \ (\text{elements} \ (\text{prefixToLevel} \ \text{level} \ (\text{getM} \ ?\text{state}'))) \) \ (is \ ?\text{false})  
using \( \langle\text{InvariantNoDecisionsWhenConflict} \ (\text{getF} \ \text{state}) \ (\text{getM} \ \text{state}) \ (\text{currentLevel} \ (\text{getM} \ \text{state})))\rangle\)  
unfolding \( \text{InvariantNoDecisionsWhenConflict-def} \)  
using \( \langle\text{level} < ?\text{level}\rangle\)  
using \( \langle?\text{level} < \text{currentLevel} \ (\text{getM} \ \text{state})\rangle\)  
using \( \langle\text{prefixToLevelPrefixToLevelHigherLevel}[\text{of} \ \text{level} ?\text{level} \ \text{getM} \ \text{state}, \ \text{THEN} \ \text{sgm}]\rangle\)  
using \( \langle\text{getF} \ ?\text{state}' = \text{getF} \ \text{state}\rangle\)  
using \( \langle\text{prefixToLevel level} \ (\text{getM} \ ?\text{state}') = \text{prefixToLevel level} \ ?\text{prefix}\rangle\)  
using \( \langle\text{prefixToLevelPrefixToLevelHigherLevel}[\text{of} \ \text{level} ?\text{level} \ \text{getM} \ \text{state}, \ \text{THEN} \ \text{sgm}]\rangle\)  
by (auto simp add: formulaFalseIffContainsFalseClause)  
moreover  

have \( \neg \ (\exists \ \text{clause literal}. \ 
\text{clause el} \ (\text{getF} \ ?\text{state}') \ \wedge \ 
\text{isUnitClause clause literal} \ (\text{elements} \ (\text{prefixToLevel} \ \text{level} \ (\text{getM} \ ?\text{state}'))) \) \ (is \ ?\text{unit})  
using \( \langle\text{InvariantNoDecisionsWhenUnit} \ (\text{getF} \ \text{state}) \ (\text{getM} \ \text{state}) \ (\text{currentLevel} \ (\text{getM} \ \text{state})))\rangle\)  
unfolding \( \text{InvariantNoDecisionsWhenUnit-def} \)  
using \( \langle\text{level} < ?\text{level}\rangle\)  
using \( \langle?\text{level} < \text{currentLevel} \ (\text{getM} \ \text{state})\rangle\)  
using \( \langle\text{getF} \ ?\text{state}' = \text{getF} \ \text{state}\rangle\)  
using \( \langle\text{prefixToLevel level} \ (\text{getM} \ ?\text{state}') = \text{prefixToLevel level} \ ?\text{prefix}\rangle\)  
using \( \langle\text{prefixToLevelPrefixToLevelHigherLevel}[\text{of} \ \text{level} ?\text{level} \ \text{getM} \ \text{state}, \ \text{THEN} \ \text{sgm}]\rangle\)  
by simp  
ultimately  

have \( ?\text{false} \ \land \ ?\text{unit} \)  
by simp  
}  
thus \( ?\text{thesis}\)  
unfolding \( \text{InvariantNoDecisionsWhenConflict-def} \)  
unfolding \( \text{InvariantNoDecisionsWhenUnit-def} \)  
by (auto simp add: Let-def)  
qed  
qed  
qed

lemma \text{InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-2}:  
assumes  
\( \text{InvariantConsistent} \ (\text{getM} \ \text{state}) \)
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and

InvariantUniqC (getC state)
getC state ≠ [opposite (getCl state)]
InvariantNoDecisionsWhenConflict (butlast (getF state)) (getM state)
(currentLevel (getM state))
InvariantNoDecisionsWhenUnit (butlast (getF state)) (getM state)
(currentLevel (getM state))
getF state ≠ [] last (getF state) = getC state
InvariantNoDecisionsWhenConflict [getC state] (getM state) (getBackjumpLevel state)
InvariantNoDecisionsWhenUnit [getC state] (getM state) (getBackjumpLevel state)

getConflictFlag state
InvariantCFalse (getConflictFlag state) (getM state) (getC state)
and
InvariantCEntailed (getConflictFlag state) F0 (getC state) and
InvariantClCharacterization (getCl state) (getC state) (getM state) and
InvariantClCharacterization (getCl state) (getCl state) (getC state) (getM state) and
InvariantClCurrentLevel (getCl state) (getM state)

isUIP (opposite (getCl state)) (getC state) (getM state)
currentLevel (getM state) > 0
shows
let state’ = applyBackjump state in
 invertNoDecisionsWhenConflict (getF state’) (getM state’)
(currentLevel (getM state’)) ∧
invertNoDecisionsWhenUnit (getF state’) (getM state’)
(currentLevel (getM state’))

proof–
let ?l = getCl state
let ?bClause = getC state
let ?bLiteral = opposite ?l
let ?level = getBackjumpLevel state
let ?prefix = prefixToLevel ?level (getM state)
let ?state’ = applyBackjump state
have getM ?state’ = ?prefix @ [(?bLiteral, False)] getF ?state’ = getF state
using assms
using applyBackjumpEffect[of state]
by (auto simp add: Let-def)
show ?thesis

674
\textbf{proof} –
\begin{verbatim}
have \(?level < \text{elementLevel} \ ?l (\text{getM} \ \text{state})
  using \text{assms}
  using \text{isMinimalBackjumpLevelGetBackjumpLevel[of state]}
  unfolding \text{isMinimalBackjumpLevel-def}
  unfolding \text{isBackjumpLevel-def}
  by \ (\text{simp add: Let-def})
\end{verbatim} 
\begin{verbatim}
hence \(?level < \text{currentLevel} (\text{getM} \ \text{state})
  using \text{elementLevelLeqCurrentLevel[of ?l getM state]}
  by \text{simp}
\end{verbatim}
\begin{verbatim}
have \text{currentLevel} (\text{getM} \ ?state') = \text{currentLevel} \ ?prefix
  using \text{getM ?state' = ?prefix @ [)?bLiteral, False]}\right)
  using \text{markedElementsAppend[of ?prefix [)?bLiteral, False]]}
  unfolding \text{currentLevel-def}
  by \text{simp}
\end{verbatim}
\begin{verbatim}
hence \text{currentLevel} (\text{getM} ?state') \leq \(?level
  using \text{currentLevelPrefixToLevel[of ?level getM state]}
  by \text{simp}
\end{verbatim}
\begin{verbatim}
show \(?thesis
\text{proof}
\{ \\
  \text{fix} \ level \\
  \text{assume} \ level < \text{currentLevel} (\text{getM} \ ?state')
  \text{hence} \ level < \text{currentLevel} \ ?prefix
    using \text{currentLevel (getM ?state') = currentLevel ?prefix}
    by \text{simp}
    hence \text{prefixToLevel level (getM (applyBackjump state)) = prefixToLevel level ?prefix}
      using \text{getM ?state' = ?prefix @ [)?bLiteral, False]}\right)
      using \text{prefixToLevelAppend[of level ?prefix [)?bLiteral, False]]}
      by \text{simp}
  \text{have} \ level < \(?level
    using \text{level < currentLevel ?prefix}
    using \text{currentLevel (getM ?state') \leq ?level}
    using \text{currentLevel (getM ?state') = currentLevel ?prefix}
    by \text{simp}
    \text{have} \ text{prefixToLevel level (getM ?state') = prefixToLevel level ?prefix}
      using \text{getM ?state' = ?prefix @ [)?bLiteral, False]}\right)
      using \text{prefixToLevelAppend[of level ?prefix [)?bLiteral, False]]}
      using \text{level < currentLevel ?prefix}
      by \text{simp}
  \text{have} \ \neg \text{formulaFalse (butlast (getF ?state')) (elements (prefixToLevel level (getM ?state')))}
    using \text{getF ?state' = getF state}
\end{verbatim}

675
using \langle InvariantNoDecisionsWhenConflict (butlast (getF state)) (getM state) (currentLevel (getM state)) \rangle
using \langle level < \#level \rangle
using \langle \#level < currentLevel (getM state) \rangle
using \langle prefixToLevel level (getM ?state') = prefixToLevel level \rangle

using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sym]

unfolding InvariantNoDecisionsWhenConflict-def
by (auto simp add: formulaFalseIffContainsFalseClause)
moreover
have \neg clauseFalse (last (getF ?state')) (elements (prefixToLevel level (getM ?state')))
using \langle getF ?state' = getF state \rangle
using \langle InvariantNoDecisionsWhenConflict [getC state] (getM state) \rangle
using \langle last (getF state) = getC state \rangle
using \langle level < \#level \rangle
using \langle prefixToLevel level (getM ?state') = prefixToLevel level \rangle

using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sym]

unfolding InvariantNoDecisionsWhenConflict-def
by (simp add: formulaFalseIffContainsFalseClause)
moreover
from \langle getF state \neq [] \rangle
have butlast (getF state) @ [last (getF state)] = getF state
using append-butlast-last-id[of getF state]
by simp
hence getF state = butlast (getF state) @ [last (getF state)]
by (rule sym)
ultimately
have \neg formulaFalse (getF ?state') (elements (prefixToLevel level (getM ?state')) (is \#false)
using \langle getF ?state' = getF state \rangle
using set-append[of butlast (getF state) [last (getF state)]]
by (auto simp add: formulaFalseIffContainsFalseClause)

have \neg (\exists clause literal.
clause el (butlast (getF ?state')) \land
isUnitClause clause literal (elements (prefixToLevel level (getM ?state'))))
using \langle InvariantNoDecisionsWhenUnit (butlast (getF state)) (getM state) (currentLevel (getM state)) \rangle

unfolding InvariantNoDecisionsWhenUnit-def
by (simp add: formulaFalseIffContainsFalseClause)
moreover
have \neg clauseFalse (last (getF ?state')) (elements (prefixToLevel level (getM ?state')))
using \langle last (getF state) = getC state \rangle
using \langle level < \#level \rangle
using \langle getF ?state' = getF state \rangle
using \langle prefixToLevel level (getM ?state') = prefixToLevel level \rangle

676
using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sym]
  by simp
moreover
  have ∼ (∃ l. isUnitClause (last (getF ?state'))) l (elements (prefixToLevel level (getM ?state'))))
    using (getF ?state' = getF state)
    using ⟨InvariantNoDecisionsWhenUnit [getC state] (getM state) (getBackjumpLevel state)⟩
    using ⟨last (getF state) = getC state⟩
    using ⟨level < ?level⟩
    using ⟨prefixToLevel level (getM ?state') = prefixToLevel level⟩

using prefixToLevelPrefixToLevelHigherLevel[of level ?level getM state, THEN sym]
unfolding InvariantNoDecisionsWhenUnit-def
  by simp
moreover
from ⟨getF state ≠ []⟩
  have butlast (getF state) @ [last (getF state)] = getF state
    using append-butlast-last-id[of getF state]
    by simp
  hence getF state = butlast (getF state) @ [last (getF state)]
    by (rule sym)
ultimately
  have ∼ (∃ clause literal.
    clause el (getF ?state') ∧
    isUnitClause clause literal (elements (prefixToLevel level (getM ?state')))) (is ?unit)
    using (getF ?state' = getF state)
    using set-append[of butlast (getF state) [last (getF state)]]
    by auto

  have ?false ∧ ?unit
    using ⟨?false⟩ ⟨?unit⟩
    by simp
  }
thus ?thesis
  unfolding InvariantNoDecisionsWhenConflict-def
  unfolding InvariantNoDecisionsWhenUnit-def
  by (auto simp add: Let-def)
qed

lemma InvariantEquivalentZLAfterApplyBackjump:
  assumes
    InvariantConsistent (getM state)

677
\textbf{InvariantUniq} \quad (\text{getM state}) \\
\textbf{InvariantWatchesEl} \quad (\text{getF state}) \ (\text{getWatch1 state}) \ (\text{getWatch2 state}) \\
\textbf{and} \\
\textbf{InvariantWatchListsContainOnlyClausesFromF} \quad (\text{getWatchList state}) \ (\text{getF state}) \ \textbf{and} \\
\text{getConflictFlag state} \\
\textbf{InvariantUniqC} \quad (\text{getC state}) \\
\textbf{InvariantCFalse} \quad (\text{getConflictFlag state}) \ (\text{getM state}) \ (\text{getC state}) \\
\textbf{and} \\
\textbf{InvariantCEntailed} \quad (\text{getConflictFlag state}) \ (\text{getF state}) \ (\text{getC state}) \ (\text{getM state}) \ (\text{getCl state}) \ (\text{getC state}) \ (\text{getM state}) \\
\textbf{and} \\
\textbf{InvariantClCharacterization} \quad (\text{getCl state}) \ (\text{getC state}) \ (\text{getM state}) \ (\text{getCl state}) \ (\text{getC state}) \ (\text{getM state}) \\
\textbf{and} \\
\textbf{InvariantClCurrentLevel} \quad (\text{getCl state}) \ (\text{getM state}) \\
\textbf{InvariantEquivalentZL} \quad (\text{getF state}) \ (\text{getM state}) \ (\text{getF state})' \ (\text{getM state})' \ F0 \\
is\text{UIP} \quad (\text{opposite (getCl state)}) \ (\text{getC state}) \ (\text{getM state}) \\
currentLevel \ (\text{getM state}) > 0 \\
\textbf{shows} \\
\textit{let state'} = \text{applyBackjump state in} \\
\quad \textbf{InvariantEquivalentZL} \quad (\text{getF state'}) \ (\text{getM state'}) \ F0 \\
\textbf{proof–} \\
\begin{align*}
\text{let } & ?l = \text{getCl state} \\
\text{let } & ?bClause = \text{getC state} \\
\text{let } & ?bLiteral = \text{opposite ?l} \\
\text{let } & ?level = \text{getBackjumpLevel state} \\
\text{let } & ?prefix = \text{prefixToLevel ?level (getM state)} \\
\text{let } & ?state = \text{applyBackjump state} \\
\text{have } & \text{formulaEntailsClause F0 ?bClause} \\
& \text{isUnitClause ?bClause ?bLiteral (elements ?prefix)} \\
& \text{getM ?state'} = ?prefix @ [(?bLiteral, False)] \\
& \text{getF ?state'} = \text{getF state} \\
& \text{using } \text{assms} \\
& \text{using } \text{applyBackjumpEffect[of state F0]} \\
& \text{by } (\text{auto simp add: Let-def}) \\
\text{note } & * = \text{this} \\
\text{show } ?\text{thesis} \\
\text{proof } (\text{cases ?level = 0}) \\
\text{case } False \\
\text{have } & ?level < \text{elementLevel ?l (getM state)} \\
& \text{using } \text{assms} \\
& \text{using } \text{isMinimalBackjumpLevelGetBackjumpLevel[of state]} \\
& \text{unfolding } \text{isMinimalBackjumpLevel-def} \\
& \text{unfolding } \text{isBackjumpLevel-def} \\
\end{align*}
by (simp add: Let-def)
hence ?level < currentLevel (getM state)
  using elementLevelLeqCurrentLevel[of ?l getM state]
  by simp

hence prefixToLevel 0 (getM ?state') = prefixToLevel 0 ?prefix
  using *
  using prefixToLevelAppend[of 0 ?prefix [(?bLiteral, False)]]
  using (?level ≠ 0)
  using currentLevelPrefixToLevelEq[of ?level getM state]
  by simp

hence prefixToLevel 0 (getM ?state') = prefixToLevel 0 (getM state)
  using (?level ≠ 0)
  using prefixToLevelPrefixToLevelHigherLevel[of 0 ?level getM state]
  by simp
thus ?thesis
  using *
  using prefixToLevelAppend[of 0 ?prefix [(?bLiteral, False)]]
  using currentLevelPrefixToLevel[of 0 getM state]
  by simp

let ?FM = getF state @ val2form (elements (prefixToLevel 0 (getM state)))
let ?FM' = getF ?state' @ val2form (elements (prefixToLevel 0 (getM ?state')))

have formulaEntailsValuation F0 (elements ?prefix)
  using (?level = 0)
  using val2formIsEntailed[of getF state elements (prefixToLevel 0 (getM state)) []]
  using (InvariantEquivalentZL (getF state) (getM state) F0)
  unfolding formulaEntailsValuation-def
  unfolding InvariantEquivalentZL-def
  unfolding equivalentFormulae-def
  unfolding formulaEntailsLiteral-def
  by auto

have formulaEntailsLiteral (F0 @ val2form (elements ?prefix)) ?bLiteral
using *
using unitLiteralIsEntailed [of ?bClause ?bLiteral elements ?prefix F0]
by simp

have formulaEntailsLiteral F0 ?bLiteral
proof–
{  
  fix valuation::Valuation
  assume model valuation F0
  hence formulaTrue (val2form (elements ?prefix)) valuation
    using ⟨formulaEntailsValuation F0 (elements ?prefix)⟩
    unfolding formulaEntailsValuation-def
    unfolding formulaEntailsLiteral-def
    by simp
    hence formulaTrue (F0 @ (val2form (elements ?prefix))) valuation
      using ⟨model valuation F0⟩
      by (simp add: formulaTrueAppend)
  hence literalTrue ?bLiteral valuation
    using ⟨model valuation F0⟩
    unfolding formulaEntailsLiteral-def
    by auto
}
thus ?thesis
unfolding formulaEntailsLiteral-def
by simp
qed

hence formulaEntailsClause F0 [?bLiteral]
unfolding formulaEntailsLiteral-def
unfolding formulaEntailsClause-def
by (auto simp add: clauseTrueIffContainsTrueLiteral)

hence formulaEntailsClause ?FM [?bLiteral]
using ⟨InvariantEquivalentZL (getF state) (getM state) F0⟩
unfolding InvariantEquivalentZL-def
unfolding equivalentFormulae-def
unfolding formulaEntailsClause-def
by auto

have ?FM’ = ?FM @ [[?bLiteral]]
using *
using ⟨?level = 0⟩
using ⟨prefixToLevel 0 (getM ?state’) = ?prefix @ [[?bLiteral, False]]⟩

by (auto simp add: val2formAppend)

show ?thesis
  using ⟨InvariantEquivalentZL (getF state) (getM state) F0⟩
  using ⟨?FM' = ?FM @ [[?bLiteral]]⟩
  using ⟨formulaEntailsClause ?FM [[?bLiteral]]⟩
  unfolding InvariantEquivalentZL-def
  using extendEquivalentFormulaWithEntailedClause[of F0 ?FM [[?bLiteral]]]
  by (simp add: equivalentFormulaeSymmetry)
qed

lemma InvariantsVarsAfterApplyBackjump:
assumes
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
  (getF state) and

  InvariantWatchListsUniq (getWatchList state)
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state)
  (getWatch2 state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state)
  (getM state) and
  getConflictFlag state
  InvariantCFalse (getConflictFlag state) (getM state) (getC state)
and
  InvariantUniqC (getC state) and
  InvariantCEntailed (getConflictFlag state) F0' (getC state) and
  InvariantClCharacterization (getCl state) (getC state) (getM state)
and
  InvariantClCharacterization (getCl state) (getCl state) (getC state)
  (getM state) and
  InvariantClCurrentLevel (getCl state) (getM state)
  InvariantEquivalentZL (getF state) (getM state) F0'

  isUIP (opposite (getCl state)) (getC state) (getM state)
  currentLevel (getM state) > 0

  vars F0' ⊆ vars F0

  InvariantVarsM (getM state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
shows
  let state' = applyBackjump state in
  InvariantVarsM (getM state') F0 Vbl ∧
  InvariantVarsF (getF state') F0 Vbl ∧
  InvariantVarsQ (getQ state') F0 Vbl

proof –

  let ?l = getCl state
  let ?bClause = getC state
  let ?bLiteral = opposite ?l
  let ?level = getBackjumpLevel state
  let ?prefix = prefixToLevel ?level (getM state)
  let ?state' = state[ getConflictFlag := False, getQ := [], getM :=
  ?prefix ]
  let ?state'' = setReason (opposite (getCl state)) (length (getF state))
  - 1) ?state'
  let ?stateB = applyBackjump state

  have formulaEntailsClause F0' ?bClause
    isUnitClause ?bClause ?bLiteral (elements ?prefix)
    getM ?stateB = ?prefix @ [(?bLiteral, False)]
    getF ?stateB = getF state
    using assms
    using applyBackjumpEffect[of state F0]
    by (auto simp add: Let-def)
  note * = this

  have var ?bLiteral ∈ vars F0 ∪ Vbl
  proof –
    have vars (getC state) ⊆ vars (elements (getM state))
      using (getConflictFlag state)
      using (InvariantCFalse (getConflictFlag state) (getM state) (getC state))
      using valuationContainsItsFalseClausesVariables[of getC state
      elements (getM state)]
      unfolding InvariantCFalse-def
      by simp
    moreover
    have ?bLiteral el (getC state)
      using (InvariantClCharacterization (getCl state) (getC state)
      (getM state))
      unfolding InvariantClCharacterization-def
      unfolding isLastAssertedLiteral-def
      using literalElListIffOppositeLiteralElOppositeLiteralList[of ?bLiteral
      el getC state]
      by simp
ultimately show thesis
  using (InvariantVarsM (getM state) F0 Vbl)
  using (vars F0' ⊆ vars F0)
  unfolding InvariantVarsM-def
  using clauseContainsItsLiteralsVariable[of ?bLiteral getC state]
  by auto
qed

hence InvariantVarsM (getM ?stateB) F0 Vbl
  using (InvariantVarsM (getM state) F0 Vbl)
  using InvariantVarsMAfterBackjump[of getM state F0 Vbl ?prefix
?bLiteral getM ?stateB]
  using *
  by (simp add: isPrefixPrefixToLevel)
moreover
  have InvariantConsistent (prefixToLevel (getBackjumpLevel state)
  (getM state)) @ [(opposite (getCl state), False))]
  InvariantUniq (prefixToLevel (getBackjumpLevel state) (getM state))
  @ [(opposite (getCl state), False))]
  InvariantWatchCharacterization (getF state) (getWatch1 state)
  (getWatch2 state) (prefixToLevel (getBackjumpLevel state) (getM state))
  using assms
  using InvariantConsistentAfterApplyBackjump[of state F0']
  using InvariantUniqAfterApplyBackjump[of state F0']
  using *
  using InvariantWatchCharacterizationInBackjumpPrefix[of state]
  by (auto simp add: Let-def)
  hence InvariantVarsQ (getQ ?stateB) F0 Vbl
  using (InvariantVarsF (getF state) F0 Vbl)
  using (InvariantWatchListsContainOnlyClausesFromF (getWatchList
state) (getF state))
  using (InvariantWatchListsUniq (getWatchList state))
  using (InvariantWatchListsCharacterization (getWatchList state)
  (getWatch1 state) (getWatch2 state))
  using (InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2
state))
  using (InvariantWatchesDiffer (getF state) (getWatch1 state)
  (getWatch2 state))
  using InvariantVarsQAAfterAssertLiteral[of if ?level > 0 then ?state''
else ?state' ?bLiteral False F0 Vbl]
  unfolding applyBackjump-def
  unfolding InvariantVarsQ-def
  unfolding setReason-def
  by (auto simp add: Let-def)
moreover
  have InvariantVarsF (getF ?stateB) F0 Vbl
  using assms
  using assertLiteralEffect[of if ?level > 0 then ?state'' else ?state']
lemma applyDecideEffect:
assumes
\neg \text{vars} (\text{elements} (\text{getM state})) \supseteq Vbl \text{ and}
\text{InvariantWatchesEl} (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state})
and
\text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList state}) (\text{getF state})
shows
let literal = selectLiteral state Vbl in
let state' = applyDecide state Vbl in
var literal \notin \text{vars} (\text{elements} (\text{getM state})) \land
var literal \in Vbl \land
getM state' = getM state \&\& ([literal, True]) \land
getF state' = getF state
using assms
using selectLiteral-def[of Vbl state]
unfolding applyDecide-def
using assertLiteralEffect[of state selectLiteral state Vbl True]
by (simp add: Let-def)

lemma InvariantConsistentAfterApplyDecide:
assumes
\neg \text{vars} (\text{elements} (\text{getM state})) \supseteq Vbl \text{ and}
\text{InvariantConsistent} (\text{getM state}) \text{ and}
\text{InvariantWatchesEl} (\text{getF state}) (\text{getWatch1 state}) (\text{getWatch2 state})
and
\text{InvariantWatchListsContainOnlyClausesFromF} (\text{getWatchList state}) (\text{getF state})
shows
let state' = applyDecide state Vbl in
InvariantConsistent (getM state')
using assms
using applyDecideEffect[of Vbl state]
using InvariantConsistentAfterDecide[of getM state selectLiteral state Vbl getM (applyDecide state Vbl)]
by (simp add: Let-def)

lemma InvariantUniqAfterApplyDecide:
assumes
¬ vars(elements (getM state)) ⊇ Vbl and
InvariantUniq (getM state) and
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
shows
let state' = applyDecide state Vbl in
InvariantUniq (getM state')
using assms
using applyDecideEffect[of Vbl state]
using InvariantUniqAfterDecide[of getM state selectLiteral state Vbl getM (applyDecide state Vbl)]
by (simp add: Let-def)

lemma InvariantQCharacterizationAfterApplyDecide:
assumes
¬ vars(elements (getM state)) ⊇ Vbl and
InvariantConsistent (getM state) and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
InvariantWatchListsUniq (getWatchList state)
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)

getQ state = []
shows
let state' = applyDecide state Vbl in
InvariantQCharacterization (getConflictFlag state') (getQ state') (getF state') (getM state')

proof—
let ?state' = applyDecide state Vbl
let ?literal = selectLiteral state Vbl
have getM ?state' = getM state @ [(?literal, True)]
  using assms
  using applyDecideEffect[of Vbl state]
  by (simp add: Let-def)
  hence InvariantConsistent (getM state @ [(?literal, True)])
  using InvariantConsistentAfterApplyDecide[of Vbl state]
  using assms
  by (simp add: Let-def)
thus ?thesis
  using assms
  using InvariantQCharacterizationAfterAssertLiteralNotInQ[of state ?literal True]
  unfolding applyDecide-def
  by simp
qed

lemma InvariantEquivalentZLAfterApplyDecide:
assumes
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
  InvariantEquivalentZL (getF state) (getM state) F0
shows
  let state' = applyDecide state Vbl in
  InvariantEquivalentZL (getF state') (getM state') F0
proof—
let ?state' = applyDecide state Vbl
let ?l = selectLiteral state Vbl

have getM ?state' = getM state @ [(?l, True)]
  getF ?state' = getF state
  unfolding applyDecide-def
  using assertLiteralEffect[of state ?l True]
  using assms
  by (auto simp only: Let-def)

have prefixToLevel 0 (getM ?state') = prefixToLevel 0 (getM state)
  proof (cases currentLevel (getM state) > 0)
    case True
    thus ?thesis
      using prefixToLevelAppend[of 0 getM state [(?l, True)]]
      using (getM ?state' = getM state @ [(?l, True)])
      by auto
  next

686
case False
  hence prefixToLevel 0 (getM state @ [([?l, True]]) =
    getM state @ (prefixToLevel-aux [([?l, True])])
  by simp
  hence prefixToLevel 0 (getM state @ [([?l, True])]) =
    getM state @ (prefixToLevel-aux [([?l, True])])
  by simp
  thus ?thesis
    using currentLevelZeroTrailEqualsItsPrefixToLevelZero
    of getM state
  by simp
qed

thus ?thesis
  using ⟨getM ?state' = getM state @ [([?l, True])],
    currentLevelZeroTrailEqualsItsPrefixToLevelZero[of getM state]⟩
  using False
  by simp
qed

lemma InvariantGetReasonIsReasonAfterApplyDecide:
  assumes
    ¬ vars (elements (getM state)) ⊇ Vbl
    InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
    (getF state)
    InvariantWatchListsCharacterization (getWatchList state)
    (getWatch1 state) (getWatch2 state) and
    InvariantWatchListsUniq (getWatchList state)
    InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
    InvariantGetReasonIsReason (getReason state) (getF state) (getM state)
    (set (getQ state))
    getQ state = []
shows
    let state' = applyDecide state Vbl in
    InvariantGetReasonIsReason (getReason state')
    (getF state') (getM state') (set (getQ state'))
proof−
  let ?l = selectLiteral state Vbl
  let ?stateM = state ( getM := getM state @ [([?l, True])] )
  have InvariantGetReasonIsReason (getReason ?stateM) (getF ?stateM)
    (getM ?stateM) (set (getQ ?stateM))
proof−
  { fix l::Literal
     assume *: l el (elements (getM ?stateM)) → l el (decisions (getM ?stateM))
     elementLevel l (getM ?stateM) > 0
  }
have \exists \text{ reason}, \text{getReason stateM l = Some reason} \land 
0 \leq \text{ reason} \land \text{ reason} < \text{ length (getF stateM)} \land 
\text{isReason (getF stateM ! reason) l (elements (getM stateM))}

proof (cases l el (elements (getM state)))
  case True
  moreover
  hence \neg l el (decisions (getM state))
  using *
  by (simp add: markedElementsAppend)
  moreover
  have elementLevel l (getM state) > 0
  proof –
    {
      assume \neg \text{thesis}
      with *
      have \text{l = ?l}
        using True
        using elementLevelAppend[of l getM state [(?l, True)]]
        by simp
      hence \text{var ?l \in \text{vars} (elements (getM state))}
        using True
        using valuationContainsItsLiteralsVariable[of l elements (getM state)]
        by simp
      hence \text{False}
        using \neg \text{vars (elements (getM state))} \supseteq \text{Vbl}
        using selectLiteral-def[of \text{Vbl state}]
        by auto
    } thus \text{thesis}
    by auto
  qed

ultimately
obtain reason
  where getReason state l = Some reason \land 
0 \leq \text{ reason} \land \text{ reason} < \text{ length (getF state)} \land 
\text{isReason (getF state ! reason) l (elements (getM state))}
  using InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state));
  unfolding InvariantGetReasonIsReason-def
  by auto
  thus \text{thesis}
  using isReasonAppend[of nth (getF stateM) reason l elements (getM state)] [?l]]
  by auto
next
  case False
  hence \text{l = ?l}
  using *
  by auto

688
hence l el (decisions (getM ?stateM))
  using markedElementIsMarkedTrue[of l getM ?stateM]
  by auto
with *
have False
  by auto
thus ?thesis
  by simp
qed 
}
thus ?thesis
  using (getQ state = [])
  unfolding InvariantGetReasonIsReason-def
  by auto
qed
thus ?thesis
  using assms
  using InvariantGetReasonIsReasonAfterNotifyWatches[of ?stateM
getWatchList ?stateM (opposite ?l)
  opposite ?l getM state True {}
  unfolding applyDecide-def
  unfolding assertLiteral-def
  unfolding notifyWatches-def
  unfolding InvariantWatchListsCharacterization-def
  unfolding InvariantWatchListsContainOnlyClausesFromF-def
  unfolding InvariantWatchListsUniq-def
  using (getQ state = []);
  by (simp add: Let-def)
qed

lemma InvariantsVarsAfterApplyDecide:
assumes
¬ vars (elements (getM state)) ⊇ Vbl
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
(getF state)
InvariantWatchListsUniq (getWatchList state)
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
InvariantVarsM (getM state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
getQ state = []
shows

\[ \text{let } \text{state}' = \text{applyDecide state Vbl in} \]
- \[ \text{InvariantVarsM (getM state') } F0 \text{ Vbl } \land \]
- \[ \text{InvariantVarsF (getF state') } F0 \text{ Vbl } \land \]
- \[ \text{InvariantVarsQ (getQ state') } F0 \text{ Vbl } \]

proof –
- \[ \text{let } ?\text{state}' = \text{applyDecide state Vbl} \]
- \[ \text{let } ?l = \text{selectLiteral state Vbl} \]

have \[ \text{InvariantVarsM (getM ?\text{state}') } F0 \text{ Vbl } \]
- \[ \text{InvariantVarsF (getF ?\text{state}') } F0 \text{ Vbl} \]
- \[ \text{using assms} \]
- \[ \text{using applyDecideEffect[of Vbl state]} \]
- \[ \text{using varsAppendValuation[of elements (getM state) } [?l]] \]
- \[ \text{unfolding InvariantVarsM-def} \]
- \[ \text{by (auto simp add: Let-def)} \]

moreover

have \[ \text{InvariantVarsQ (getQ ?\text{state}') } F0 \text{ Vbl} \]
- \[ \text{using InvariantVarsQAAfterAssertLiteral[of state } ?l \text{ True } F0 \text{ Vbl]} \]
- \[ \text{using assms} \]
- \[ \text{using InvariantConsistentAfterApplyDecide[of Vbl state]} \]
- \[ \text{using InvariantUniqAfterApplyDecide[of Vbl state]} \]
- \[ \text{using assertLiteralEffect[of state } ?l \text{ True]} \]
- \[ \text{unfolding applyDecide-def} \]
- \[ \text{unfolding InvariantVarsQ-def} \]
- \[ \text{by (simp add: Let-def)} \]

ultimately

\[ \text{show } ?\text{thesis} \]
- \[ \text{by (simp add: Let-def)} \]

qed

end

theory SolveLoop

imports UnitPropagate ConflictAnalysis Decide

begin


lemma soundnessForUNSAT:

assumes
- \[ \text{equivalentFormulae } (F \@ val2form M) F0 \]
- \[ \text{formulaFalse } F M \]

shows
\[ \neg \text{ satisfiable } F0 \]

690
proof
  have formulaEntailsValuation \( F \circ\) val2form \( M \) \( M \)
    using val2formIsEntailed[of \( F \) \( M \) []]
    by simp
  moreover
  have formulaFalse \( F \circ\) val2form \( M \) \( M \)
    using formulaFalse \( F \) \( M \)
    by (simp add: formulaFalseAppend)
  ultimately
  have \( \neg \) satisfiable \( F \circ\) val2form \( M \)
    using formulaFalseInEntailedValuationIsUnsatisfiable[of \( F \circ\) val2form \( M \) \( M \) []]
    by simp
  thus ?thesis
    using (equivalentFormulae \( F \circ\) val2form \( M \) \( F_0 \))
    by (simp add: satisfiableEquivalent)
qed

lemma soundnessForSat:
  fixes \( F_0 \) :: Formula and \( F \) :: Formula and \( M \) :: LiteralTrail
  assumes vars \( F_0 \) \( \subseteq \) Vbl and InvariantVarsF \( F \) \( F_0 \) Vbl and InvariantConsistent \( M \) and InvariantEquivalentZL \( F \) \( M \) \( F_0 \) and
  \( \neg \) formulaFalse \( F \) (elements \( M \)) and vars (elements \( M \)) \( \supseteq \) Vbl
  shows model (elements \( M \)) \( F_0 \)
proof
  from (InvariantConsistent \( M \))
  have consistent (elements \( M \))
    unfolding InvariantConsistent-def
    .
  moreover
  from (InvariantVarsF \( F \) \( F_0 \) Vbl)
  have vars \( F \) \( \subseteq \) vars \( F_0 \) \( \cup \) Vbl
    unfolding InvariantVarsF-def
    .
  with (vars \( F_0 \) \( \subseteq \) Vbl)
  have vars \( F \) \( \subseteq \) Vbl
    by auto
  with (vars (elements \( M \)) \( \supseteq \) Vbl)
  have vars \( F \) \( \subseteq \) vars (elements \( M \))
    by simp
  hence formulaTrue \( F \) (elements \( M \)) \( \lor \) formulaFalse \( F \) (elements \( M \))
    by (simp add: totalValuationForFormulaDefinesItsValue)
  with \( \neg \) formulaFalse \( F \) (elements \( M \))
  have formulaTrue \( F \) (elements \( M \))
    by simp
  ultimately
  have model (elements \( M \)) \( F \)
    by simp
  moreover
obtain \( s \) where elements (prefixToLevel 0 M) @ s = elements M using isPrefixPrefixToLevel[of 0 M] using isPrefixElements[of prefixToLevel 0 M M] unfolding isPrefix-def by auto hence elements M = elements (prefixToLevel 0 M) @ s by (rule sym) hence formulaTrue (val2form (elements (prefixToLevel 0 M))) (elements M) using val2formFormulaTrue[of elements (prefixToLevel 0 M) elements M] by auto hence model (elements M) (val2form (elements (prefixToLevel 0 M))) using ⟨consistent (elements M)⟩ by simp ultimately show ?thesis using ⟨InvariantEquivalentZL F M F0⟩ unfolding InvariantEquivalentZL-def unfolding equivalentFormulae-def using formulaTrueAppend[of F val2form (elements (prefixToLevel 0 M)) elements M] by auto qed

definition satFlagLessState = \{ (state1::State, state2::State). (getSATFlag state1) ≠ UNDEF ∧ (getSATFlag state2) = UNDEF \}
have ∀ state'. (state', stateDef) ∈ satFlagLessState → state' /∈ Q

proof
  fix state'
  show (state', stateDef) ∈ satFlagLessState → state' /∈ Q
  proof
    assume (state', stateDef) ∈ satFlagLessState
    hence getSATFlag stateDef = UNDEF
    unfolding satFlagLessState-def
    by auto
    with (getSATFlag stateDef /≠ UNDEF) have False
    by simp
    thus state' /∈ Q
    by simp
  qed
  qed
  with ⟨stateDef ∈ Q⟩
  show ?thesis
  by auto
next
  case False
  have ∀ state'. (state', state) ∈ satFlagLessState → state' /∈ Q
  proof
    fix state'
    show (state', state) ∈ satFlagLessState → state' /∈ Q
    proof
      assume (state', state) ∈ satFlagLessState
      hence getSATFlag state' /≠ UNDEF
      unfolding satFlagLessState-def
      by simp
      with False
      show state' /∈ Q
      by auto
    qed
    qed
    with ⟨state ∈ Q⟩
    show ?thesis
    by auto
  qed
  qed
  with ⟨stateDef ∈ Q⟩
  show ?thesis
  by auto

definition
lexLessState1 Vbl = {(state1::State, state2::State).
  getSATFlag state1 = UNDEF ∧ getSATFlag state2 = UNDEF ∧
  (getM state1, getM state2) ∈ lexLessRestricted Vbl}
lemma wellFoundedLexLessState1:
assumes
finite Vbl
shows
wf (lexLessState1 Vbl)
unfolding wf-eq-minimal
proof−
  show ∀ Q state. state ∈ Q → (∃ stateMin ∈ Q. ∀ state′. (state′, stateMin) ∈ lexLessState1 Vbl → state′ /∈ Q)
  proof−
  { 
    fix Q :: State set and state :: State
    assume state ∈ Q 
    let ?Q1 = { M::LiteralTrail. ∃ state. state ∈ Q ∧ getSATFlag state = UNDEF ∧ (getM state) = M }
    have ∃ stateMin ∈ Q. (∀ state′. (state′, stateMin) ∈ lexLessState1 Vbl → state′ /∈ ?Q1) 
        proof (cases ?Q1 ≠ {}) 
          case True 
            then obtain M::LiteralTrail 
                where M ∈ ?Q1 
                by auto 
            then obtain MMMin::LiteralTrail 
                where MMMin ∈ ?Q1 ∀ M′. (M′, MMMin) ∈ lexLessRestricted Vbl → M′ /∈ ?Q1 
                using wfLexLessRestricted[of Vbl] (finite Vbl) 
                unfolding wf-eq-minimal 
                apply simp 
                apply (erule-tac x=?Q1 in allE) 
                by auto 
            from ⟨ MMMin ∈ ?Q1 ⟩ obtain stateMin 
                where stateMin ∈ Q (getM stateMin) = MMMin getSATFlag stateMin = UNDEF 
                by auto 
            have ∀ state′. (state′, stateMin) ∈ lexLessState1 Vbl → state′ /∈ Q 
                proof 
                  fix state′ 
                  show (state′, stateMin) ∈ lexLessState1 Vbl → state′ /∈ Q 
                    proof 
                      assume (state′, stateMin) ∈ lexLessState1 Vbl 
                      hence getSATFlag state′ = UNDEF (getM state′, getM stateMin) ∈ lexLessRestricted Vbl 
                        unfolding lexLessState1-def 
                      by auto 
                      hence getM state′ /∈ ?Q1 
                        using (∀ M′. (M′, MMMin) ∈ lexLessRestricted Vbl → M′ /∈ ?Q1)
        } 
  } 

using \((get\ M \text{stateMin}) = \text{MMin}\) by auto
thus \(\text{state}' \notin Q\)
using \((get\SAT\text{Flag} \text{state}' = \text{UNDEF})\) by auto
qed
qed
thus \(?\text{thesis}\)
using \(\langle\text{stateMin} \in Q\rangle\) by auto
next
case False
have \(\forall \text{state}', (\text{state}', \text{state}) \in \text{lexLessState1 Vbl} \rightarrow \text{state}' \notin Q\)
proof
fix \text{state}'
show \((\text{state}', \text{state}) \in \text{lexLessState1 Vbl} \rightarrow \text{state}' \notin Q\)
proof
assume \((\text{state}', \text{state}) \in \text{lexLessState1 Vbl}\)

hence \(\text{get}\SAT\text{Flag} \text{state} = \text{UNDEF}\)
unfolding \text{lexLessState1-def}\nby simp
hence \((\text{get}\ M \text{state}) \in ?Q1\)
using \(\langle\text{state} \in Q\rangle\) by auto
hence False
using False
by auto
thus \(\text{state}' \notin Q\)
by simp
qed
qed
thus \(?\text{thesis}\)
using \(\langle\text{state} \in Q\rangle\) by auto
qed
 qed

definition
\text{terminationLessState1 Vbl} = \{(\text{state1}::\text{State}, \text{state2}::\text{State}).
(\text{state1}, \text{state2}) \in \text{satFlagLessState} \lor
(\text{state1}, \text{state2}) \in \text{lexLessState1 Vbl}\}

lemma \text{wellFoundedTerminationLessState1}:
assumes \text{finite Vbl}
shows \( \text{wf} (\text{terminationLessState1} \ \text{Vbl}) \)

unfolding \( \text{wf-eq-minimal} \)

proof –

\[
\begin{align*}
\text{show} \forall \ Q \ \text{state}. \ \text{state} \in Q \rightarrow (\exists \ \text{stateMin} \in Q. \ \forall \ \text{state}'. (\text{state}', \text{stateMin}) \in \text{terminationLessState1} \ \text{Vbl} \rightarrow \text{state}' \notin Q) \\
\text{proof} – \\
\{ \\
\text{fix} \ Q::\text{State set} \\
\text{fix} \ \text{state}::\text{State} \\
\text{assume} \ \text{state} \in Q \\
\text{have} \ \exists \ \text{stateMin} \in Q. \ \forall \ \text{state}'. (\text{state}', \text{stateMin}) \in \text{terminationLessState1} \ \text{Vbl} \rightarrow \text{state}' \notin Q \\
\text{proof} – \\
\text{obtain} \ \text{state0} \\
\text{where} \ \text{state0} \in Q \ \forall \ \text{state}'. (\text{state}', \text{state0}) \in \text{satFlagLessState} \rightarrow \text{state}' \notin Q \\
\text{using} \ \text{wellFoundedSatFlagLessState} \\
\text{unfolding} \ \text{wf-eq-minimal} \\
\text{using} \ \langle \text{state} \in Q \rangle \\
\text{by} \ \text{auto} \\
\text{show} \ \text{thesis} \\
\text{proof} \ (\text{cases \ getSATFlag} \ \text{state0} = \text{UNDEF}) \\
\text{case} \ False \\
\text{hence} \ \forall \ \text{state}'. (\text{state}', \text{state0}) \in \text{terminationLessState1} \ \text{Vbl} \rightarrow \text{state}' \notin Q \\
\text{using} \ \forall \ \text{state}'. (\text{state}', \text{state0}) \in \text{satFlagLessState} \rightarrow \text{state}' \\
\notin Q) \\
\text{unfolding} \ \text{terminationLessState1-def} \\
\text{unfolding} \ \text{lexLessState1-def} \\
\text{by} \ \text{simp} \\
\text{thus} \ \text{thesis} \\
\text{using} \ \langle \text{state0} \in Q \rangle \\
\text{by} \ \text{auto} \\
\text{next} \\
\text{case} \ True \\
\text{then obtain} \ \text{state1} \\
\text{where} \ \text{state1} \in Q \ \forall \ \text{state}'. (\text{state}', \text{state1}) \in \text{lexLessState1} \ Vbl \rightarrow \text{state}' \notin Q \\
\text{using} \ \langle \text{finite} \ \text{Vbl} \rangle \\
\text{using} \ \langle \text{state} \in Q \rangle \\
\text{using} \ \text{wellFoundedLexLessState1}[\text{of} \ \text{Vbl}] \\
\text{unfolding} \ \text{wf-eq-minimal} \\
\text{by} \ \text{auto} \\
\text{have} \ \forall \ \text{state}'. (\text{state}', \text{state1}) \in \text{terminationLessState1} \ \text{Vbl} \rightarrow \text{state}' \notin Q \\
\text{using} \ \forall \ \text{state}'. (\text{state}', \text{state1}) \in \text{lexLessState1} \ \text{Vbl} \rightarrow \text{state}' \\
\notin Q) \\
\text{unfolding} \ \text{terminationLessState1-def}
\end{align*}
\]

696
using \( \forall \text{state}', \text{state}', \text{state0} \in \text{satFlagLessState} \rightarrow \text{state}' \notin Q \) using True unfolding satFlagLessState-def by simp thus \(?\)thesis using \( \text{state1} \in Q \) by auto qed qed thus \(?\)thesis by auto qed qed lemma transTerminationLessState1: trans \((\text{terminationLessState1 Vbl})\) proof— \{ fix \( x::\text{State} \) and \( y::\text{State} \) and \( z::\text{State} \) assume \((x, y) \in \text{terminationLessState1 Vbl} \) \((y, z) \in \text{termination- LessState1 Vbl} \) have \((x, z) \in \text{terminationLessState1 Vbl} \) proof (cases \((x, y) \in \text{satFlagLessState} \) ) case True hence \( \text{getSATFlag x} \neq \text{UNDEF} \) \( \text{getSATFlag y} = \text{UNDEF} \) unfolding satFlagLessState-def by auto hence \( \text{getSATFlag z} = \text{UNDEF} \) unfolding \( (y, z) \in \text{terminationLessState1 Vbl} \) unfolding \((\text{satFlagLessState-def})\) unfolding \((\text{lexLessState1-def})\) by auto thus \(?\)thesis unfolding \((\text{getSATFlag x} \neq \text{UNDEF})\) unfolding \((\text{terminationLessState1-def})\) unfolding \((\text{satFlagLessState-def})\) by simp next case False with \((x, y) \in \text{terminationLessState1 Vbl} \) have \( \text{getSATFlag x} = \text{UNDEF} \) \( \text{getSATFlag y} = \text{UNDEF} \) \((\text{getM x, getM y}) \in \text{lexLessRestricted Vbl} \) unfolding \((\text{terminationLessState1-def})\) unfolding \((\text{lexLessState1-def})\) by auto hence \( \text{getSATFlag z} = \text{UNDEF} \) \((\text{getM y, getM z}) \in \text{lexLessRe-
restricted Vbl
using \((y, z) \in \text{terminationLessState1 Vbl}\)
unfolding \text{terminationLessState1-def}
unfolding \text{satFlagLessState-def}
unfolding \text{lexLessState1-def}
by auto
thus \(?thesis
using \((\text{getSATFlag } x = \text{UNDEF})\)
using \(((\text{getM } x, \text{getM } y) \in \text{lexLessRestricted Vbl})\)
using \text{transLexLessRestricted[of Vbl]}
unfolding \text{trans-def}
unfolding \text{terminationLessState1-def}
unfolding \text{satFlagLessState-def}
unfolding \text{lexLessState1-def}
by blast
qed

}\) thus \(?thesis
unfolding \text{trans-def}
by blast
qed

lemma \text{transTerminationLessState1I}:
assumes
\((x, y) \in \text{terminationLessState1 Vbl}\)
\((y, z) \in \text{terminationLessState1 Vbl}\)
shows
\((x, z) \in \text{terminationLessState1 Vbl}\)
using \text{assms}
using \text{transTerminationLessState1[of Vbl]}
unfolding \text{trans-def}
by blast

lemma \text{TerminationLessAfterExhaustiveUnitPropagate}:
assumes
exhaustiveUnitPropagate-dom state
InvariantUniq (getM state)
InvariantConsistent (getM state)
InvariantWatchListsContainOnlyClausesFromF (getWatchList state)
\((getF state) \text{ and} \)
InvariantWatchListsUniq (getWatchList state) \text{ and}
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
InvariantWatchesEl \((getF state) (getWatch1 state) (getWatch2 state)\)
\text{ and}
InvariantWatchesDiffer \((getF state) (getWatch1 state) (getWatch2 state)\)
InvariantWatchCharacterization \((getF state) (getWatch1 state) (getWatch2 state)\)

698
state) (getM state)
  InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
  InvariantUniqQ (getQ state)
  InvariantVarsM (getM state) F0 Vbl
  InvariantVarsQ (getQ state) F0 Vbl
  InvariantVarsF (getF state) F0 Vbl
  finite Vbl
  getSATFlag state = UNDEF

shows
  let state' = exhaustiveUnitPropagate state in
  state' = state ∨ (state', state) ∈ terminationLessState1 (vars F0 ∪ Vbl)
  using assms

proof (induct state rule: exhaustiveUnitPropagate-dom.induct)
  case (step state')
  note ih = this
  show ?case
  proof (cases (getConflictFlag state') ∨ (getQ state') = [])
    case True
    with exhaustiveUnitPropagate.simps[of state']
    have exhaustiveUnitPropagate state' = state'
      by simp
    thus ?thesis
      using True
      by (simp add: Let-def)
  next
  case False
  let ?state'' = applyUnitPropagate state'
  have exhaustiveUnitPropagate state' = exhaustiveUnitPropagate ?state''
    using exhaustiveUnitPropagate.simps[of state']
    using False
    by simp
  have InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'') and
    InvariantWatchListsUniq (getWatchList ?state'') and
    InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'') (getWatch2 ?state'')
    using ih
    using WatchInvariantsAfterAssertLiteral[of state' hd (getQ state')] False

699
unfolding applyUnitPropagate-def
by (auto simp add: Let-def)
moreover
have InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getM ?state'')
  using ih
  using InvariantWatchCharacterizationAfterApplyUnitPropagate[of state']
  unfolding InvariantQCharacterization-def
  using False
  by (simp add: Let-def)
moreover
have InvariantQCharacterization (getConflictFlag ?state'') (getQ ?state'') (getF ?state'') (getM ?state'')
  using ih
  using InvariantQCharacterizationAfterApplyUnitPropagate[of state']
  using False
  by (simp add: Let-def)
moreover
have InvariantConflictFlagCharacterization (getConflictFlag ?state'')
  (getF ?state'') (getM ?state'')
  using ih
  using InvariantConflictFlagCharacterizationAfterApplyUnitPropagate[of state']
  using False
  by (simp add: Let-def)
moreover
have InvariantUniqQ (getQ ?state'')
  using ih
  using InvariantUniqQAfterApplyUnitPropagate[of state']
  using False
  by (simp add: Let-def)
moreover
have InvariantConsistent (getM ?state'')
  using ih
  using InvariantConsistentAfterApplyUnitPropagate[of state']
  using False
  by (simp add: Let-def)
moreover
have InvariantUniq (getM ?state'')
  using ih
  using InvariantUniqAfterApplyUnitPropagate[of state']
  using False
  by (simp add: Let-def)
moreover
have InvariantVarsM (getM ?state'') F0 Vbl InvariantVarsQ (getQ ?state'') F0 Vbl
  using ih
using False
using InvariantsVarsAfterApplyUnitPropagate[of state' F0 Vbl]
by (auto simp add: Let-def)
moreover
have InvariantVarsF (getF ?state'') F0 Vbl
  unfolding applyUnitPropagate-def
  using assertLiteralEffect[of state' hd (getQ state') False]
  using ih
  by (simp add: Let-def)
moreover
have getSATFlag ?state'' = UNDEF
  unfolding applyUnitPropagate-def
  using ⟨InvariantWatchListsContainOnlyClausesFromF (getWatchList state') (getF state')⟩
  (getWatch1 state')
  (getWatch2 state')
  using ⟨InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state')⟩
  using ⟨InvariantUniq (getM state')⟩
  using ⟨InvariantConsistent (getM state')⟩
  using ⟨InvariantVarsM (getM state') F0 Vbl⟩
  using ⟨InvariantWatchesEl (getF state') (getWatch1 state') (getWatch2 state')⟩
  using ⟨InvariantWatchListsContainOnlyClausesFromF (getWatchList state') (getF state')⟩
  using ⟨InvariantQCharacterization (getConflictFlag state') (getQ state') (getF state') (getM state')⟩
  using ⟨InvariantUniq (getM ?state'')⟩
  using ⟨InvariantConsistent (getM ?state'')⟩
  using ⟨InvariantVarsM (getM ?state'') F0 Vbl⟩
  using ⟨getSATFlag state' = UNDEF⟩
  using ⟨getSATFlag ?state'' = UNDEF⟩
unfolding terminationLessState1-def

701
lemma InvariantsAfterSolveLoopBody:
assumes
  getSATFlag state = UNDEF
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  and
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
  and
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  and
  InvariantWatchListsUniq (getWatchList state)
  and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
and
  InvariantUniqQ (getQ state)
  and
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
  and
  InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
  and
  InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))
  and
  InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))
and
  InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
and
  InvariantEquivalentZL (getF state) (getM state) F0' and
  InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
and
  finite Vbl
  vars F0' ⊆ vars F0
  vars F0 ⊆ Vbl
  InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ \ (getQ \ state) \ F0 \ Vbl
InvariantVarsF \ (getF \ state) \ F0 \ Vbl

shows

let \( state' = \) solve-loop-body \( state \) \( Vbl \) in

\( \langle \text{InvariantConsistent} \ (getM \ state') \ \land \\
\text{InvariantUniq} \ (getM \ state') \ \land \\
\text{InvariantWatchesEl} \ (getF \ state') \ (getWatch1 \ state') \ (getWatch2 \ state') \ \land \\
\text{InvariantWatchesDiffer} \ (getF \ state') \ (getWatch1 \ state') \ (getWatch2 \ state') \ \land \\
\text{InvariantWatchCharacterization} \ (getF \ state') \ (getWatch1 \ state') \ \\
\text{InvariantWatchListsContainOnlyClausesFromF} \ (getWatchList \ state') \ (getF \ state') \ \land \\
\text{InvariantWatchListsUniq} \ (getWatchList \ state') \ (getWatch1 \ state') \ (getWatch2 \ state') \ \land \\
\text{InvariantWatchListsCharacterization} \ (getWatchList \ state') \ (getWatch1 \ state') \ (getWatch2 \ state') \ \\
\text{InvariantQCharacterization} \ (getConflictFlag \ state') \ (getQ \ state') \ \\
\text{InvariantConflictFlagCharacterization} \ (getConflictFlag \ state') \ (getF \ state') \ (getM \ state') \ \land \\
\text{InvariantConflictClauseCharacterization} \ (getConflictFlag \ state') \ (getConflictClause \ state') \ (getF \ state') \ (getM \ state') \ \land \\
\text{InvariantUniqQ} \ (getQ \ state') \ \\
\text{InvariantConflictFlagCharacterization} \ (getConflictFlag \ state') \ (getConflictClause \ state') \ (getF \ state') \ (getM \ state') \ \land \\
\text{InvariantEquivalentZL} \ (getF \ state') \ (getM \ state') \ F0' \ \land \\
\text{InvariantGetReasonIsReason} \ (getReason \ state') \ (getF \ state') \ (getM \ state') \ (set \ (getQ \ state')) \ \land \\
\text{InvariantVarsM} \ (getM \ state') \ F0 \ Vbl \ \land \\
\text{InvariantVarsQ} \ (getQ \ state') \ F0 \ Vbl \ \land \\
\text{InvariantVarsF} \ (getF \ state') \ F0 \ Vbl \ \land \\
\text{(state', state)} \in \text{terminationLessState1} \ (vars \ F0 \cup \ Vbl) \ \land \\
\langle \text{getSATFlag} \ state' = \text{FALSE} \rightarrow \neg \text{satisfiable} \ F0' \ \land \\
\text{getSATFlag} \ state' = \text{TRUE} \rightarrow \text{satisfiable} \ F0' \rangle \ \\
\rangle \ (\text{is let states'} = \text{solve-loop-body state Vbl in ?inv' state' \ \land \ ?inv'' state' \ ^{\land -} )

proof–

let ?state-up = \text{exhaustiveUnitPropagate} \ state

have \text{exhaustiveUnitPropagate-dom} \ state
using \text{exhaustiveUnitPropagateTermination[of state F0 Vbl]}
using assms
by simp

have ?inv' ?state-up
using assms
using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

using InvariantsAfterExhaustiveUnitPropagate[of state]

using InvariantConflictClauseCharacterizationAfterExhaustivePropagate[of state]

by (simp add: Let-def)

have \textquote{\textit{inv}'} state-up

using assms

using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

using InvariantsNoDecisionsWhenConflictNorUnitAfterExhaustivePropagate[of state]

by (simp add: Let-def)

have InvariantEquivalentZL(getF state-up) (getM state-up) F0'

using assms

using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

using InvariantEquivalentZLAfterExhaustiveUnitPropagate[of state]

by (simp add: Let-def)

have InvariantGetReasonIsReason (getReason state-up) (getF state-up) (getM state-up) (set (getQ state-up))

using assms

using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

using InvariantGetReasonIsReasonAfterExhaustiveUnitPropagate[of state]

by (simp add: Let-def)

have getSATFlag state-up = getSATFlag state

using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

by (simp add: Let-def)

have getConflictFlag state-up ∨ getQ state-up = []

using conflictFlagOrQEmptyAfterExhaustiveUnitPropagate[of state]

using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

by (simp add: Let-def)

have InvariantVarsM (getM state-up) F0 Vbl

InvariantVarsQ (getQ state-up) F0 Vbl

InvariantVarsF (getF state-up) F0 Vbl

using assms

using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

using InvariantsAfterExhaustiveUnitPropagate[of state F0 Vbl]

by (auto simp add: Let-def)

have state-up = state ∨ (?state-up, state) ∈ terminationLessState1 (vars F0 ∪ Vbl)

using assms

using \langle\textit{exhaustiveUnitPropagate-dom state}\rangle

by (simp add: Let-def)

show ?thesis

proof (cases getConflictFlag state-up)
case True
show ?thesis
proof (cases currentLevel (getM ?state-up) = 0)
case True
  hence prefixToLevel 0 (getM ?state-up) = (getM ?state-up)
  using currentLevelZeroTrailEqualsItsPrefixToLevelZero[of getM ?state-up]
  by simp
moreover
have formulaFalse (getF ?state-up) (elements (getM ?state-up))
  using (getConflictFlag ?state-up)
  using (?inv' ?state-up)
  unfolding InvariantConflictFlagCharacterization-def
  by simp
ultimately
have ¬ satisfiable F0'
  using (InvariantEquivalentZL (getF ?state-up) (getM ?state-up))
F0'
  unfolding InvariantEquivalentZL-def
  using soundnessForUNSAT[of getF ?state-up elements (getM ?state-up)]
  by simp
moreover
let ?state' = ?state-up ( getSATFlag := FALSE )
have (?state', state) ∈ terminationLessState1 ( vars F0 ∪ Vbl)
  unfolding terminationLessState1-def
  unfolding satFlagLessState-def
  using (getSATFlag state = UNDEF)
  by simp
ultimately
show ?thesis
  using (?inv' ?state-up)
  using (?inv'' ?state-up)
  using (InvariantEquivalentZL (getF ?state-up) (getM ?state-up))
F0'
  using (InvariantVarsM (getM ?state-up) F0 Vbl)
  using (InvariantVarsQ (getQ ?state-up) F0 Vbl)
  using (InvariantVarsF (getF ?state-up) F0 Vbl)
  using (getConflictFlag ?state-up)
  using (currentLevel (getM ?state-up) = 0)
  unfolding solve-loop-body-def
  by (simp add: Let-def)
next
case False
show ?thesis
proof—
let ?state-c = applyConflict ?state-up

have ?inv' ?state-c
  ?inv'' ?state-c
  getConflictFlag ?state-c
  InvariantEquivalentZL (getF ?state-c) (getM ?state-c) F0'
  currentLevel (getM ?state-c) > 0
  using (?inv' ?state-up) (?inv'' ?state-up)
  using (getConflictFlag ?state-up)
  using (InvariantEquivalentZL (getF ?state-up) (getM ?state-up))

F0'
  using (?currentLevel (getM ?state-up) ≠ 0)
  unfolding applyConflict-def
  unfolding setConflictAnalysisClause-def
by (auto simp add: Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def)

have InvariantCFalse (getConflictFlag ?state-c) (getM ?state-c) (getF ?state-c)
  (getM ?state-c)
  InvariantCEntailed (getConflictFlag ?state-c) F0' (getC ?state-c)
  (getM ?state-c)
  InvariantClCharacterization (getCl ?state-c) (getC ?state-c) (getM ?state-c)
  InvariantCnCharacterization (getCn ?state-c) (getC ?state-c) (getM ?state-c)
  InvariantClCurrentLevel (getCl ?state-c) (getM ?state-c)
  InvariantUniqC (getC ?state-c)
  using (getConflictFlag ?state-up)
  using (?currentLevel (getM ?state-up) ≠ 0)
  using (?inv' ?state-up)
  using (?inv'' ?state-up)
  using (InvariantEquivalentZL (getF ?state-up) (getM ?state-up))

F0'
  using InvariantsClAfterApplyConflict[of ?state-up]
by (auto simp only: Let-def)

have getSATFlag ?state-c = getSATFlag state
  using (getSATFlag ?state-up = getSATFlag state)
  unfolding applyConflict-def
  unfolding setConflictAnalysisClause-def
by (simp add: Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def)

have getReason ?state-c = getReason ?state-up
  getF ?state-c = getF ?state-up
  getM ?state-c = getM ?state-up
  getQ ?state-c = getQ ?state-up
  unfolding applyConflict-def
  unfolding setConflictAnalysisClause-def
by (auto simp add: Let-def findLastAssertedLiteral-def countCurrentLevelLiterals-def)

hence InvariantGetReasonIsReason (getReason ?state-c) (getF)
have getM ?state-c = getM state ∨ (?state-c, state) ∈ terminationLessState1 (vars F0 ∪ Vbl)
using (?state-up = state ∨ (?state-up, state) ∈ terminationLessState1 (vars F0 ∪ Vbl))

let ?state-euip = applyExplainUIP ?state-c
let ?l′ = getC ?state-euip

have applyExplainUIP-dom ?state-c
using ApplyExplainUIPTermination[of ?state-c F0]
using (getConflictFlag ?state-c)
using (InvariantEquivalentZL (getF ?state-c) (getM ?state-c) (getC ?state-c))
"F0"
using (currentLevel (getM ?state-c) > 0)
using (?inv' ?state-c)
using (InvariantCFalse (getConflictFlag ?state-c) (getM ?state-c) (getC ?state-c))
using ⟨InvariantCEntailed (getConflictFlag ?state-c) F0’ (getC ?state-c)⟩
using ⟨InvariantClCharacterization (getCl ?state-c) (getC ?state-c) (getM ?state-c)⟩
using ⟨InvariantCnCharacterization (getCn ?state-c) (getC ?state-c) (getM ?state-c)⟩
using ⟨InvariantClCurrentLevel (getCl ?state-c) (getM ?state-c)⟩
by simp

have ?inv’ ?state-euip ?inv’’ ?state-euip
using ⟨?inv’ ?state-c⟩ ⟨?inv’’ ?state-c⟩
using ⟨applyExplainUIP-dom ?state-c⟩
using ApplyExplainUIPPreservedVariables[of ?state-c]
by (auto simp add: Let-def)

have ⟨InvariantCFalse (getConflictFlag ?state-euip) (getM ?state-euip)⟩
InvariantCEntailed (getConflictFlag ?state-euip) F0’ (getC ?state-euip)
InvariantClCurrentLevel (getCl ?state-euip) (getM ?state-euip)
InvariantUniqC (getC ?state-euip)
using ⟨?inv’ ?state-c⟩
using ⟨InvariantCFalse (getConflictFlag ?state-c) (getM ?state-c)⟩
using ⟨InvariantCEntailed (getConflictFlag ?state-c) F0’ (getC ?state-c)⟩
using ⟨InvariantClCharacterization (getCl ?state-c) (getC ?state-c) (getM ?state-c)⟩
using ⟨InvariantCnCharacterization (getCn ?state-c) (getC ?state-c) (getM ?state-c)⟩
using ⟨InvariantClCurrentLevel (getCl ?state-c) (getM ?state-c)⟩
using ⟨InvariantUniqC (getC ?state-c)⟩
using ⟨getConflictFlag ?state-c⟩
using ⟨currentLevel (getM ?state-c) > 0⟩
using ⟨applyExplainUIP-dom ?state-c⟩
using InvariantsClAfterExplainUIP[of ?state-c F0’]
by (auto simp only: Let-def)
have InvariantEquivalentZL (getF ?state-euip) (getM ?state-euip) F0' using ⟨InvariantEquivalentZL (getF ?state-c) (getM ?state-c)⟩

using ⟨applyExplainUIP-dom ?state-c⟩
using ApplyExplainUIPPreservedVariables[of ?state-c]
by (simp only: Let-def)

using ⟨applyExplainUIP-dom ?state-c⟩
using ApplyExplainUIPPreservedVariables[of ?state-c]
by (simp only: Let-def)

have getConflictFlag ?state-euip using ⟨getConflictFlag ?state-c⟩
using ⟨applyExplainUIP-dom ?state-c⟩
using ApplyExplainUIPPreservedVariables[of ?state-c]
by (simp add: Let-def)

hence getSATFlag ?state-euip = getSATFlag state using ⟨getSATFlag ?state-c = getSATFlag state⟩
using ⟨applyExplainUIP-dom ?state-c⟩
using ApplyExplainUIPPreservedVariables[of ?state-c]
by (simp add: Let-def)

have isUIP (opposite (getCl ?state-euip)) (getC ?state-euip) (getM ?state-euip) using ⟨applyExplainUIP-dom ?state-c⟩
using ⟨inv' ?state-c⟩
using ⟨InvariantCFalse (getConflictFlag ?state-c) (getM ?state-c) (getC ?state-c)⟩
using ⟨InvariantCEntailed (getConflictFlag ?state-c) F0' (getC ?state-c)⟩
using ⟨InvariantCCharacterization (getCl ?state-c) (getC ?state-c) (getM ?state-c)⟩
using ⟨InvariantCnCharacterization (getCn ?state-c) (getM ?state-c)⟩
using ⟨InvariantClCurrentLevel (getCl ?state-c) (getM ?state-c)⟩
using ⟨InvariantEquivalentZL (getF ?state-c) (getM ?state-c)⟩

F0'
using ⟨getConflictFlag ?state-c⟩
using ⟨currentLevel (getM ?state-c) > 0⟩
using isUIPApplyExplainUIP[of ?state-c]
by (simp add: Let-def)
have currentLevel (getM ?state-euip) > 0
  using (applyExplainUIP-dom ?state-c)
  using ApplyExplainUIPPreservedVariables[of ?state-c]
  using (currentLevel (getM ?state-c) > 0)
  by (simp add: Let-def)

have InvariantVarsM (getM ?state-euip) F0 Vbl
  InvariantVarsQ (getQ ?state-euip) F0 Vbl
  InvariantVarsF (getF ?state-euip) F0 Vbl
  using (InvariantVarsM (getM ?state-c) F0 Vbl)
  using (InvariantVarsQ (getQ ?state-c) F0 Vbl)
  using (InvariantVarsF (getF ?state-c) F0 Vbl)
  using (applyExplainUIP-dom ?state-c)
  using ApplyExplainUIPPreservedVariables[of ?state-c]
  by (auto simp add: Let-def)

have getM ?state-euip = getM state ∨ (?state-euip, state) ∈
terminationLessState1 (vars F0 ∪ Vbl)
  using (getM ?state-c = getM state ∨ (?state-c, state) ∈
terminationLessState1 (vars F0 ∪ Vbl))
  using (applyExplainUIP-dom ?state-c)
  using ApplyExplainUIPPreservedVariables[of ?state-c]
  unfolding terminationLessState1-def
  unfolding satFlagLessState1-def
  unfolding lexLessState1-def
  unfolding lexLessRestricted-def
  by (simp add: Let-def)

let ?state-l = applyLearn ?state-euip
let $?l' = getCl ?state-l

have $: getM ?state-l = getM ?state-euip ∧
  getQ ?state-l = getQ ?state-euip ∧
  getC ?state-l = getC ?state-euip ∧
  getCl ?state-l = getCl ?state-euip ∧
  getConflictFlag ?state-l = getConflictFlag ?state-euip ∧
  getConflictClause ?state-l = getConflictClause ?state-euip
  ∧
  getF ?state-l = (if getC ?state-euip = [opposite $?l'] then
    getF ?state-euip
  else
    (getF ?state-euip @ [getC ?state-euip]))
  )
  using applyLearnPreservedVariables[of ?state-euip]
  by (simp add: Let-def)

have $inv' ?state-l
proof
  have InvariantConflictFlagCharacterization (getConflictFlag ?state-l) (getF ?state-l) (getM ?state-l)
    using ⟨inv? ?state-euip⟩
    using ⟨getConflictFlag ?state-euip⟩
    using InvariantConflictFlagCharacterizationAfterApplyLearn[of ?state-euip]
    by (simp add: Let-def)
  moreover
  hence InvariantQCharacterization (getConflictFlag ?state-l)
    (getQ ?state-l) (getF ?state-l) (getM ?state-l)
    using ⟨inv? ?state-euip⟩
    using ⟨getConflictFlag ?state-euip⟩
    using InvariantQCharacterizationAfterApplyLearn[of ?state-euip]
    by (simp add: Let-def)
  moreover
  have InvariantUniqQ (getQ ?state-l)
    using ⟨inv? ?state-euip⟩
    using InvariantUniqQAfterApplyLearn[of ?state-euip]
    by (simp add: Let-def)
  moreover
  have InvariantConflictClauseCharacterization (getConflictFlag ?state-l)
    (getConflictClause ?state-l) (getF ?state-l) (getM ?state-l)
    using ⟨inv? ?state-euip⟩
    using ⟨getConflictFlag ?state-euip⟩
    using InvariantConflictClauseCharacterizationAfterApplyLearn[of ?state-euip]
    by (simp only: Let-def)
  ultimately
  show ?thesis
    using ⟨inv? ?state-euip⟩
    using ⟨getConflictFlag ?state-euip⟩
    using ⟨InvariantUniqC (getC ?state-euip)⟩
    using ⟨InvariantClCharacterization (getCl ?state-euip)⟩
    using ⟨isUIP (opposite (getCl ?state-euip))⟩
    (getM ?state-euip)
    using WatchInvariantsAfterApplyLearn[of ?state-euip]
    using $
    by (auto simp only: Let-def)
  qed

  have InvariantNoDecisionsWhenConflict (getF ?state-euip) (getM ?state-l) (currentLevel (getM ?state-l))
    InvariantNoDecisionsWhenUnit (getF ?state-euip) (getM ?state-l) (currentLevel (getM ?state-l))
    InvariantNoDecisionsWhenConflict [getC ?state-euip] (getM ?state-l)

using InvariantNoDecisionsWhenConflictNorUnitAfterApplyLearn[of ?state-euip]
using ⟨?inv′ ?state-euip⟩
using ⟨?inv′′ ?state-euip⟩
using ⟨getConflictFlag ?state-euip⟩
using ⟨InvariantUniqC (getM ?state-euip)⟩

using ⟨InvariantClCharacterization (getCl ?state-euip) (getM ?state-euip)⟩
using ⟨currentLevel (getM ?state-euip) > 0⟩
by (auto simp only: Let-def)

have isUIP (opposite (getCl ?state-l)) (getC ?state-l) (getM ?state-l)

using ⟨isUIP (opposite (getCl ?state-l)) (getC ?state-l)⟩
by simp

have InvariantClCurrentLevel (getCl ?state-l) (getM ?state-l)

using ⟨InvariantClCurrentLevel (getCl ?state-euip) (getM ?state-euip)⟩
by simp

have InvariantCEntailed (getConflictFlag ?state-l) F0’ (getC ?state-l)

using ⟨InvariantCEntailed (getConflictFlag ?state-euip) F0’⟩
by simp

have InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l) (getC ?state-l)

using ⟨InvariantCFalse (getConflictFlag ?state-euip)⟩
by simp

712
have InvariantUniqC (getC ?state-l)
  using ⟨InvariantUniqC (getC ?state-euip)⟩
  using $ by simp

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)

have InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)
  using ⟨InvariantClCharacterization (getCl ?state-euip) (getC ?state-euip)⟩
  unfolding applyLearn-def
  unfolding setWatch1-def
  unfolding setWatch2-def
  by (auto simp add: Let-def)
using $\langle$InvariantCFalse (getConflictFlag ?state-euip) (getM ?state-euip) (getC ?state-euip)$\rangle$
using $\langle$getConflictFlag ?state-euip$\rangle$
using InvariantVarsFAfterApplyLearn[of ?state-euip F0 Vbl]
by auto

have getConflictFlag ?state-l
  using (getConflictFlag ?state-euip)
  using $\langle$getConflictFlag ?state-euip$\rangle$
  using simp

have getSATFlag ?state-l = getSATFlag state
  using (getSATFlag ?state-euip = getSATFlag state)
  unfolding applyLearn-def
  unfolding setWatch2-def
  unfolding setWatch1-def
  by (simp add: Let-def)

have currentLevel (getM ?state-l) > 0
  using (currentLevel (getM ?state-euip) > 0)
  using $\langle$getConflictFlag ?state-euip = getM state$\rangle$
  by simp

have getM ?state-l = getM state $\lor$ (?state-l, state) $\in$ terminationLessState1 (vars F0 $\cup$ Vbl)
proof (cases getM ?state-euip = getM state)
case True
  thus $?thesis
  using $\langle$getConflictFlag ?state-euip = getM state$\rangle$
  by simp
next
case False
with (getM ?state-euip = getM state $\lor$ (?state-euip, state) $\in$ terminationLessState1 (vars F0 $\cup$ Vbl))
have (?state-euip, state) $\in$ terminationLessState1 (vars F0 $\cup$ Vbl)
  by simp
hence (?state-l, state) $\in$ terminationLessState1 (vars F0 $\cup$ Vbl)
using $\langle$getConflictFlag ?state-l = getSATFlag state$\rangle$
using (getSATFlag ?state-euip = getSATFlag state)
unfolding terminationLessState1-def
unfolding satFlagLessState-def
unfolding lexLessState1-def
unfolding lexLessRestricted-def
by (simp add: Let-def)
thus \( ?\text{thesis} \)
by simp
qed

let \( ?\text{state-bj} = \text{applyBackjump} ?\text{state-l} \)

have \( ?\text{inv}' ?\text{state-bj} \land \)
\( \text{InvariantVarsM} (\text{getM} ?\text{state-bj}) F0 Vbl \land \)
\( \text{InvariantVarsQ} (\text{getQ} ?\text{state-bj}) F0 Vbl \land \)
\( \text{InvariantVarsF} (\text{getF} ?\text{state-bj}) F0 Vbl \)

proof (cases \( \text{getC} ?\text{state-l} = \text{[opposite ?l']} \))
  case True
  thus \( ?\text{thesis} \)
  using \( \text{WatchInvariantsAfterApplyBackjump} [\text{of} ?\text{state-l} F0'] \)
  using \( \text{InvariantUniqAfterApplyBackjump} [\text{of} ?\text{state-l} F0'] \)
  using \( \text{InvariantConsistentAfterApplyBackjump} [\text{of} ?\text{state-l} F0'] \)

  using \( \text{InvariantQCharacterizationAfterApplyBackjump-1} [\text{of} ?\text{state-l} F0'] \)
  using \( \text{InvariantConflictFlagCharacterizationAfterApplyBackjump-1} [\text{of} ?\text{state-l} F0'] \)
  using \( \text{InvariantConflictClauseCharacterizationAfterApplyBackjump} [\text{of} ?\text{state-l}] \)
  using \( \text{InvariantVarsAfterApplyBackjump} [\text{of} ?\text{state-l} F0' F0 Vbl] \)

  using \( ?\text{inv}' ?\text{state-bj} \)
  using \( \text{getConflictFlag} ?\text{state-l} \)
  using \( \text{InvariantClCurrentLevel} (\text{getCl} ?\text{state-l}) (\text{getM} ?\text{state-l}) \)
  using \( \text{InvariantClC} \) (\text{getC} ?\text{state-l}) \)
  using \( \text{InvariantCFalse} (\text{getConflictFlag} ?\text{state-l}) (\text{getM} ?\text{state-l}) (\text{getC} ?\text{state-l}) \)
  using \( \text{InvariantCEntailed} (\text{getConflictFlag} ?\text{state-l} F0') (\text{getC} ?\text{state-l}) \)

  using \( \text{InvariantCICharacterization} (\text{getCl} ?\text{state-l}) (\text{getM} ?\text{state-l}) \)
  using \( \text{InvariantClCharacterization} (\text{getCl} ?\text{state-l}) (\text{getC} ?\text{state-l}) \)
  using \( \text{isUIP} (\text{opposite} (\text{getCl} ?\text{state-l})) (\text{getC} ?\text{state-l}) (\text{getM} ?\text{state-l}) \)

  using \( \text{currentLevel} (\text{getM} ?\text{state-l}) > 0 \)
  using \( \text{InvariantNoDecisionsWhenConflict} (\text{getF} ?\text{state-euip}) (\text{getM} ?\text{state-l}) (\text{currentLevel} (\text{getM} ?\text{state-l})) \)
  using \( \text{InvariantNoDecisionsWhenUnit} (\text{getF} ?\text{state-euip}) (\text{getM} ?\text{state-l}) (\text{currentLevel} (\text{getM} ?\text{state-l})) \)
  using \( \text{InvariantEquivalentZL} (\text{getF} ?\text{state-l}) (\text{getM} ?\text{state-l}) F0' \)

715
next
  case False
  thus thesis
    using WatchInvariantsAfterApplyBackjump[of ?state-l F0']
    using InvariantUniqAfterApplyBackjump[of ?state-l F0']
    using InvariantConsistentAfterApplyBackjump[of ?state-l F0']
    using invariantQCharacterizationAfterApplyBackjump-2[of ?state-l F0']
    using InvariantConflictFlagCharacterizationAfterApplyBackjump-2[of ?state-l F0']
    using InvariantUniqQAfterApplyBackjump[of ?state-l F0']
    using InvariantConflictClauseCharacterizationAfterApplyBackjump[of ?state-l]
    using InvariantsVarsAfterApplyBackjump[of ?state-l F0' F0 Vbl]
  using (?inv' ?state-l)
  using (getConflictFlag ?state-l)
    using InvariantCICurrentLevel (getCl ?state-l) (getM ?state-l)
    using InvariantUniqC (getC ?state-l)
    using InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l)
    using InvariantCEntailed (getConflictFlag ?state-l) F0' (getC ?state-l)
    using InvariantCICharacterization (getCl ?state-l) (getC ?state-l)
    using InvariantClCharacterization (getCl ?state-l) (getC ?state-l)
    using isUIP (opposite (getC ?state-l)) (getC ?state-l) (getM ?state-l)
    using (currentLevel (getM ?state-l) > 0)
  using InvariantNoDecisionsWhenConflict (getF ?state-euip) (getM ?state-l) (currentLevel (getM ?state-l))
    using InvariantNoDecisionsWhenUnit (getF ?state-euip) (getM ?state-l) (currentLevel (getM ?state-l))
    using $
using \( \text{InvariantVarsM} \ (\text{getM} \ ?\text{state-l}) \ F0' \ Vbl \)

using \( \text{InvariantVarsQ} \ (\text{getQ} \ ?\text{state-l}) \ F0' \ Vbl \)

using \( \text{InvariantVarsF} \ (\text{getF} \ ?\text{state-l}) \ F0' \ Vbl \)

using \( \text{vars} \ F0' \subseteq \text{vars} \ F0 \)

by (simp add: Let-def)

qed

have \(?\text{inv''} \ ?\text{state-bj} \)

proof (cases \(\text{getC} \ ?\text{state-l} = \text{[\em opposite} \ ?\text{!']}\))

  case True
  thus ?thesis using InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-1[of \(?\text{state-l} \ F0'\)]

    using (?\text{inv'} \ ?\text{state-l})

    using \(\text{getConflictFlag} \ ?\text{state-l} \)

    using \(\text{InvariantClCurrentLevel} \ (\text{getCl} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \)

    using \(\text{InvariantUniqC} \ (\text{getC} \ ?\text{state-l}) \)

    using \(\text{InvariantCFalse} \ (\text{getConflictFlag} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \)

    using \(\text{InvariantCEntailed} \ (\text{getConflictFlag} \ ?\text{state-l}) \ F0' \)

    using \(\text{InvariantClCharacterization} \ (\text{getCl} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \)

    using \(\text{InvariantCllCharacterization} \ (\text{getCl} \ ?\text{state-l}) \ (\text{getCl} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \)

    using \(\text{isUIP} \ (\text{opposite} \ (\text{getCl} \ ?\text{state-l})) \ (\text{getC} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \)

    using \(\text{currentLevel} \ (\text{getM} \ ?\text{state-l}) > 0 \)

    using \(\text{InvariantNoDecisionsWhenConflict} \ (\text{getF} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-l}) \ (\text{currentLevel} \ (\text{getM} \ ?\text{state-l})) \)

    using \(\text{InvariantNoDecisionsWhenUnit} \ (\text{getF} \ ?\text{state-euip}) \ (\text{getM} \ ?\text{state-l}) \ (\text{currentLevel} \ (\text{getM} \ ?\text{state-l})) \)

    using $

    by (simp add: Let-def)

next

  case False

  thus ?thesis using InvariantsNoDecisionsWhenConflictNorUnitAfterApplyBackjump-2[of \(?\text{state-l} \)]

    using (?\text{inv'} \ ?\text{state-l})

    using \(\text{getConflictFlag} \ ?\text{state-l} \)

    using \(\text{InvariantClCurrentLevel} \ (\text{getCl} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \)

    using \(\text{InvariantCFalse} \ (\text{getConflictFlag} \ ?\text{state-l}) \ (\text{getM} \ ?\text{state-l}) \)

    using \(\text{InvariantUniqC} \ (\text{getC} \ ?\text{state-l}) \)

    using \(\text{InvariantCEntailed} \ (\text{getConflictFlag} \ ?\text{state-l}) \ F0' \)

(\text{getC} \ ?\text{state-l})

717
using ⟨InvariantClCharacterization (getCl ?state-l) (getC ?state-l)) (getM ?state-l)⟩
using ⟨InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)⟩
using ⟨isUIP (opposite (getCl ?state-l)) (getM ?state-l)⟩
using ⟨currentLevel (getM ?state-l) > 0⟩
using ⟨InvariantNoDecisionsWhenConflict (getF ?state-eup) (getM ?state-l)⟩
using ⟨InvariantNoDecisionsWhenUnit (getF ?state-eup) (getM ?state-l)⟩
using ⟨InvariantNoDecisionsWhenConflict [getC ?state-eup] (getBackjumpLevel ?state-l)⟩
using ⟨InvariantNoDecisionsWhenUnit [getC ?state-eup] (getM ?state-l)⟩
using $by (simp add: Let-def)

have getBackjumpLevel ?state-l > 0 → (getF ?state-l) ≠ [] ∧
(last (getF ?state-l) = (getC ?state-l))
proof (cases getC ?state-l = [opposite ?l′'])
case True
thus ?thesis
  unfolding getBackjumpLevel-def
  by simp
next
case False
thus ?thesis
  using $by simp
qed

using ⟨?inv' ?state-l⟩
using ⟨isConflictFlag ?state-l⟩
using ⟨isUIP (opposite (getCl ?state-l)) (getM ?state-l) (getM ?state-l)⟩
using ⟨InvariantClCurrentLevel (getCl ?state-l) (getM ?state-l)⟩
using ⟨InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l) (getC ?state-l)⟩
using ⟨InvariantUniqC (getC ?state-l)⟩
using ⟨InvariantClCharacterization (getCl ?state-l) (getC ?state-l)⟩

718
\( \text{?action} \) (getM ?state-l)


**using**: currentLevel (getM ?state-l) > 0

**using**: InvariantGetReasonIsReasonAfterApplyBackjump[of ?state-l F0’]

**by**: (simp only: Let-def)

\( \text{have} \) InvariantEquivalentZL (getF ?state-l) (getM ?state-l)

**F0’**

**using**: InvariantEquivalentZL (getF ?state-l) (getM ?state-l)

**F0’’**

**using**: currentLevel (getM ?state-l) > 0

**by**: (simp only: Let-def)

**have**: getSATFlag ?state-bj = getSATFlag state

**using**: currentLevel (getM ?state-l) > 0

**by**: (simp only: Let-def)

**let**: ?level = getBackjumpLevel ?state-l

**let**: ?prefix = prefixToLevel ?level (getM ?state-l)

**let**: ?l = opposite (getCl ?state-l)

**have**: isMinimalBackjumpLevel (getBackjumpLevel ?state-l) (opposite (getCl ?state-l)) (getC ?state-l) (getM ?state-l)

**using**: isMinimalBackjumpLevelGetBackjumpLevel[of ?state-l]

**using**: currentLevel (getM ?state-l) > 0


**using**: InvariantCEntailed (getConflictFlag ?state-l) F0’ (getC ?state-l)

**using**: InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l)


**using**: InvariantClCharacterization (getCl ?state-l) (getM ?state-l)

**using**: InvariantClCharacterization (getCl ?state-l) (getM ?state-l)

719
using ⟨InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l) (getC ?state-l))⟩
using ⟨InvariantUniqC (getC ?state-l)⟩
using ⟨InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)⟩
using ⟨isUIP (opposite (getCl ?state-l)) (getC ?state-l) (getM ?state-l)⟩
using ⟨getConflictFlag ?state-l⟩
using ⟨currentLevel (getM ?state-l) > 0⟩
by (simp add: Let-def)


unfolding isMininalBackjumpLevel-def
unfolding isBackjumpLevel-def
by simp

hence getBackjumpLevel ?state-l < currentLevel (getM ?state-l)

by simp

hence (?state-bj, ?state-l) ∈ terminationLessState1 (vars F0 ∪ Vbl)

using applyBackjumpEffect[of ?state-l F0]
using (?inv’ ?state-l)
using (getConflictFlag ?state-l)
using (isUIP (opposite (getCl ?state-l)) (getC ?state-l) (getM ?state-l))
using ⟨InvariantClCurrentLevel (getCl ?state-l) (getM ?state-l)⟩
using ⟨InvariantCEntailed (getConflictFlag ?state-l) F0’ (getC ?state-l)⟩

using ⟨InvariantCFalse (getConflictFlag ?state-l) (getM ?state-l) (getC ?state-l)⟩
using ⟨InvariantUniqC (getC ?state-l)⟩
using ⟨InvariantClCharacterization (getCl ?state-l) (getC ?state-l) (getM ?state-l)⟩
using ⟨currentLevel (getM ?state-l) > 0⟩
using (getSATFlag ?state-bj = getSATFlag state)
using (getSATFlag ?state-l = getSATFlag state)
using (getSATFlag state = UNDEF)
using (?inv’ ?state-l)
using ⟨InvariantVarsM (getM ?state-l) F0 Vbl⟩
using (?inv’ ?state-bj ∧InvariantVarsM (getM ?state-bj) F0 Vbl ∧
Vbl ∧
InvariantVarsQ (getQ ?state-bj) F0 Vbl ∧
InvariantVarsF (getF ?state-bj) F0 Vbl)
unfolding InvariantConsistent-def
unfolding InvariantUniq-def
unfolding InvariantVarsM-def
unfolding terminationLessState1-def
unfolding satFlagLessState-def
unfolding lexLessState1-def
unfolding lexLessRestricted-def
by (simp add: Let-def)
hence (?state-bj, state) ∈ terminationLessState1 (vars F0 ∪ Vbl)
  using (getM ?state-l = getM state ∨ (?state-l, state) ∈ terminationLessState1 (vars F0 ∪ Vbl))
  using (getSATFlag state = UNDEF)
  using (getSATFlag ?state-bj = getSATFlag state)
  using (getSATFlag ?state-l = getSATFlag state)
  using transTerminationLessState1I[of ?state-bj ?state-l vars F0 ∪ Vbl state]
unfolding terminationLessState1-def
unfolding satFlagLessState-def
unfolding lexLessState1-def
unfolding lexLessRestricted-def
by auto

show ?thesis
  using (?inv' ?state-bj ∧ InvariantVarsM (getM ?state-bj) F0 Vbl ∧
  InvariantVarsQ (getQ ?state-bj) F0 Vbl ∧
  InvariantVarsF (getF ?state-bj) F0 Vbl)
  using (?inv'' ?state-bj)
  using (InvariantEquivalentZL (getF ?state-bj) (getM ?state-bj) F0'')
  using (InvariantGetReasonIsReason (getReason ?state-bj)
  (getF ?state-bj) (getM ?state-bj) (set (getQ ?state-bj)))
  using (getSATFlag state = UNDEF)
  using (getSATFlag ?state-bj = getSATFlag state)
  using (getConflictFlag ?state-up)
  using (currentLevel (getM ?state-up) ≠ 0)
  using (?state-bj, state) ∈ terminationLessState1 (vars F0 ∪ Vbl)
unfolding solve-loop-body-def
by (auto simp add: Let-def)
qed
qed
next
case False
show ?thesis
proof (cases vars (elements (getM ?state-up)) ⊇ Vbl)
case True
  hence satisfiable F0'
using soundnessForSat \{ F0' \} Vbl getF ?state-up getM ?state-up
using \{ InvariantEquivalentZL (getF ?state-up) (getM ?state-up) \} F0'
using \{ inv' ?state-up \}
using \{ InvariantVarsF (getF ?state-up) F0 Vbl \}
using (\neg getConflictFlag ?state-up)
using \{ vars F0 \subseteq Vbl \}
using \{ vars F0' \subseteq vars F0 \}
using True
unfolding InvariantConflictFlagCharacterization-def
unfolding satisfiable-def
unfolding InvariantVarsF-def
by blast
moreover
let \( ?\text{state}' = \text{state-up} \) \{ getSATFlag := TRUE \}
have \( (\text{state'}, \text{state}) \in \text{terminationLessState1} \) \{ vars F0 \cup Vbl \}
using (getSATFlag state = UNDEF)
unfolding terminationLessState1-def
unfolding satFlagLessState-def
by simp
ultimately
show \( \text{thesis} \)
using \{ vars (elements (getM ?state-d)) \supseteq Vbl \}
using \{ inv' ?state-up \}
using \{ inv' ?state-up \}
using \{ InvariantEquivalentZL (getF ?state-up) (getM ?state-up) \} F0'
using \{ InvariantVarsM (getM ?state-up) F0 Vbl \}
using \{ InvariantVarsQ (getQ ?state-up) F0 Vbl \}
using (\neg getConflictFlag ?state-up)
unfolding solve-loop-body-def
by (simp add: Let-def)
next
case False
let \( ?\text{literal} = \text{selectLiteral} \) ?state-up Vbl
let \( ?\text{state-d} = \text{applyDecide} \) ?state-up Vbl

have InvariantConsistent (getM ?state-d)
using InvariantConsistentAfterApplyDecide \{ of Vbl ?state-up \}
using False
using \{ inv' ?state-up \}
by (simp add: Let-def)
moreover
have InvariantUniq (getM ?state-d)
using InvariantUniqAfterApplyDecide \{ of Vbl ?state-up \}
using False
moreover
have InvariantEquivalentZL (getF ?state-d) (getM ?state-d) F0'
  using InvariantEquivalentZLAfterApplyDecide[of ?state-up F0'
Vbl]
  using (?inv' ?state-up)
  using InvariantEquivalentZL (getF ?state-up) (getM ?state-up)
F0'
by (simp add: Let-def)
moreover
have InvariantGetReasonIsReason (getReason ?state-d) (getF
?state-d) (getM ?state-d) (set (getQ ?state-d))
  using InvariantGetReasonIsReasonAfterApplyDecide[of Vbl
?state-up]
  using (?inv' ?state-up)
  using InvariantGetReasonIsReason (getReason ?state-up) (getF
?state-up) (getM ?state-up) (set (getQ ?state-up));
  using False
  using (¬ getConflictFlag ?state-up)
  using (getConflictFlag ?state-up ∨ getQ ?state-up = [])
by (simp add: Let-def)
moreover
have getSATFlag ?state-d = getSATFlag state
  unfolding applyDecide-def
  using (getSATFlag ?state-up = getSATFlag state);
  using assertLiteralEffect[of ?state-up selectLiteral ?state-up Vbl
True]
  using (?inv' ?state-up);
by (simp only: Let-def)
moreover
have InvariantVarsM (getM ?state-d) F0 Vbl
  InvariantVarsF (getF ?state-d) F0 Vbl
  InvariantVarsQ (getQ ?state-d) F0 Vbl
  using InvariantsVarsAfterApplyDecide[of Vbl ?state-up]
  using False
  using (?inv' ?state-up)
  using (¬ getConflictFlag ?state-up)
  using (getConflictFlag ?state-up ∨ getQ ?state-up = [])
  using InvariantVarsM (getM ?state-up) F0 Vbl;
  using InvariantVarsQ (getQ ?state-up) F0 Vbl;
  using InvariantVarsF (getF ?state-up) F0 Vbl;
  by (auto simp only: Let-def)
moreover
have (?state-d, ?state-up) ∈ terminationLessState1 (vars F0 ∪ Vbl)
  using (getSATFlag ?state-up = getSATFlag state);
  using assertLiteralEffect[of ?state-up selectLiteral ?state-up Vbl
True]
  using (?inv' ?state-up);
  using (InvariantVarsM (getM state) F0 Vbl)
using \(\langle\text{InvariantVarsM}\ (\text{get}\ M\ \text{?state-up})\ F0\ Vbl\rangle\)
using \(\langle\text{InvariantVarsM}\ (\text{get}\ M\ \text{?state-d})\ F0\ Vbl\rangle\)
using \(\langle\text{getSATFlag}\ \text{state} = \text{UNDEF}\rangle\)
using \(\langle\text{?inv'}\ \text{?state-up}\rangle\)
using \(\langle\text{InvariantConsistent}\ (\text{get}\ M\ \text{?state-d})\rangle\)
using \(\langle\text{InvariantUniq}\ (\text{get}\ M\ \text{?state-d})\rangle\)
using \(\text{lexLessAppend}\{\text{[(selectLiteral}\ \text{?state-up}\ Vbl,\ True)]}\text{getM}\ \text{?state-up}\]

\[\text{unfolding applyDecide-def}
\text{unfolding terminationLessState1-def}
\text{unfolding lexLessState1-def}
\text{unfolding lexLessRestricted-def}
\text{unfolding InvariantVarsM-def}
\text{unfolding InvariantUniq-def}
\text{unfolding InvariantConsistent-def}
\text{by (simp add: Let-def)}\]

\[\text{hence}\ (\text{?state-d, state})\in\text{terminationLessState1}\ (\text{vars}\ F0\ \cup\ Vbl)\]
\[\text{using}\ (\text{?state-up} = \text{state} \vee (\text{?state-up, state})\in\text{terminationLessState1}\ (\text{vars}\ F0\ \cup\ Vbl))\]
\[\text{using}\ \text{transTerminationLessState1I}\{\text{of}\ \text{?state-d}\ \text{?state-up}\ \text{vars}\ F0\ \cup\ Vbl\ \text{state}\}
\text{by auto}\]

\[\text{ultimately}\]
\[\text{show}\ \text{?thesis}\]
\[\text{using}\ (\text{?inv'}\ \text{?state-up})\]
\[\text{using}\ \langle\text{getSATFlag}\ \text{state} = \text{UNDEF}\rangle\]
\[\text{using}\ \langle\neg\ \text{getConflictFlag}\ \text{?state-up}\rangle\]
\[\text{using}\ \text{False}\]
\[\text{using}\ \text{WatchInvariantsAfterAssertLiteral}\{\text{of}\ \text{?state-up}\ \text{?literal}\ True\}
\][\text{using}\ \text{InvariantWatchCharacterizationAfterAssertLiteral}\{\text{of}\ \text{?state-up}\ \text{?literal}\ True\}
\[\text{using}\ \text{InvariantUniqQAfterAssertLiteral}\{\text{of}\ \text{?state-up}\ \text{?literal}\ True\}
\[\text{using}\ \text{assertLiteralEffect}\{\text{of}\ \text{?state-up}\ \text{?literal}\ True\}
\text{unfolding solve-loop-body-def}
\text{unfolding applyDecide-def}
\text{unfolding selectLiteral-def}
\text{by (simp add: Let-def)}\]
\[\text{qed}\]
\[\text{qed}\]
\[\text{qed}\]

\[\text{lemma}\ \text{SolveLoopTermination} :\]
assumes
InvariantConsistent (getM state)
InvariantUniq (getM state)
InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state) and
InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state) and
InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state) and
InvariantWatchListsUniq (getWatchList state) and
InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state) and
InvariantUniqQ (getQ state) and
InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state) and
InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state) and
InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state)) and
InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state)) and
InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state)) and
gSATFlag state = UNDEF → InvariantEquivalentZL (getF state) (getM state) F0′ and
InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state) and
finite Vbl
vars F0′ ⊆ vars F0
vars F0 ⊆ Vbl
InvariantVarsM (getM state) F0 Vbl
InvariantVarsQ (getQ state) F0 Vbl
InvariantVarsF (getF state) F0 Vbl
shows
solve-loop-dom state Vbl
using assms
proof (induct rule: wf-induct[of terminationLessState1 (vars F0 ∪ Vbl)])
case 1
thus ?case
using \finite Vbl
using \finiteVarsFormula[of F0]
using \wellFoundedTerminationLessState1[of vars F0 ∪ Vbl]
by simp
next
case (2 state')
note \ih = this
show \texttt{?case}

\textbf{proof (cases getSATFlag state' = UNDEF)}

\texttt{case False}

\texttt{show \texttt{?thesis}}

\texttt{apply (rule solve-loop-dom.intros)}

\texttt{using False by simp}

\texttt{next}

\texttt{case True}

\texttt{let \texttt{?state'' = solve-loop-body state' Vbl}}

\texttt{have \texttt{InvariantConsistent (getM ?state'')}}

\texttt{InvariantUniq (getM ?state'')}}

\texttt{InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')} \texttt{and}

\texttt{InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'')} \texttt{and}

\texttt{InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'')}

\texttt{(getWatch2 ?state'') (getM ?state'')} \texttt{and}

\texttt{InvariantWatchesDifferFromF (getWatchList ?state'') (getF ?state'') and}

\texttt{InvariantWatchListsUniq (getWatchList ?state'')} \texttt{and}

\texttt{InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'')}

\texttt{(getWatch2 ?state'') and}

\texttt{InvariantUniqQ (getQ ?state'')} \texttt{and}

\texttt{InvariantQCharacterization (getConflictFlag ?state'') (getQ ?state'')}

\texttt{(getF ?state'')} \texttt{and}

\texttt{InvariantConflictFlagCharacterization (getConflictFlag ?state'')}

\texttt{(getF ?state'')} \texttt{and}

\texttt{InvariantNoDecisionsWhenConflict (getF ?state'') (getM ?state'')}

\texttt{(currentLevel (getM ?state'')) and}

\texttt{InvariantNoDecisionsWhenUnit (getF ?state'') (getM ?state'') and}

\texttt{InvariantConflictClauseCharacterization (getConflictFlag ?state'')}

\texttt{(getConflictClause ?state'') (getF ?state'') (getM ?state'') and}

\texttt{InvariantGetReasonIsReason (getReason ?state'') (getF ?state'')}

\texttt{(getM ?state'')} \texttt{(set (getQ ?state''))}

\texttt{InvariantEquivalentZL (getF ?state'') (getM ?state'') F0'}

\texttt{InvariantVarsM (getM ?state'') F0 Vbl}

\texttt{InvariantVarsQ (getQ ?state'') F0 Vbl}

\texttt{InvariantVarsF (getF ?state'') F0 Vbl}

\texttt{getSATFlag ?state'" = FALSE \rightarrow \neg \text{satisfiable F0'}}

\texttt{getSATFlag ?state'" = TRUE \rightarrow \text{satisfiable F0'}}

\texttt{(?state'', state') \in \text{terminationLessState1 (vars F0 \cup Vbl)}}

\texttt{using \texttt{InvariantsAfterSolveLoopBody[of state' F0' Vbl F0]}}

\texttt{using \texttt{ih(2) \ ih(3) \ ih(4) \ ih(5) \ ih(6) \ ih(7) \ ih(8) \ ih(9) \ ih(10) \ ih(11) \ ih(12) \ ih(13) \ ih(14) \ ih(15) \ ih(16) \ ih(17) \ ih(18) \ ih(19) \ ih(20) \ ih(21) \ ih(22) \ ih(23)}}

\texttt{using True
by (auto simp only: Let-def)
hence solve-loop-dom ?state'' Vbl
  using sh
by auto
thus ?thesis
  using solve-loop-dom.intros[of state' Vbl]
  using True
by simp
qed
qed

lemma SATFlagAfterSolveLoop:
assumes
  solve-loop-dom state Vbl
  InvariantConsistent (getM state)
  InvariantUniq (getM state)
  InvariantWatchesEl (getF state) (getWatch1 state) (getWatch2 state)
and
  InvariantWatchesDiffer (getF state) (getWatch1 state) (getWatch2 state)
  and
  InvariantWatchCharacterization (getF state) (getWatch1 state) (getWatch2 state) (getM state)
  and
  InvariantWatchListsContainOnlyClausesFromF (getWatchList state) (getF state)
  and
  InvariantWatchListsUniq (getWatchList state)
  and
  InvariantWatchListsCharacterization (getWatchList state) (getWatch1 state) (getWatch2 state)
  and
  InvariantUniqQ (getQ state)
  and
  InvariantQCharacterization (getConflictFlag state) (getQ state) (getF state) (getM state)
  and
  InvariantConflictFlagCharacterization (getConflictFlag state) (getF state) (getM state)
  and
  InvariantNoDecisionsWhenConflict (getF state) (getM state) (currentLevel (getM state))
  and
  InvariantNoDecisionsWhenUnit (getF state) (getM state) (currentLevel (getM state))
  and
  InvariantGetReasonIsReason (getReason state) (getF state) (getM state) (set (getQ state))
  and
  getSATFlag state = UNDEF → InvariantEquivalentZL (getF state) (getM state) F0'
  and
  InvariantConflictClauseCharacterization (getConflictFlag state) (getConflictClause state) (getF state) (getM state)
  and
  getSATFlag state = FALSE → ¬ satisfiable F0'
  and
  getSATFlag state = TRUE → satisfiable F0'
finite Vbl
vars F0' ⊆ vars F0
vars F0 ⊆ Vbl
InvariantVarsM (getM state) F0 Vbl

728
\[\text{InvariantVarsF} \ (\text{getF state}) \ F0 \ Vbl\]
\[\text{InvariantVarsQ} \ (\text{getQ state}) \ F0 \ Vbl\]

\textbf{shows}

let state' = solve-loop state Vbl in
\((\text{getSATFlag state'} = \text{FALSE} \land \neg \text{satisfiable F0'}) \lor (\text{getSATFlag state'} = \text{TRUE} \land \text{satisfiable F0'})\)

\textbf{using} assms

\textbf{proof} (induct state Vbl rule: solve-loop-dom.induct)

\textbf{case} (step state' Vbl)

\textbf{note} ih = this

\textbf{show} \ ?case

\textbf{proof} (cases getSATFlag state' = \text{UNDEF})

\textbf{case} False

\textbf{with} solve-loop.simps[of state']

\textbf{have} solve-loop state' Vbl = state'

\textbf{by} simp

\textbf{thus} \ ?thesis

\textbf{using} False

\textbf{using} ih(19) ih(20)

\textbf{using} ExtendedBool.nchotomy

\textbf{by} (auto simp add: Let-def)

\textbf{next}

\textbf{case} True

let ?state'' = solve-loop-body state' Vbl

\textbf{have} solve-loop state' Vbl = solve-loop ?state'' Vbl

\textbf{using} solve-loop.simps[of state']

\textbf{using} True

\textbf{by} (simp add: Let-def)

\textbf{moreover}

\textbf{have} InvariantEquivalentZL (getF state') (getM state') F0'

\textbf{using} True

\textbf{using} ih(17)

\textbf{by} simp

\textbf{hence}

InvariantConsistent (getM ?state'')
InvariantUniq (getM ?state'')
InvariantWatchesEl (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') and
InvariantWatchesDiffer (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') and
InvariantWatchCharacterization (getF ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getM ?state'') and
InvariantWatchListsContainOnlyClausesFromF (getWatchList ?state'') (getF ?state'') and
InvariantWatchListsUniq (getWatchList ?state'') and
InvariantWatchListsCharacterization (getWatchList ?state'') (getWatch1 ?state'') (getWatch2 ?state'') (getM ?state'') and
InvariantUniqQ (getQ ?state'') and
InvariantQCharacterization (getConflictFlag ?state'') (getQ ?state'')

729
8.2 Total correctness theorem

theorem correctness:
shows
(solve F0 = TRUE ∧ satisfiable F0) ∨ (solve F0 = FALSE ∧ ¬ satisfiable F0)
proof—
let \(?istate = initialize F0 \) initial\(\text{State} \)

\[ \text{let } \?F0' = \text{filter } (\lambda \ c. \ \neg \text{clauseTautology } c) \ F0 \]

\text{have}

\( \text{InvariantConsistent } (\text{getM } \?istate) \)

\( \text{InvariantUniq } (\text{getM } \?istate) \)

\( \text{InvariantWatchesEl } (\text{getF } \?istate) \ (\text{getWatch1 } \?istate) \ (\text{getWatch2 } \?istate) \) \text{ and}

\( \text{InvariantWatchesDiffer } (\text{getF } \?istate) \ (\text{getWatch1 } \?istate) \ (\text{getWatch2 } \?istate) \) \text{ and}

\( \text{InvariantWatchCharacterization } (\text{getF } \?istate) \ (\text{getWatch1 } \?istate) \ (\text{getWatch2 } \?istate) \ (\text{getM } \?istate) \) \text{ and}

\( \text{InvariantWatchListsContainOnlyClausesFromF } (\text{getWatchList } \?istate) \) \text{ and}

\( \text{InvariantUniqQ } (\text{getQ } \?istate) \) \text{ and}

\( \text{InvariantQCharacterization } (\text{getConflictFlag } \?istate) \ (\text{getQ } \?istate) \) \text{ and}

\( \text{InvariantConflictFlagCharacterization } (\text{getConflictFlag } \?istate) \) \text{ and}

\( \text{InvariantNoDecisionsWhenConflict } (\text{getF } \?istate) \ (\text{getM } \?istate) \) \text{ and}

\( \text{InvariantNoDecisionsWhenUnit } (\text{getF } \?istate) \ (\text{getM } \?istate) \) \text{ and}

\( \text{InvariantGetReasonIsReason } (\text{getReason } \?istate) \) \text{ and}

\( \text{InvariantConflictClauseCharacterization } (\text{getConflictFlag } \?istate) \) \text{ and}

\( \text{InvariantVarsM } (\text{getM } \?istate) \) \text{ and}

\( \text{InvariantVarsQ } (\text{getQ } \?istate) \) \text{ and}

\( \text{InvariantVarsF } (\text{getF } \?istate) \) \text{ and}

\( \text{getSATFlag } \?istate = \text{UNDEF } \rightarrow \text{InvariantEquivalentZL } (\text{getF } \?istate) \) \text{ and}

\( \text{getSATFlag } \?istate = \text{FALSE } \rightarrow \neg \text{satisfiable } \?F0' \)

\( \text{getSATFlag } \?istate = \text{TRUE } \rightarrow \text{satisfiable } \?F0 \)

\text{using } \text{assms}

\text{using } \text{InvariantsAfterInitialization[of } \?F0 \]

\text{using } \text{InvariantEquivalentZLAfterInitialization[of } \?F0 \]

\text{unfolding } \text{InvariantVarsM-def}

\text{unfolding } \text{InvariantVarsF-def}

\text{unfolding } \text{InvariantVarsQ-def}

\text{by } (\text{auto simp add: Let-def})

\text{moreover}

\text{hence } \text{solve-loop-dom } \?istate \ (\text{vars } \F0) \)

\text{using } \text{SolveLoopTermination[of } \?istate \ ?F0' \ \text{vars } \F0 \ F0\]

\text{using } \text{finiteVarsFormula[of } \F0\]

\text{using } \text{varsSubsetFormula[of } \?F0' \ F0\]

\text{by } \text{auto}
ultimately

show ?thesis 
  using finiteVarsFormula[of $F_0$]
  using SATFlagAfterSolveLoop[of ?istate vars $F_0$ ?$F_0'$ $F_0$]
  using satisfiableFilterTautologies[of $F_0$]
  unfolding solve-def
  using varsSubsetFormula[of ?$F_0'$ $F_0$]
  by (auto simp add: Let-def)

qed

end

References